

RICE UNIVERSITY

A First-Class Approach to Genericity

by

Eric Ethan Allen

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

Keith Cooper
Professor of Computer Science

Michael Fagan
Faculty Fellow,
Computational and Applied Mathematics

Houston, Texas

April, 2003

A First-Class Approach to Genericity

Eric Ethan Allen

Abstract

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

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

What makes our treatment of first-class genericity particularly interesting and important is the fact that it can be added to the existing Java language without any change to the underlying Java Virtual Machine. Moreover, the extension is backward compatible with legacy Java source and class files. Although our discussion of implementation issues focuses on Java, the same implementation techniques could be

applied to other object-oriented languages such as C# or Eiffel that support incremental compilation, dynamic class loading, and a static type system with nominal subtyping.

For Lucy

Acknowledgments

In addition to his professional guidance, my advisor, Robert “Corky” Cartwright, has taught me much of what I know about effective software development, specification, debugging, programming languages, and the practice of scientific research. He has given me a great deal of opportunity to assist in teaching, in developing my own ideas, and in mentoring new graduate students. I am quite fortunate to have had him as an advisor, and I am very grateful for all of his help.

I also owe a great deal of gratitude to Matthias Felleisen for introducing me to the nature of rigorous software specification and analysis. The rest of Matthias’ team, particularly John Clements, Robby Findler, and Shriram Krishnamurthi, have been very strong influences during my graduate career. I am better off for having had the opportunity to work with them all.

My undergraduate research advisor, Claire Cardie, first introduced me to academic research, and the nature of thesis writing, as well as programming in the large.

I’ve also been quite privileged to work alongside the Rice JavaPLT team. In particular, I am glad to have had the chance to work with Brian Stoler, the former lead developer of DrJava. Our many enlightening conversations about the nature of software development have often influenced my own views. I am also very glad to have worked with Charles Reis, the current lead developer and soon to be Ph.D. student at the University of Washington, as well as Rice Ph.D. student Moez Abdel-Gawad. I have benefitted immensely from the skills and insights of all of these individuals, and I have no doubt that they will all be extremely successful in their future pursuits.

I am also glad to have had the opportunity to work with many of the other graduate students at Rice. Most notably, I am quite lucky to have had the opportunity

to work with Geraldine Morin. Her deep understanding of many areas of computer science and mathematics have often helped to bolster my own understanding. She has also provided me with extremely valuable advice on many occasions, and she has helped to keep me motivated and focused on tangible goals while in graduate school. I am grateful for the many engaging conversations I've had with Sameer Siruguri, and for the enjoyable debates we had through the Rice Blackboard Debate Society, which we formed together. Henrik Weimer, Alex Grosul, Kevin Charter, Scott Crosby, Sitaram Iyer, and Steve Miller have also helped to make my experience at Rice more fulfilling.

Many instructors at Rice have helped me to form a deeper understanding of computer science. In particular, Lydia Kaviraki, Kathi Fisler, Ian Barland, John Greiner, Dan Wallach, Keith Cooper, Walid Taha, and Devika Subramanian all have been very influential.

I'd also like to thank the members of my M.S. and Ph.D. thesis committees for their feedback and hard work: Dan Wallach, Scott Rixner, Corky Cartwright, Michael Fagan, and Keith Cooper.

There are also many undergraduates at Rice whom I have had the pleasure to work with. Most notably, Jonathan Bannet has been an extremely valuable member of the JavaPLT research group for two years. Jonathan has played an influential role both as a developer on the DrJava project, and as an assistant language designer and developer on the NextGen project. I've also enjoyed working with students Amy Egan, Chris Klick, and Luke Hoban.

Other old friends who have stayed in touch have often helped to provide essential diversions from my thesis studies, and to provide many invaluable and enlightening conversations: Jonathan Marvin, Julie Rathbun, Janine Joseph, and Josh Fruhlinger. Josh was also the copy editor for my book, "Bug Patterns", and the book is much better because of his efforts.

In addition, the editors for my IBM developerWorks column, Jenni Aloï, Kane

Scarlett, and Christine Stackel, have helped me to produce monthly articles while at graduate school, and have helped to improve the quality of those articles significantly. I'm very grateful to have had the opportunity to engage in this supplemental activity while pursuing my degree. Kane Scarlett was also the developmental editor for "Bug Patterns".

I'd like to thank many of the administrative assistants at Rice who play such an essential role in actually getting things done, and who lighten up the atmosphere at Rice in many ways: Iva Jean Jorgensen, Donna Jares, Rhonda Guajardo, Melissa Cisneros, and Darnell Price.

My colleagues and friends at Cycorp have played an important role in my growth as a software developer and a researcher: Mike Reimers, David Crabbe, Paul Strominger, Bjoern Aldag, Daniel Mahler, Steven DeVoy, Keith Goolsbey, Stefano Bertolo, Steven Reed, Ming Xu, Fritz Lehmann, Cindy Matuszek, Nancy Salay, and, of course, Doug Lenat.

Of course, successful completion of a degree requires that one is successfully engaged in activity after the degree. Therefore, I am very grateful to have the opportunity to begin a career as a researcher at Sun Microsystems following the completion of my Ph.D. This opportunity is due primarily to the efforts of Steve Heller, as well as Miram Kadansky, and Guy Steele.

My family has supported me in many ways throughout my entire education. My parents, Bruce and Edith Allen, instilled in me a strong sense of the value of education from an early age. I'd also like to thank my sister, Nichole Allen, for all of her support and encouragement. My grandparents, Charles and Esther Allen, helped to educate me in many ways, and many of my aunts and uncles, Craig Allen, Steve Allen, Mike Rejewski, Janice Rejewski, and Curtis Allen, have helped me both financially and morally, especially during rough times.

Finally, I'd like to thank my wife, Kori, for all of her advice, moral support, perspective, and understanding. A Ph.D. can be as hard on a spouse as it is on the

student who earns it, and I am lucky to have such an understanding wife. I'd also like to thank her for giving birth to our daughter, Lucy Jane Allen, who is two weeks old at the time of this writing. Most of all, I'd like to thank Lucy for being born.

Contents

Abstract	ii
Acknowledgments	v
List of Illustrations	xiii
List of Tables	xiv
1 Introduction	1
1.1 Background	3
1.1.1 Nominal versus Structural Subtyping	3
1.1.2 Generic Types	9
1.1.3 First-Class Generic Types	18
1.2 Roadmap	25
2 Motivation for First-Class Genericity	27
2.1 Why Support Type Dependent Operations	27
2.1.1 The Singleton Pattern	28
2.1.2 Casting and Catching Generic Types	30
2.1.3 Functional Mapping over Arrays	31
2.1.4 Legacy Classes and Interfaces	31
2.1.5 Transparent Debugging	33
2.2 Why Support Mixins	33
2.3 Conclusion	34
3 The Safe Instantiation Principle	35
3.1 Motivation	35

3.2	GJ, Type Erasure, and Safe Instantiation	36
3.3	NEXTGEN and Safe Instantiation	37
3.4	MIXGEN and Safe Instantiation	39
3.5	Conclusion	42
4	NextGen Compiler Design	43
4.1	Generic Classes	43
4.2	Polymorphic Methods	45
4.3	The GJ Implementation Scheme	45
4.4	Implications of Type Erasure in GJ	46
4.5	Restrictions on Generic Java Imposed by GJ	47
4.6	The NEXTGEN Implementation Scheme	47
4.7	NEXTGEN Architecture	48
4.7.1	Modeling Generic Types in a Class Hierarchy	48
4.7.2	Snippet Methods	53
4.7.3	Extensions of Generic Classes	54
4.7.4	Polymorphic Methods	54
5	Design Complications	57
6	NextGen Compiler Implementation	62
6.1	The NEXTGEN Class Loader	63
7	NextGen Performance	65
8	Implementing First-Class Genericity	74
8.1	Preliminaries	74
8.2	MIXGEN Extensions	75
8.3	Cyclic and Infinite Class Hierarchies	76

8.4	Accidental Overriding	80
8.5	Hygienic Mixins	82
8.5.1	MIXGEN and MIXEDJAVA	84
8.6	Implementing MIXGEN in the JVM	85
8.6.1	Enforcing Hygienic Method Invocation	86
8.6.2	Compiling with clauses	90
9	Core MixGen	91
9.1	Definitions	91
9.1.1	Syntax	92
9.1.2	Subtyping and Valid Class Tables	95
9.1.3	Type Checking	96
9.1.4	Well-formed Types and Class Definitions	97
9.1.5	Constructors and Methods	98
9.1.6	Expression Typing	100
9.1.7	Explicit Polymorphism	102
9.1.8	Fields and Field Values	102
9.1.9	Constructor Call Resolution	103
9.1.10	Computation	103
9.2	Proof of Type Soundness	104
9.2.1	Ground Expressions	105
9.2.2	Class Hierarchies	109
9.2.3	Preservation	111
9.2.4	Progress	118
9.2.5	Type Soundness	120
10	Conclusion	122
10.1	Future Research	123
10.1.1	Performance of Hygienic Method Lookup	123

10.1.2	Selective Covariance of Generic Types	123
10.1.3	Integration of Raw Types	124
A	The NextGen Implementation Code Structure	125
A.1	The NEXTGEN CVS Repository	126
A.2	The NEXTGEN Project Directory Structure	127
A.3	Ant Targets in the NEXTGEN Project	129
A.4	Releasing a New Version of NEXTGEN	131
A.5	NEXTGEN Package Design	131
A.5.1	ClassLoader Package Design	131
A.5.2	Compiler Package Design	131
A.6	Unit Tests in NEXTGEN	133
A.6.1	Class PrintableObject	134
A.6.2	Class NextGenTestCase	135
A.7	Phases of The NEXTGEN Compiler	136
A.7.1	Trees	138
A.7.2	Trees and PrintableObjects	138
A.7.3	Environments	141
A.7.4	AnalyzerContexts	142
A.7.5	Scopes and Entries	143
A.7.6	Symbols	146
A.7.7	SymbolTables and ClassReaders	147
A.7.8	CodeGenerators, GenContexts, and ClassWriters	148
B	A Sample Conversion of the NextGen Compiler	150
B.1	The Original Source Code	150
B.2	The Transformed Source Code	152
	Bibliography	157

Illustrations

1.1	An illustration of nominal vs. structural subtyping	4
1.2	A Simple Mixin Class	20
4.1	Naive Implementation of Generic Types Over the Existing Java Class Structure	48
4.2	Illegal Class Hierarchy in Naive JVM Class Representation	49
4.3	Simple Parametric Type Hierarchy and its JVM Class Representation	50
7.1	Performance Results for Sun 1.3 Client (in milliseconds)	65
7.2	Performance Results for Sun 1.3 Server (in milliseconds)	66
7.3	Performance Results for Sun 1.4 Client (in milliseconds)	67
7.4	Performance Results for Sun 1.4 Server (in milliseconds)	68
7.5	Performance Results for IBM 1.3 (in milliseconds)	69
8.1	A mixin with accidental overriding	81
8.2	An example of a mixin implementing two instantiations of the same interface.	89
A.1	Abstract Syntax Trees	138
A.2	Symbols	147

Tables

9.1	Core MixGen Syntax	93
9.2	Subtyping and Type Bounds	95
9.3	Well-formed Constructs	96
9.4	Constructors and Methods	99
9.5	Expression Typing	100
9.6	Fields and Field Values	102
9.7	Computation	103

Chapter 1

Introduction

Every motion shall be of such a kind that the Engine shall either break itself or stop itself or execute the intended motion.

Charles Babbage

Static type systems have been predominant in mainstream object-oriented languages, such as Eiffel, C++, Java, and C#, for over a decade. A well-designed type system provides the programmer with strong guarantees about the behavior of a program at run-time, thereby allowing a programmer to reason about his program more effectively. However, the theory of type systems developed by languages researchers has focused on a fundamentally different subtyping model than the one used in mainstream object-oriented languages. With a few notable exceptions [26, 28, 34, 38], the type systems studied by languages researchers are based on **structural subtyping** [44]. In contrast, mainstream object-oriented languages employ a formulation known as **nominal subtyping**.¹ These two models of subtyping have significantly different formal properties; technical results from one discipline do not necessarily hold for the other [44].

Nominal subtyping has been predominant in mainstream object-oriented languages because it offers significant advantages over structural subtyping. These advantages include:

- simple, intuitive, type-checking rules

¹These two forms of type system, and their differences, will be explained in Section 1.1.1.

- manifest type hierarchies
- transparent support for mutually recursive datatypes
- easily understood run-time error diagnostics

Nevertheless, the nominal type systems *currently* used in mainstream object-oriented languages are too simple to provide precise type checking. Although Java and C# are type-safe, the imprecision of their type systems still allows for many run-time type errors. Their most significant failing is the lack of support for generic (parameterized) types. This omission restricts the range of abstractions that programs can express and the precision of static type annotation and checking. The incorporation of a comprehensive generic type system would simplify the structure of many programs, eliminate the need for nearly all explicit type casts, and enable programmers to catch far more bugs at compile time through much more precise static type checking.

Both Java and C# will soon be extended to support generic types [37, 49], but neither of these extensions is what we call **first-class**. We define a *first-class* generic type system to be a type system in which the use of generic types (both type variables and parametric type instantiations) can be used in any context where an ordinary type can occur.

The forthcoming JSR-14 extension for Java is second-class; it provides no support for operations that require generic type information at run-time. In contrast, C# will support operations that depend on run-time generic type information, but it still imposes some restrictions on the use of generic types. In particular, it prohibits the superclass of a generic class from being a “naked” type parameter—precluding the definition of generic classes of the form `class C<T> extends T`. Such generic classes are called **mixins**. *Mixins* are classes parameterized by their parent type, or, equivalently, functions mapping classes to new subclasses. They are an important form of object-oriented abstraction that has been supported in various object systems for Lisp and

some research languages, but not in a mainstream programming language other than the crude macro-based implementation in C++. Additionally, because mixins can be viewed as parameterized classes, they are naturally related to generic types. However, this relationship has not been formally analyzed previously.

In this thesis, we show how to extend object-oriented languages with nominal subtyping to support full first-class genericity including mixins. In the process, we prove that the resulting type system is sound for a core subset of Java with generic types. Moreover, in the case of Java, we show how this extension can be efficiently implemented on top of the existing Java Virtual Machine while retaining compatibility with legacy binaries. Although our discussion focuses on Java, the same analysis could be applied to other object-oriented languages that support incremental compilation, dynamic class loading, and a static type system with nominal subtyping.

1.1 Background

Before we proceed to discuss first-class genericity, we need to explain some of the key concepts underlying genericity and subtyping. In this section, we will review these concepts and discuss related work.

1.1.1 Nominal versus Structural Subtyping

In object-oriented languages with *structural subtyping*, objects are formally modeled as untagged records where a record type τ_1 is a subtype of record type τ_2 iff τ_1 includes all of the member names of τ_2 and the type of each member in τ_2 is a supertype of the corresponding member of τ_1 . There is no relationship between subtyping and class inheritance because the subtyping relationship between two record types depends only on the signatures of the members of the two types.

In object-oriented languages with nominal subtyping, objects are modeled as tagged records where the tag is typically the name of the class to which the object belongs. In such type systems, a class **A** is a subtype of a class **B** iff **A** inherits members from **B**. A class **C** with exactly the same members as **B** that does not inherit

```

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

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

```

Figure 1.1 : An illustration of nominal vs. structural subtyping

from `B` is not a subtype of `B`. Since inheritance relationships are declared by the programmer, the programmer explicitly determines whether a given type is a subtype of another.

To illustrate the difference between nominal and structural subtyping, consider the Java program fragment in Figure 1.1.1. Because Java is based on nominal subtyping, the two abstract classes `Set` and `MultiSet` have no subtyping relationship: no value of type `Set` (other than the degenerate value `null`) is a value of the type `MultiSet` and vice-versa. This distinction is appropriate for the abstract types `Set` and `MultiSet` because the contracts for their methods are incompatible.

In contrast, suppose that Java were structurally subtyped. Then `Set` would be a subtype of `MultiSet` and vice versa because the two classes have identical public member signatures. Structural subtyping forces subtyping relationships on the basis of matching method signatures *even when the method contracts conflict*.

Nominal Subtyping and Run-time Type Representations

Nominal subtyping has a major impact on the design of the associated object-oriented programming languages. Since class types are embodied as run-time type tags, it is natural to treat types as predicates that can be queried at run-time and to support

operations that explicitly depend on these tags such as casts and class membership tests. In this regard, nominally subtyped object-oriented languages are closer to dynamically typed languages than they are to statically typed languages with structural subtyping. We call operations that depend on types at run-time **type dependent**.

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

Nominal Subtyping and Recursive Data Types

Languages based on nominal subtyping are able to support recursive, and even mutually recursive, types much more easily than languages with structural subtyping. For example, a class `Tree` representing binary trees can make references to itself inside its own definition, and this self-reference causes no complications in the underlying type system. Nominal languages can easily even support **mutually recursive** datatypes, where multiple classes make reference to each other in their respective definitions. We enjoy this advantage in nominal languages because the type names used for determining subtyping provide an extra level of indirection. In nominal languages, self-references are simply references to a type name, and can be handled just like references to any other type name. Recursive types in structural languages must be defined via fixed point operators, which introduce many complications, particularly in the case of mutual recursion [44].

Subtyping and Inheritance

Luca Cardelli first proposed the identification of subtyping with inheritance in [18]. However, rather than defining subtyping via an explicit inheritance hierarchy, Cardelli

proposed explaining inheritance *as* structural subtyping, ignoring the fact that the class hierarchy is an explicitly defined system in most object-oriented languages.

In his doctoral thesis, Cook criticized Cardelli's characterization of inheritance as subtyping [22]:

The types that are related by Cardelli's subtype scheme are not analogous to the classes in object-oriented programming. A record type can define only the format of instances, specifying the names and parameter types of function that might represent methods. A class is a mechanism for constructing objects and providing them with similar behavior. Though a type can describe the instances of a class, it is not a class itself.

As an alternative to Cardelli's explanation of inheritance, Cook presented a notion of inheritance as a constructive mechanism for assembling a program. He contrasted it with a (structural) subtyping relation, but he did not suggest defining subtyping via the inheritance hierarchy.

Nominal Systems in Mainstream Languages

Meanwhile, mainstream object-oriented languages such as Eiffel and C++ began to include static type systems where subtyping was defined directly in terms of the class hierarchy. Unfortunately, the resulting type systems were not carefully analyzed, resulting in many insidious errors in language designs. In many cases, these errors were discovered by subsequent analyses of languages researchers. For example, Eiffel, as defined by Bertrand Meyer in [39], contained several flaws in the type system, which were discussed by Cook in [23]. Most notably, Eiffel allowed covariant method parameter types, a feature that is now well known to break type soundness.

In contrast, C++, which also supports nominal subtyping, was deliberately designed with an unsound type system [50]. One key reason for this design decision was that the type system supported was not strong enough to encode many of the design patterns needed to support production programming.² But the result has

²As will be discussed in the section 1.1.2, the C++ template facility is used to write generic classes, but template classes are not type-checked independently of their instantiations, allowing for various run-time errors that a generic type system should prevent. The C++ type system is defined only for generic instantiations and non-generic classes; hence it does not truly support generic types.

been that programs written in this language are notoriously error-prone. Their lack of robustness has no doubt contributed to a bias against nominally subtyped languages by the languages research community.

Research Analyses of Nominal Subtyping

Gilad Bracha, in his doctoral thesis, criticized object-oriented languages where implementation inheritance and subtype inheritance are identified because it links the implementation of a class to the interfaces it supports [12]. Linking implementation inheritance and interface inheritance is undesirable because it inhibits reuse. If a client class depends only on an implementation adhering to a declared interface, then that client can be used by any implementation of the interface. However, if an interface is represented as a class, then only explicit subclasses can be used with the client.

Although linking implementation and interface inheritance is undesirable, it is not a necessary consequence of nominal subtyping. In fact, the Java programming language largely eliminates the linking of interface and implementation inheritance through the use of an explicit interface hierarchy that is distinct from the class hierarchy.

In their design of the PolyTOIL language, Bruce, Schuett, and van Gent argued that separating subtyping from the class hierarchy allows for greater language expressiveness [17]. In PolyTOIL, classes are run-time values that are unrelated to types. Subtyping is structural and includes support for explicit references to `MyType` (i.e., the run-time type of the receiver of a method call) inside a class definition. PolyTOIL's support for `MyType` is presented as an example of the greater expressive power that can be achieved by separating subtyping from the class hierarchy. However, the authors fail to mention the many advantages of nominal subtyping, particularly with regard to preventing spurious subtyping relationships, supporting type dependent operations, and easily supporting recursive datatypes.

Fortunately, some languages researchers have started to recognize the merits of nominal subtyping. In his tome on type systems, Pierce discusses some of the advantages of nominal subtyping, particularly with regard to mutually recursive type definitions [44]. Fisher and Reppy provide several examples of the limitations of structural subtyping in [26]. They argue that the nominal subtyping (a.k.a., “inheritance-based subtyping”) discipline employed by mainstream object-oriented languages has many advantages for production programming that are often overlooked by languages researchers. In order to demonstrate these advantages, they present an extension of Moby, a non-generic object-oriented language with structural subtyping. In their extension, both structural and nominal subtyping are supported. They formally model a small core of their extended language that includes Moby classes, but not modules.

Fisher and Reppy illustrate the value of nominal subtyping through a series of compelling examples:

- First, they show how class abstractions such as “friends” in C++ are difficult to encode with structural subtyping. Such abstractions critically depend upon class names to enforce special sharing relationships.
- Second, they point out the difficulties with any attempt to encode nominal subtyping relationships via “class tags” in a structurally subtyped language. One might consider encoding nominal subtyping by including structurally complex class tags in data types. The structural subtyping relations among the class tags should mirror the intended subtyping relations among the classes. Unfortunately, it is difficult to maintain such an encoding as a class hierarchy evolves; every time a new class is inserted into the hierarchy, the tags of all subclasses must be manually altered. Furthermore, such an encoding does not simplify the treatment of recursive data types, and it does not allow for type dependent operations.
- Third, they show that, because structural subtyping is typically based on only

the public members of a class, it does not allow for straightforward definitions of methods that rely on private members, such as binary methods and copy constructors.

- Fourth, they show that mixins, which are not supported in Moby but could be added in a future release, are much more useful in the presence of nominal subtyping.³ With nominal subtyping, shadowed fields of a mixin instantiation can be accessed via “upcasting” the static type of the receiver. With structural subtyping, there would be no benefit to upcasting; shadowed fields could not be accessed.
- Finally, Fisher and Reppy also mention that it is much easier to implement method dispatch and field lookup efficiently in a nominally subtyped language.

1.1.2 Generic Types

In an object-oriented language with generic types, class definitions and methods may be parameterized by type variables and program text may use generic (parameterized) types in many contexts instead of conventional types. For example, in Generic Java, a family of extensions to Java that support generic types, [14], a *generic class definition* has the form

```
class Identifier < TypeParameters > extends ...
```

where each entry in the list of *TypeParameters* (separated by commas) is a type variable with an optional *type bound* of the form `extends ClassType` or `implements InterfaceType`. A generic vector class in such a language might have the header `class Vector<T>`.⁴

A *generic type* consists of either a type variable or an application of a generic class to type arguments that may also be generic. In a generic type application, a

³The Moby implementation is in the alpha stage of development. Parameterized modules are expected in a future version, but are not yet supported and are not modeled in Fisher and Reppy’s core calculus [26].

⁴Throughout the rest of this thesis, we will write generic types using the syntax of Generic Java.

type parameter may be instantiated as any reference type that satisfies the specified bound. If the bound for a type parameter is omitted, the universal reference type (e.g., `Object`) is assumed. We refer to types that do not contain type variables as **ground types**.

Structural Type Systems Supporting Genericity

Parameterized types have been an active subject of research for nearly thirty years and have been analyzed extensively in the context of languages with structural subtyping.⁵ However, as we will see, the properties of generic type systems in the structural world are radically different from their counterparts in the nominal world, making cross-application of results difficult.

In the structural world, generic types were first formalized in an extension of the λ -calculus known as System F . System F was discovered independently by Girard [32], and by Reynolds [45] in the early 1970’s. In System F , there is no subtyping, and, therefore, no bounds on type parameters. It is possible to define a “type erasure” semantics for System F , where programs are simply mapped to programs in the untyped λ -calculus with observably equivalent behavior.⁶ Such a semantics is possible because System F does not include type dependent operations.

System F is a surprisingly expressive language. In it, we are able to encode many commonly used datatypes, including polymorphic lists. Nevertheless, Girard proved that System F is *normalizing*, i.e., the evaluation of every well-typed program terminates, just like the simply-typed λ -calculus [32].

Cardelli’s calculus of subtyping, presented in [18], was integrated with System F by Cardelli and Wegner in [20]. The resulting type system is known as System $F_{<}$. This system allows for parametric bounds on type parameters. However, recursive and mutually recursive type bounds are not allowed. Given a type parameter declaration

⁵In the parlance of structural type systems, generic typing is usually referred to as “parametric polymorphism”.

⁶However, Wells proved in [52] that type reconstruction is undecidable for System F .

\mathbb{T} extends \mathbb{N} the only type parameters within scope at type \mathbb{N} are those parameters whose declarations lexically enclose the declaration of \mathbb{T} . System $F_{<}$ was refined in [16] and [25], but the ban on recursive and mutually recursive bounds was not removed by these refinements. As was shown by Ghelli in [30], System $F_{<}$, like System F , is normalizing. More surprisingly, Pierce showed in [43] that the subtype relation (and hence type checking itself) in System $F_{<}$ is undecidable!⁷

System $F_{<}$ was formally extended by Ghelli in [31] to System *F-bounded*, where recursive (but not mutually recursive) type bounds were allowed. In that same paper, Ghelli proved that System *F-bounded* was also normalizing. Baldan showed in [10] that subtyping in *F-bounded* is not antisymmetric, and furthermore, that the subtype relation is not decidable. It should be noted that System *F-bounded* does not include recursive types, severely limiting the utility of type parameters with recursively defined type bounds [10].

As we shall show in the sequel, the properties of generic types in nominal type systems are radically different than in the structural world. Fundamental properties such as the difficulties in supporting mutually recursive type bounds and the undecidability of the subtype relation do not carry over to the nominal world. Also, even extremely simple calculi in the nominal world are not normalizing. As a result, most existing research on generic types is not directly applicable to mainstream object-oriented languages. At best it is suggestive and at worst misleading.

Nominal Type Systems Supporting Genericity

Until recently, there has been little formal analysis of generic types in the context of nominal type systems [34, 44]. But the lack of formal analysis has not prevented designers of nominally subtyped object-oriented languages from including generic type systems in their languages. This premature incorporation has resulted in many insidious errors.

⁷Object-oriented languages based on System $F_{<}$ and its extensions generally restrict the subtyping relation to a decidable subset.

As explained by Cook in [23], the Eiffel type system includes unrestricted covariant subtyping of type arguments in generic types, breaking type soundness. In the case of C++, no attempt was made to type check the parametric versions of classes at all. Each class instantiation is type checked separately. Because of this approach, there is no way for a programmer to know whether a particular instantiation of a generic class will cause a type error.

The first compiler to support generic types in Java, including type dependent operations, was an experimental compiler developed by Agesen, Freund, and Mitchell [2]. This compiler relies on a purely **heterogeneous** implementation of generic classes: a complete, independent copy of a generic class is generated for each instantiation. In their implementation, a customized class loader generates complete class instantiations from template class files in much the same way that C++ expands templates.

The heterogeneous approach to implementing genericity has two disadvantages. First, the heterogeneous expansion of every generic instantiation can produce an exponential blow-up in the size of the executable code and can seriously degrade program performance. Second, the heterogeneous approach does not provide a common superclass for all instantiations of a particular generic class, preventing the use of **raw types**. *Raw types* are references to parametric types that do not include the expected type arguments. When all instantiations of a generic type share a common base class, a reference to a raw type can match either a non-generic legacy class *or* a corresponding generic base class, allowing for intermingling of generic and non-generic code.

The Pizza programming language was an extension of Java that added generic types, first-class functions, and “algebraic data types” (type case statements with pattern matching) [42]. The Pizza formulation of Generic Java was based on the notion of **type erasure**. In a system based on *type erasure*, generic types are present only during static type checking. After type checking, every generic type in the program is *erased*, i.e., it is replaced with a non-generic upper bound. For example,

type annotations such as `List<T>` are erased to `List`, and (more significantly) naked type variables such as `T` are erased to their declared bound (typically `Object`). In a generic type system based on type erasure, generic types cannot be used safely in type dependent operations, which explicitly refer to run-time types, such as casts, `instanceof` operations, and `new` expressions.

Pizza was based on research by Odersky and Wadler on adding genericity to Java [42] using nominal type systems loosely based on System *F-bounded*. Because type erasure is a common practice in the world of structural subtyping, it is natural to employ it when adding a generic type system for a nominally subtyped language *if* the theory on which generic types are based is System *F-bounded*. In fact, employing type erasure has the advantage that erased generic classes will look like their non-generic legacy counterparts, providing the potential for code reuse [15]. With type erasure, references to non-generic legacy classes can be explicitly supported via raw types.

Despite the advantages of type erasure, there is a serious downside in the context of nominally subtyped languages. Because of the key role that types play at run-time in these languages, type erasure severely restricts the ways in which we can use generic types when compared to non-generic types. As with C++, some formulations of generic types for Java that are based on type erasure forego type soundness and allow selective violations of type safety so that programmers can work around the restrictions caused by this implementation scheme.

The Pizza extension to the Java language was the first in a family of language extensions to Java that include generic types via a common syntax and a mutually compatible semantics. This family of languages is referred to as *Generic Java*. Following the Pizza compiler, Martin Odersky developed a well-engineered compiler for a formulation of Generic Java called GJ that supports genericity via type erasure as in Pizza, but does not include the other extensions of Java included in Pizza [15]. Because GJ, like Pizza, is based on the notion of type erasure, it does not support type dependent operations on generic types. But, unlike Pizza, which simply forbids

such operations, GJ allows them, issuing only an *“unchecked warning”* during compilation. GJ then erases the types involved in the operations, breaking type safety [15]. For example, given a type parameter `T extends Object`, a cast to type `T` will be erased to a cast to type `Object`. The value of the expression cast can then be used as an instance of type `T`, *but no check is done to confirm that the value is of type T*.

An alternative proposal for Generic Java, called NEXTGEN and developed by Cartwright and Steele in [21], addressed the shortcomings of Pizza and GJ by adding support for type dependent operations on generic types. NEXTGEN provides this support through an efficient encoding of the type relationships in a non-generic Java class hierarchy. NEXTGEN and GJ were designed in concert with one another so that NEXTGEN would be backward compatible with GJ. We will give a detailed account of the NEXTGEN language and compiler design, as well as our implementation scheme, in Chapters 4, 5, and 6. A prototype release of the NEXTGEN compiler is available for download at

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

Viroli and Natali have proposed supporting run-time generic type information in Java by embedding the information in an extra field attached to objects of generic type and using reflection to implement type dependent operations [51]. This approach has two obvious disadvantages. First, every object of generic type requires an extra word of memory. For small objects such as list nodes, this space penalty can be significant (25% for an object with two fields assuming a two word header). Second, using reflection to implement type dependent operations is slow. Viroli and Natali argue that the overhead of reflection can largely be eliminated by performing some load-time optimizations to streamline the implementation of type dependent operations. They use synthetic micro-benchmarks to compare the performance of their implementation scheme with the original NEXTGEN implementation scheme described by Cartwright and Steele [21]. We believe their results are misleading because they presume no method inlining for NEXTGEN while presuming it for their

implementation. Moreover, they model the performance of the NEXTGEN implementation scheme described in [21] where a separate instantiation class and interface file must be read from disk for each class instantiation. In Chapter 4, we will explain how, in our implementation of a NEXTGEN compiler, we improved upon the scheme suggested in [21] and greatly reduced the number of disk accesses. Unfortunately, their Generic Java implementation is not yet complete, preventing a direct empirical comparison with NEXTGEN.

Another related implementation of generic types is the PolyJ extension of Java developed at MIT [41]. The PolyJ website suggests that PolyJ is similar to NEXTGEN in some respects, but neither inner classes nor polymorphic methods are supported. In addition, PolyJ does not attempt to support interoperability between generic classes and their non-generic counterparts in legacy code. The language design includes a second notion of interface that uses a structural matching scheme. Despite the description on the PolyJ website, the only published paper on PolyJ describes a completely different approach to implementing genericity than NEXTGEN that relies on modifying the JVM. The distributed PolyJ compiler generates JVM compatible class files but the techniques involved have not been published. The PolyJ language design is not compatible with GJ or with recent Java evolution.

Formal Analysis of Generic Types in Nominally Subtyped Languages

Recognizing the limitations of the applicability of existing formal analyses to generic types in a nominally subtyped language like Java, Igarashi, Pierce, and Wadler developed Featherweight GJ, a small formal model of Generic Java, and proved this generic type system to be sound [34]. The generic type system modeled by Featherweight GJ is loosely based on that of *F-bounded*. But, unlike *F-bounded*, it is a nominally subtyped language, and therefore it naturally supports both mutually recursive type definitions and **mutually recursive type parameter bounds**. With *mutually recursive type parameter bounds*, the bound on a type parameter may include other

type parameters declared in the same context. For example, we can have generic class headers like the following:

```
class C<S extends D<T>, T extends D<S>> {...}
```

In class `C`, the bound on type parameter `S` refers to type parameter `T`, which is declared in the same context. Similarly, the bound on type parameter `T` refers to `S`. Mutually recursive type bounds are important for encoding many object-oriented designs [15].

In addition, Featherweight GJ naturally supports recursive function definitions, and is therefore not normalizing. The following, trivial, Featherweight GJ program does not halt:

```
class C extends Object { C() {} Object m() {return this.m();} }
new C().m()
```

Featherweight GJ was presented in concert with Featherweight Java, a small core calculus for the Java language that is useful for studying formal properties of various extensions to Java. A type soundness proof for Featherweight GJ was presented in [34] as a case study of the utility of Featherweight Java in modeling the formal properties of Java extensions. It is quite striking that in this single study of a generic type system in a nominally subtyped language, all of the key limitations of the analyses of generic types in the structural world (i.e., the prohibition against mutually recursive type bounds, the lack of integration with recursive types, the undecidability of subtype checking, etc.) were easily overcome in the nominal world, thus illustrating how fundamentally different these two forms of type system are.

Igarashi et al presented two separate operational semantics for Featherweight GJ. The first employs a straightforward “type carrying” semantics. The second employs type erasure, where Featherweight GJ programs are erased to Featherweight Java programs. These two semantics are shown to be equivalent. This equivalence holds in the presence of type dependent operations because, unlike GJ, Featherweight GJ severely

restricts the use of type dependent operations on generic types. Type dependent operations on type variables are disallowed, references to raw’ types are disallowed, and casting to a parametric type is allowed only when the success of the cast can be determined solely by comparing erased types. By making these restrictions explicit, Featherweight GJ elucidates the key properties that make the full GJ type system unsound.

Featherweight Java has since been extended in many directions to formally model various aspects of the Java language in isolation, such as inner classes [35]. It has also been used to analyze extensions to generic type systems, such as raw types [36]. It will form the basis of our analysis of first-class genericity in Chapter 9.

Generic Types in Java

In 1999, Sun Microsystems publicly announced its interest in adding generic types to Java by publishing *Java Specification Request 14: Adding Generics to the Java Programming Language* [49].⁸

In 2001, Sun released an “early access” compiler for Generic Java (called JSR-14) based on the GJ compiler. Sun Microsystems has indicated that the next major release of the Java Platform (J2SDK 1.5) will include support for “second-class” generic types based on GJ.

Generic Types in C#

The extension of the .NET common runtime by Kennedy and Syme to support generic types in C# [37] bears a striking resemblance to the NEXTGEN formulation of Generic Java. Kennedy and Syme follow the same, mostly homogeneous, approach to implementing genericity described in the NEXTGEN design [34]. Because C# includes primitive types in the object type hierarchy, they support class instantiations involving primitive types and rely on a heterogeneous implementation in these cases.

⁸Odersky’s GJ compiler preceded JSR-14 and strongly influenced the definition of Generic Java.

To handle cycles in the type application graph, they dynamically generate instantiation classes from templates as they are demanded by program execution. Because they were free to modify the .NET common language runtime, their design is less constrained by compatibility than proposed extensions to Java with generic types. Because of the similarity of this implementation to that of the NEXTGEN compiler, many of the same implementation issues we will discuss for adding first-class genericity to Java apply just as well to C#.

1.1.3 First-Class Generic Types

Although the forthcoming addition of second-class generic types to Java represents a major step forward in the evolution of that language, the restriction of generic types to second-class contexts prevents programmers from applying generic typing to many important object-oriented coding patterns. For example, as we will show in Chapter 2, the `Cloneable` interface from the core Java API cannot be used in generic classes in GJ because the output of the `clone()` method cannot be cast to the appropriate generic type. Even the JSR-14 compiler, which is written in GJ, must *breach* the GJ type system because the compiler source code uses a cast to type `T` where `T` is a type parameter [11, 48].⁹

To work around this restriction, the JSR-14 compiler, like the GJ compiler, accommodates breaches in the type system by generating code for programs that use expressions of erased type in contexts requiring a specific generic type—provided that the base types match. Of course, sound static type checking is lost in the process. Furthermore, there are cases where the compiler generates incorrect code for untypable programs.¹⁰

In contrast, NEXTGEN eliminates these pathologies by retaining parametric type information at run-time and customizing the code in generic classes where necessary to

⁹See the polymorphic method `get` in class `Context` in package `com.sun.tools.javac.v8.util` of version 1.3 of the JSR14 compiler [48].

¹⁰For example, `new T[]` compiles to `new E[]` where `E` is the erasure (bounding interface) for `T`.

support (parametric) type dependent operations. NEXTGEN is backward compatible with the forthcoming JSR-14 implementation of Java.

In NEXTGEN, the only significant restriction on the use of generic types is the prohibition against using naked type variables as superclasses in generic class definitions [21],¹¹ *i.e.*, class definitions the form

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

are prohibited.

In NextGen, each instantiation of a generic class is a separate class that exists during program execution. As shown in Chapter 7, benchmark results for our prototype compiler for NEXTGEN demonstrate that NEXTGEN-style treatment of generic types does not have any significant performance overhead when compared with either conventional Java or GJ/JSR-14 [6].

The lone restriction on the use of generic types in NEXTGEN is significant because it prevents NextGen from supporting mixins. A simple example of a mixin class definition forbidden in NEXTGEN (and other object-oriented languages supporting genericity) is shown in Figure 1.2. This class definition is obviously illegal in NEXTGEN because the class extends its own type parameter T. However, the intended meaning of this class definition is clear: each distinct instantiation of the class, such as `TimeStamped<Hashtable>`, should extend a distinct superclass. In essence, each instantiation defines a new version of the superclass that supports the functionality embodied in class `TimeStamp`.

To simulate the behavior of this class in Generic Java (as embodied in GJ or NEXTGEN) or Generic C#, we would either have to copy this class definition once for each class we want to extend, or we would have to use *composition* (*e.g.*, the Decorator Pattern [29]) to encode the subclassing relation. The former solution involves

¹¹The proposed generic extension of C# imposes the same restriction. Both NextGen and Generic C# also exclude naked type variables from appearing in the list of interface types implemented by a class. But such constructions are not semantically sensible because each superinterface for a class specifies a lower bound on the member methods defined in the class. If a superinterface were a type variable, the corresponding lower bound would depend on the particular binding of the type variable, forcing the class to *define a method for every possible method signature*.

```

class TimeStamped<T> extends T {

    public long time;

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

```

Figure 1.2 : A Simple Mixin Class

undesirable code replication. The latter solution forces source programs to include a multitude of forwarding methods with less precise types. Moreover, the decorated classes must be designed with decoration in mind. Clearly, there is strong motivation to relax the language definition to allow for class definitions such as `TimeStamped`. However, we will see that eliminating this apparently small restriction on the syntax of Generic Java has profound ramifications on the language semantics.

Mixins

Nearly 20 years ago, the Lisp object-oriented community invented the term *mixin* [40] to describe a class with a parametric parent such as the `TimeStamped` class in Figure 1.2. The name was inspired by the fact that such classes can be mixed together (via subclassing) in various ways, like nuts and cookie crumbs in ice cream. Denotationally, mixins can be thought of as functions mapping classes to new subclasses. For example, class `TimeStamped` can be viewed as a function that takes a class such as `Hashtable` and returns a new subclass of `Hashtable` that contains a timestamp.

Because mixins can be viewed as functions over classes, we believe it is natural to formulate them in the context of a generic type system. In this context, mixins are simply generic classes that extend one of their own type parameters. However, this formulation is quite different than historical formulations. Indeed, because the notion of mixins originated in dynamically typed languages, mixins existed long before they

were analyzed in the context of static type checking, generic or otherwise.

In CLOS, programmers were able to express mixins by breaking some of the key abstractions of the language [12]. In CLOS, all classes that affect a class are ordered by a linearization algorithm. This ordering determines class precedence during method lookup. However, the linearized ordering will not always put a class next to its parent, affecting the semantics of `super` (a.k.a, `call-next-method`) calls in strange ways. Additionally, because CLOS programs are not statically type-checked, it is possible for programmers to place `super` calls in a class that has no parent. By ensuring that a linearization algorithm places a mixin class into the class precedence list of an object, the mixin class could be used to influence the behavior of the object. `super` calls in the mixin would simply refer to whatever class was placed directly above the mixin by the linearization algorithm.

Gilad Bracha, in his doctoral thesis, developed a coherent semantics for mixins by developing the Jigsaw language [12]. Jigsaw is a structurally subtyped language, with a semantics defined via translation to the λ -calculus. Jigsaw supports mixins and multiple inheritance. Mixins (a.k.a. “mixin modules”) can be combined via a variety of operators. These various operators differ in the way they handle conflicting member names. When a mixin is combined with another class or mixin, the names of its members may clash. For example, consider the `TimeStamped` class in Figure 1.2, which contains a public field `time`. Suppose we want to combine this class with a class `Calendar` that also contains a field `time`. How do we resolve the clashing field names in the combined entity? Also, how do we resolve name clashes in members that can be overridden, such as methods? The question of how to handle name conflicts like these is a recurring theme in the analysis of mixins.

In Jigsaw, a symmetric `merge` operator allows combination of mixins that do not have naming conflicts.¹² A non-symmetric `override` operator was also provided. Bracha discussed two mechanisms for handling naming conflicts when apply-

¹²Unlike some of the calculi will discuss in later chapters, naming conflicts in Jigsaw are statically detectable.

ing **override**. In the first mechanism, one of the members with a given name conflict shadows the other. Hence, the shadowed member can no longer be accessed. In the second mechanism, the programmer explicitly renames conflicting members, allowing both members to be accessed. Because all name conflicts in Jigsaw can be detected statically, it is possible (if cumbersome) for programmers to detect all name conflicts and rename them.¹³ Bracha pointed out that, in the context of structural subtyping, member renaming has the undesirable consequence of altering subtyping relationships. Bracha’s system formed the basis of a formal algebra of “mixin modules” defined by Ancona and Zucca in [9].

Hygienic Mixins

The problem of accidental name clashes in mixin instantiations was approached in a significantly different manner by Flatt, Krishnamurthi, and Felleisen in a toy language loosely based on Java called MIXEDJAVA [28]. Their solution to accidental name clashes was to devise what we call **hygienic mixins**. In *hygienic mixins*, if two members of a mixin instantiation have conflicting names, neither is shadowed. Instead, member resolution is determined based on context. Each use of a hygienic mixin has a corresponding *view* that resolves conflicting member references at the point of usage. The programmer is given control to modify the view of a mixin in various contexts.

MIXEDJAVA does not include generic types; mixins are formulated as a separate language construct. We will see that when mixins are encoded as generic types, hygienic mixins are essential for type soundness. In MIXEDJAVA, because all mixin instantiations (and hence all naming conflicts) can be determined statically, their use of a hygienic semantics was not critical to type soundness, but it was included in order to maintain class encapsulation and to allow programmers to reason about the components of their programs in isolation. In MIXEDJAVA, all classes are constructed

¹³We will see that in a language based on first-class genericity, naming conflicts are not always statically detectable. In that case, requiring the programmer to explicitly rename conflicting members is not a viable option.

by mixins. Views are sequences of mixin names used to specify both the static types of expressions and the dynamic types of program values. Every value in MIXEDJAVA is a pair consisting of an explicit view and an object reference. As a result, there is a major implementation penalty for hygiene in MIXEDJAVA: the size of every reference value is doubled. An object o has type T iff the type is a segment of the chain of mixins used to form the class of o . Programs can cast a value to a compatible view, creating a new value with the explicit view specified in the cast.

MIXEDJAVA is an interesting design study but it does not provide a practical basis for extending existing production run-time systems such as C# or the Java Programming Language. The fact that values are pairs containing a view and an object reference means that every value occupies two machine addresses instead of one, nearly doubling the memory footprint of many applications and significantly slowing computation. In addition, it is not clear how to map MIXEDJAVA onto existing run-time systems in a way that preserves compatibility with legacy binary code.

Mixins for the Java Language

To our knowledge, the first reference to mixins in Java was made by Agesen, Freund, and Mitchell [2] while describing their extension to Java to support genericity via syntactic expansion in the class loader. Although they mention that their approach can support mixin constructions, no type system supporting mixins is given and the critical language design issues involved in such a language extension—such as mixin hygiene, the status of abstract methods in mixins, and the definition of mixin class constructors—are not discussed. Since their model for supporting generics is syntactic expansion, they presumably are proposing non-hygienic mixins. In that case, we do not believe that the static type checking of mixins is compatible with the separate class compilation provided by modern compilers for Java and C# because type correctness requires a whole-program analysis to confirm that overridden methods in

mixin instantiations have the proper type signatures.

Two other practical proposals for adding mixins to Java appearing in the literature are Jam [7] and Jiazzi [38]. Neither accomodates generic types. Jam is an extension of Java 1.0 developed by Ancona and Zucca that supports mixin definitions as a new form of top-level definition supplementing classes and interfaces. Each mixin instantiation is explicitly defined by a special form of class definition that includes the constructors for the new class. Jam is based on the theoretical framework for “mixin modules” as presented in [8]. That theoretical framework is not hygienic because it provides no mechanism for altering the “view” of a mixin based on context. Multiple operators are provided for mixin composition with various rules for shadowing colliding methods, but the set of visible methods resulting from each of these rules is not dependent on the context of the reference to a mixin instantiation. Also, it does not account for the many complexities that arise when mixins are formulated as generic types.

Since the Jam type system lacks the expressiveness of generics, it must restrict the use of `this` within the body of a mixin. In particular, `this` cannot be passed as an argument to a method, which is a severe restriction on the design of object-oriented programs. Although Jam mixins are not hygienic, programs that perform accidental method overriding with incompatible type signatures are rejected by the type checker.¹⁴ Jam is implemented by a preprocessor that maps Jam to conventional Java.

Jiazzi is a component system for Java developed by McDirmid, Flatt, and Hsieh that supports component level mixins [38]. Jiazzi is implemented by a linker that processes class files to produce new class files. Using Jiazzi, a program can be partitioned into components with unresolved references (wires) and compositions that wire components together. Since Jiazzi is a component system that is not part of the program source language, it is not directly comparable to language-level mixins. But mixins can be created by the component linking process because the superclass

¹⁴Notice that static detection of accidental overrides is possible in Jam because Jam does not formulate mixins via first-class genericity.

of a class may be an unresolved reference within a component.¹⁵ Since mixins can be instantiated only in the meta-language used to wire components together, Jiazzi does not have to address the same language design issues that arise in language-level mixins. Nevertheless, Jiazzi mixins must be hygienic because components only expose selected public methods. In the absence of hygiene, component composition would break component encapsulation. Jiazzi enforces mixin hygiene and the static typing of mixins by performing a whole-program analysis on a program composition.

1.2 Roadmap

Although nominal type systems, generic types, and mixins have each been studied separately, they have yet to be analyzed together under the unifying concept of first-class genericity. We will see that, by combining all of these notions, a first-class generic type system is expressive in ways beyond what can be achieved by each of these features in isolation.

The remainder of this thesis is organized as follows. First, we explain why first-class genericity should be supported in object-oriented languages. We also motivate the “Safe Instantiation Principle”, a new principle we propose for evaluating generic type systems in nominally subtyped object-oriented languages. We proceed to outline our design and implementation of a compiler for Cartwright and Steele’s NEXTGEN extension to Generic Java, which supports quasi-first-class generic types. We demonstrate, through a series of benchmark results over our compiler, that including run-time support for generic types can be done without incurring any significant overhead. We then proceed to consider extending NEXTGEN to support first-class genericity. We define the MIXGEN language, an extension of NEXTGEN that supports mixins. We show how to map this language onto an existing and widely used run-time system, the Java Virtual Machine, while maintaining compatibility (including the capacity for subclassing) with existing compiled binaries. Then, to demonstrate the

¹⁵To appease the Java compiler, which will not compile a class with an undefined superclass, a stub class for each such unresolved reference is defined.

soundness of the MIXGEN type system, we present an operational semantics, type inference system, and type soundness theorem for CORE MIXGEN, a subset of MIXGEN that encapsulates the important aspects of the MIXGEN type system. Finally, we conclude and discuss directions for future research.

Chapter 2

Motivation for First-Class Genericity

The NEXTGEN programming language adds generic types to Java in such a way as to support type dependent operations. As we will explain in chapters 4, 5 and 6, NEXTGEN provides such support by creating a separate class to represent each distinct instantiation of a generic type. In essence, NEXTGEN supports the same Generic Java language as GJ, but eliminates the restrictions against using operations that depend on run-time generic type information. In NEXTGEN, generic types are *quasi-first-class* types; they can be used anywhere that conventional types can *except for extends* clauses of class definitions. The NEXTGEN compiler employs partial type erasure as an optimization technique to avoid code replication, but this optimization has no impact on the semantics of NEXTGEN programs. In this chapter, we will show through a series of short coding examples that type dependent operations matter because they enable programmers to write cleaner, more reliable code. We then proceed to explain why true first-class genericity, including mixins, adds a powerful form of expressiveness to a language. We will ground our discussion by comparing the NEXTGEN formulation of generic types with a particular formulation based on type erasure: the JSR-14 prototype compiler.

2.1 Why Support Type Dependent Operations

Because first-class genericity entails support for type dependent operations, we will first discuss why the use of type dependent operations on generic types is important.

2.1.1 The Singleton Pattern

One of the most widely applicable design patterns in programming practice is the Singleton Pattern.[29] In this pattern, all instances of a class are indistinguishable, allowing them to be represented by a single instance which can be bound to a static field of the class. The only way to “generate” an instance of this class is to refer to this field (to enforce this practice, class constructors can be declared private).

For example, consider the following class hierarchy for representing binary trees:

```
abstract class Tree {}

class Leaf extends Tree {}

class Branch extends Tree {
    Object value;
    Tree left;
    Tree right;
}
```

Notice that class `Leaf` contains no fields. All instances of this class are functionally equivalent. Therefore it would be wasteful to construct a new instance of this class every time a `Leaf` is needed. Instead, we can apply the Singleton Pattern and include an extra static field `ONLY` in class `Leaf`:

```
class Leaf extends Tree {
    static final Leaf ONLY = new Leaf();
}
```

Now, this field can be referred to whenever a `Leaf` is needed.

If we express this class hierarchy in Generic Java, we obviously should parameterize the classes by the type of the elements contained in `Branch` nodes, as follows:

```
abstract class Tree<T> {}

class Leaf<T> extends Tree<T> {}

class Branch<T> extends Tree<T> {
    T value;
```

```

    Tree<T> left;
    Tree<T> right;
}

```

Now, consider the static field `ONLY` we added to class `Leaf` in the non-parametric class hierarchy. What should be the type of this field?

In the JSR-14 extension of Java, a static field of generic class is shared across all instantiations of the class [15]. This design choice is dictated by the fact that JSR-14 uses type erasure to implement generic types: a generic class is translated to a conventional class—called a *base class*—by converting all references to generic types to their upper bounds (which are conventional types). Hence, there can only be only one static field `ONLY` for all of the instantiations of the generic type `Leaf<T>()`, e.g. `Leaf<Integer>`, `Leaf<String>`, But such a static field shared across all instantiations cannot have type `Leaf<T>`, because no Java object can belong to type `Leaf<T>` for *all* types `T`. In Generic Java, the types `Leaf<Integer>`, `Leaf<String>`, ... are all disjoint. For this reason, the JSR-14 compiler prohibits the type parameters of a generic class from appearing within the type of a static field of the class.

The only way to support the Singleton Pattern in the JSR-14 extension of Java is to declare a separate static field for each anticipated instantiation of a generic class as follows:

```

class Leaf<T> extends Tree<T> {
    static final Leaf<Integer> INT_ONLY = new Leaf<Integer>();
    static final Leaf<String> STRING_ONLY = new Leaf<String>();
    ...
}

```

This solution is clumsy, requires code duplication, and is not stable under program extension. If a program requires an instantiation of `Leaf<T>` for which no singleton `ONLY` field has been declared the generic class `Leaf<T>` must be modified to include a new static field. Furthermore, singleton instances with types other than ground types cannot be expressed.

In contrast, the NEXTGEN formulation of generic types, where each instantiation of a generic class by a separate class at run-time, allows for the specification of *per-instantiation* static fields with generic types [21]. Although the current release of NEXTGEN does not support generic types, it could be gracefully extended to support them. For example, we could extend the language so that, if a static field declaration includes the new modifier *generic*, then each instantiation of the generic type contains its own copy of this generic field. In the case of class `Leaf<T>`, this facility allows us to apply the Singleton Pattern in the natural way:

```
class Leaf<T> extends Tree<T> {
    static generic final Leaf<T> ONLY = new Leaf<T>();
}
```

Each instantiation of class `Leaf` will then contain its own field `ONLY` of the appropriate type.

2.1.2 Casting and Catching Generic Types

In JSR-14, casting to a generic type `C<E>` is unsafe unless the parametric information `E` can be statically deduced by the type checker (enabling the compiler to implement the cast `(C<E>)` by its erasure `(C)`). Similarly, the declaration of generic exception types is prohibited because catch clauses cannot match generic types (only their erasures).

A good illustration of the awkwardness of the inability to dynamically confirm or query the generic type of an object arises when trying to use the `Cloneable` interface in a generic class. The method `Object clone()` is inherited by all classes from class `Object` but it throws an exception unless the invoking class implements the `Cloneable` interface. Of course, to make much use of the `clone()` method for a class `C`, a client must cast the result to type `C`. If `C` is generic, then the cast must be to a generic type `C<E>`, which is prohibited in JSR-14 unless the parametric type information `E` can be statically inferred (an uncommon occurrence in practice). In NEXTGEN, all generic casts are legal so the `Cloneable` interface naturally applies to generic classes in exactly the same way it does to conventional classes.

2.1.3 Functional Mapping over Arrays

When working with arrays and other compound data structures, it is often useful to map a transformation over the constituent elements of the data structure, producing a new structure consisting of the resulting elements. Generic types allow us to perform such transformations with far less casting. For example, consider the following parametric `Mapper` interface:

```
public interface Mapper<A,B> {
    public B map(A element);
}
```

Suppose we want to apply a particular implementation of this `Mapper` interface to an array. In the process, we would like to create a new array of type `B[]` to hold the resulting elements. We could write the code to do this in `NEXTGEN` as follows:

```
public class ArrayMapper {
    <A,B> public static B[] mapArray<A,B> aMapper) {
        B[] out = new B[in.length];
        for (int i = 0; i < in.length; i++) {
            out[i] = aMapper.map(in[i]);
        }
    }
}
```

But since `JSR-14` does not support run-time generic type operations such as `new B[]`, the preceding code is invalid in `JSR-14`. Some other approach would be needed, such as passing an extra argument `dummy` of type `B[]` to the `map` method and using the static method invocation `Array.newInstance(dummy.getClass(), in.length)` to create the resulting array.

2.1.4 Legacy Classes and Interfaces

Any viable extension of Java with generic types must have an efficient implementation on top of the existing Java Virtual Machine and must interoperate with existing compiled binaries. `JSR-14` adheres to this constraint through the use of type erasure.

Because generic types are erased at compile-time, they are identical at run-time to ordinary Java classes. Consequently, the JSR-14 compiler can be fooled into compiling generic class references into references to existing Java classes. The trick employed is to set the class path during compilation to include generic “stub” classes, which have the same signatures as the erasures of the non-generic classes they spoof. This feature enables JSR-14 to re-interpret existing “naturally generic” library classes or other binaries as classes with generic signatures.

In addition, JSR-14 allows Generic Java code to refer directly to the base classes corresponding to generic classes. However, this feature breaks the type soundness of Generic Java: a JSR-14 program can pass type-checking, yet generate a run-time type error (a `ClassCastException`) at a point in the source program where there is no cast [34, 15]! The run-time error is produced by a cast that is automatically inserted by the JSR-14 compiler. When this happens, how is the programmer supposed to diagnose what went wrong, much less repair it?

In NEXTGEN, there is a clear distinction between instances of a “naturally generic” legacy (non-generic) class and instances of a corresponding generic class. The latter can be defined as subclasses of the former, but they cannot be freely used in place of one another. To support the interoperation of code using a “naturally generic” legacy class and Generic Java code using a corresponding generic class, the programmer must write explicit conversion routines that convert legacy class instances to generic class instances and vice-versa¹ and perform the conversions whenever “naturally generic” data is passed between legacy code and new code.² This task superficially looks like extra work, but it provides scaffolding for checking that legacy (non-generic) data passed to a site requiring data of generic type actually satisfies the invariant associated with that generic type.

¹ For naturally generic classes in the standard Java libraries, the NEXTGEN extension of the libraries would obviously include the required conversion methods.

² If the new generic class is defined as a subclass of the old “naturally generic” class, conversion is required whenever old data is passed to a new context or new data is passed to an old context that mutates the data.

2.1.5 Transparent Debugging

In the JSR-14 extension of Java, generic type information cannot be used in the program debugging process because it is erased by the compiler. Hence, at a breakpoint, a debugger can only report the erased types of data objects, not their generic types. For example, an empty `List<Integer>` will be reported simply as an empty list. In contrast, NEXTGEN preserves all generic type information at run-time so a debugger can report the precise types of all data objects. The output of the debugger is complete and consistent with the source code.

2.2 Why Support Mixins

In addition to the desirability of supporting type dependent operations, a first-class generic type system also allows for mixins. Mixins constitute a powerful abstraction mechanism with many important applications [7, 13, 28]. We briefly cite three of them:

- First, mixins can define *uniform* class extensions that add the same behavior to a variety of classes meeting a specified interface. The `TimeStamped` class in Figure 1.2 is an example of this form of mixin.
- Second, mixins provide a simple, disciplined alternative to multiple implementation inheritance. A class constructed as the result of multiple mixin applications contains implementation code from multiple independent sources. Mixins provide essentially all of the expressive power but none of the pathologies of multiple inheritance [28].
- Finally, mixins provide the critical machinery required to partition applications into logically independent “modules” or “components” in which all of a module’s contextual requirements are decoupled and captured in its visible interface. Existing package systems for mainstream languages like Java and C# are severely limited by the fact that packages contain embedded references to

specific external class names, akin to hard coded filename paths in Unix scripts. These embedded references inhibit the reuse of a package in new contexts, and often prevent programmers from testing a package in isolation.

Formulating mixins as generic classes is particularly appealing because it provides precise parametric type signatures for mixins and enforces them through static type checking. In addition, this approach to defining mixins accommodates the precise typing of non-mixin classes provided by genericity. Hence, code containing mixins can be subjected to the same level of precise parametric type checking applied to other generic types, implying that the parametric types declared in mixins are respected during program execution. Previous formalizations of language-level mixins [40, 46, 13, 12, 28, 7] have not incorporated them into a generic typing discipline, and have sacrificed either precise type checking or expressiveness as a result.

2.3 Conclusion

Generic types hold great promise for simplifying the application of many design patterns and coding practices in nominally subtyped object-oriented languages. But, as the preceding examples have shown, their full value cannot be realized without support for first-class genericity. If such operations that depend on first-class generic types are excluded from the language, programmers will find generic types to be frustrating and awkward in many situations. Moreover, programmers who are not intimately familiar with the type erasure model for supporting generic types will be perplexed as to when they can or cannot use a generic type. Finally, as we have demonstrated in the last two sections, support for run-time type operations is critical for type-safe compatibility with non-generic legacy code and transparent debugging.

In the next chapter, we will discuss another desirable property of a generic type system, i.e., safe instantiation, and we will explore the implications of this property for first-class genericity.

Chapter 3

The Safe Instantiation Principle

In this chapter, we introduce the “safe-instantiation principle” a new design principle for evaluating generic types systems for object-oriented languages. We discuss the GJ and NEXTGEN formulations of Generic Java and the implications of safe instantiation on both approaches. We then consider the implications of safe-instantiation for a first-class generic type system. Finally, we defend the formulation of mixins as *hygienic* program constructs, arguing that a hygienic formulation is the only way to maintain safe instantiation and type soundness in the context of first-class genericity, and to prevent the introduction of insidious bugs with no clearly defined point of blame.

3.1 Motivation

In order to motivate the Safe Instantiation Principle, consider a situation in which an applications programmer is attempting to use a generic class `C<T>` that he does not maintain. Because Java supports separate class compilation, the only constraints on instantiations of `T` that are accessible in general by the client programmer are the declared constraints, *e.g.*, the upper bounds on type instantiations such as `T extends Comparable`. Also because of separate compilation, it is impossible when compiling a generic class to determine all of the instantiations of `C` that will occur at run-time. If a client programmer instantiates `C<T>` with a type argument that satisfies the declared constraints but still causes an error such as a `ClassCastException`, the client programmer may be left with no recourse but to report a “bug” to the maintainer of `C`. But if the maintainer of `C` is unable to specify constraints that prohibit such instantiations, he is powerless to prevent `C`’s misuse. Informal documentation may help

to some extent, but if the error caused by an instantiation has no immediate symptoms, diagnosis of the problem once it is noticed can be exceedingly difficult. Similar difficulties with error diagnosis have provided some of the most powerful arguments *for* type safety and *against* unsafe language features such as pointer arithmetic and manual storage management. For this reason, we propose that the following principle should be adhered to by any generic type system:

The Safe Instantiation Principle: Instantiating a parametric class with types that meet the declared constraints on the parameters should not cause an error.

In a statically typed language, the Safe Instantiation Principle entails type soundness.¹ If a generic type system does not enforce it, then the value of static type checking is substantially reduced, just as it is reduced by unchecked template expansion in C++. But although this principle may appear hard to disagree with, it is actually missing in many existing generic types systems [7, 21, 15, 42]. In fact, this principle is deceptively difficult to adhere to. In the remainder of this chapter, we will examine three formulations of generic type systems for Java, and discuss the implications of safe instantiation on each of them.

3.2 GJ, Type Erasure, and Safe Instantiation

In the GJ formulation of Generic Java, generic type information is kept only for static type checking. After type checking, all generic types are “erased” and are therefore unavailable at run-time. For this reason, type dependent operations such as casts, `instanceof` checks, and `new` expressions are not safe in general in GJ [15]. As was demonstrated in Chapter 2, the lack of run-time type information in GJ significantly impairs the expressiveness of Generic Java. However, an orthogonal problem with GJ is that it fundamentally violates safe instantiation. Although the compiler issues

¹However, the implications of safe instantiation extend far beyond type soundness, since instantiation-dependent compile-time errors caused by legal instantiations are also forbidden.

a warning when a generic class includes unsafe operations, the programmer is not prevented from producing unsafe class files. For this reason, a client programmer has no guarantee that the instantiation of a generic class will not result in a type error, such as a `ClassCastException`, when the instantiation types are misapplied in the body of the class.

It is natural to respond that the GJ compiler should simply prohibit unsafe operations, but such a prohibition would be an unacceptable restriction of program expressiveness. Because of the limitations of the GJ type system, many very reasonable GJ programs are forced to include unsafe operations.²

In a nominally subtyped language, we do not know of any satisfactory way to implement generic types via type erasure that respects safe instantiation, and we believe that this very fact is a strong argument in favor of an approach such as NEXTGEN that includes run-time type information. However, as we will see, it is also easy to violate safe instantiation in the design of NEXTGEN if we are not careful.

3.3 NEXTGEN and Safe Instantiation

In the case of NEXTGEN the key issue with safe instantiation is `new` expressions. NEXTGEN allows `new` operations on type parameters, but if their use is not restricted, a matching constructor for a `new` expression may not exist. Additionally, unless every instantiation of a type parameter is concrete, a `new` operation may be attempted on an abstract class.

In principle, we could maintain safe instantiation by *(i)* restricting `new` operations on naked type variables to call zeroary constructors, *(ii)* checking that every instantiation of a type parameter is concrete, and *(iii)* checking that every instantiation of a generic class includes a zeroary constructor. However, such Draconian constraints vastly reduce the expressiveness of our language. Furthermore, constraints *(ii)* and

²As mentioned in Chapter 1, the JSR-14 prototype compiler itself, which is written in GJ, includes such an operation!

(iii) break backward compatibility with GJ, an important property to maintain if NEXTGEN is to be incorporated into Java.

To address these threats to safe instantiation, we can augment the constraints on type parameters to include `with` clauses, in which we specify the constructors required by each type parameter. `with` clauses for Generic Java were suggested informally by Agesen, Freund and Mitchell in [2]. We will model them formally in Chapter 9. By specifying a `with` clause on a type parameter, instantiations of the parameter that do not meet the constraints of the `with` clause are prevented. For example, suppose we wanted to define a generic class `Sequence<T>`, with a factory method `newSequence(int n)` that produced a new sequence containing `n` default elements of type `T`. Naturally, the method `newSequence` will have to construct new elements of type `T`. We can use a `with` clause to specify a legal constructor to call as follows:³

```
class Sequence<T with {init()}> {
  public Sequence() {...}
  ...
  public static Sequence<T> newSequence(int n) {
    if (n == 0) return new Sequence<T>();
    else return Sequence<T>.newSequence(n - 1).cons(new T());
  }
}
```

The `with` clause on type parameter `T` requires that every instantiation of `T` include a zeroary constructor, preventing instantiations such as, *e.g.*, `Sequence<Integer>`, but allowing instantiations such as `Sequence<String>` or (because of the zeroary constructor defined in class `Sequence`), `Sequence<Sequence<String>>`. Notice that a separate generic class may instantiate these type parameters with type parameters of its own. For example, we might have:

```
class SequenceBox<R with{init()}> {
  Sequence<R> value = new Sequence<R>();
  ...
}
```

³Unlike GJ, static members in NEXTGEN can be within the scope of type parameters, preventing us from having to declare a method-level type parameter in the definition of `newSequence`.

In this example, the instantiation of type parameter `T` in class `Sequence` with type parameter `R` in `SequenceBox` succeeds because the constructors specified in the `with` clause of `R` contain those specified in the `with` clause of `T`. Because of separate class compilation, we have no information other than the `with` clause of `R` to ensure that instantiation `Sequence<R>` is safe.⁴

The other potential difficulty with `new` expressions in `NEXTGEN` is the possibility of instantiating type parameters with abstract classes. Fortunately, `with` clauses allow us to escape this problem easily. Because the only purpose of `with` clauses in `NEXTGEN` is to allow for `new` operations, we can require without loss of expressiveness that every instantiation of a type parameter that includes a `with` clause must be concrete, guaranteeing safe instantiation. Unfortunately, once we extend `NEXTGEN` to include first-class genericity, this simple solution is no longer sufficient.

3.4 MIXGEN and Safe Instantiation

In the `MIXGEN` extension to Generic Java, support is included for the occurrence of naked type parameters in any context where an ordinary type can occur, including even the `extends` clause of a class definition. As explained in Chapters 8 and 9, the ramifications of this seemingly benign extension are profound in terms of analysis, design, and compatible implementation on the JVM. Additionally, any extension of Java with support for mixins, either through generic types or through a separate facility such as in `Jam`⁵ must address several significant implications for safe instantiation. In particular, the following three threats to safe instantiation arise:

1. The instantiation of a superclass might not contain a constructor matching the signature of a super-constructor call in a mixin constructor.

⁴The semantics of `with` clauses are also complicated by the fact that they are within scope of the type parameters of a class definition, requiring substitution of actual for formal parameters before checking for matching parameters. A formal semantics is presented in Chapter 9.

⁵`Jam` is an extension to Java 1.0 that includes a form of mixin. It was first presented in [7].

2. A mixin’s parent may be instantiated with an abstract class, and an inherited abstract method that is not overridden may be invoked on the mixin.
3. A mixin may “accidentally” override a method inherited from an instantiation of its superclass, and the overridden method may contain an incompatible type signature.

In MIXGEN, the first two problems can be handled by `with` clauses, just as the two analogous problems are handled in NEXTGEN. However, one complication in the case of MIXGEN is that we want to allow instantiated superclasses of concrete mixins to be abstract. Because we still need to call super-constructors in the constructor definitions of mixins, we need to specify valid constructor signatures in the `with` clauses of parametric parent types, and therefore we cannot require the instantiation of every type parameter including a `with` clause to be concrete. But then the instantiation of a type parameter with an abstract class may match all of the constructors specified in the `with` clause and still cause an error when an inherited abstract method (that is not overridden) is invoked on a mixin instantiation. To solve this problem, we must extend `with` clauses so that, in addition to specifying the required constructors of a type parameter, they also include the set of allowed abstract methods in an instantiation.⁶ For example, suppose we define a generic mixin `ScrollPane<T>` intended to extend any subtype of an abstract class `Pane`. We may want to allow a superclass to declare an abstract method `scroll` that is overridden in `ScrollPane`. We can allow that as follows:

```
class ScrollPane<T with {init(), abstract scroll()}> extends T {
  public ScrollPane() { super(); ... }
  public void scroll() {...}
  ...
}
```

Then every instantiation of the parent type of `ScrollPane` must include a zeroary constructor, and may not include any abstract methods other than `scroll()`. The

⁶Note that this extension of `with` clauses maintains backward compatibility with NEXTGEN.

instantiation of a type parameter that includes a `with` clause may not include abstract methods not specified in the `with` clause. If there is no `with` clause, the type parameter cannot occur as the parent type of a class, and no new instances of it can be constructed, so we can safely allow it to be instantiated with an abstract class.

The third problem for mixins is more subtle. Because the programmer defining a mixin does not know all of the instantiations of the mixin’s parent that will exist at runtime, there is no way that he can know exactly what methods will be inherited at runtime. Therefore, there is no way he can prevent the possibility of “accidentally overriding” an inherited method. This problem was first discussed by Flatt, Krishnamurthi, and Felleisen, in [28]. In that work, accidental overriding was presented as undesirable in the context of component-based software engineering; a programmer should never override the functionality of a class unless he intends to do so. But, in the context of first-class genericity, accidental overriding is also undesirable for a more fundamental reason: it is intrinsically incompatible with both type soundness and safe instantiation. To see why, one only needs to consider that if an accidentally overridden method is invoked on a mixin instantiation, the return type of the method may be incompatible with the return type in the static type of the receiver, breaking subject reduction, and violating safe instantiation. Furthermore, in a language like Java, which allows for separate class compilation, the possibility of instantiating type parameters with yet other type parameters eliminates all hope of statically detecting all such overrides.

The original solution to accidental overriding, as proposed by Felleisen et. al. and demonstrated in the toy language MIXEDJAVA, is “hygienic mixins” [28]. Instead of allowing accidental overriding, hygienic mixins allow a mixin instantiation to contain multiple methods of the same name, with resolution of method application dependent on context. Hygienic mixins have been criticized by Ancona, Lagorio, and Zucca, which they say, leads to “ambiguity problems typical of multiple inheritance”

[7].⁷ But in light of the fact that hygienic mixins are a necessary condition for safe instantiation, we believe they are a necessary feature of any production language supporting first-class genericity. In MIXGEN, generic types are leveraged to simplify the semantics of hygienic mixins as compared with their incarnation in MIXEDJAVA. Additionally, it is explained in Chapter 8 how the MIXGEN formulation of hygienic mixins can be implemented compatibly on the JVM, making them an attractive candidate for a production extension of Java.

3.5 Conclusion

As the above case studies have shown, adhering to the Safe Instantiation Principle when formulating a generic extension of Java is deceptively difficult. Nevertheless, we believe that adherence to this principle is necessary to ensure that generic types will help to improve the robustness and reliability of Java programs. We hope that the considerations presented here can be of use to other designers when composing new additions to object-oriented type systems.

⁷Consequently, hygienic mixins have been omitted from the Jam language. If an accidental overriding occurs in which the signatures of the methods are incompatible, the program is rejected by the type checker. More seriously, if the type signatures of the methods happen to match, the inherited method will be overridden without signaling an error, causing an accidental altering of the semantics of the program. We believe that both of these problems are incompatible with the construction of reliable large-scale software systems.

Chapter 4

NextGen Compiler Design

In this chapter, we discuss the design of the NEXTGEN compiler, and how it manages to support type dependent operations without sacrificing compatibility with legacy Java bytecode or the Java virtual machine.

The NEXTGEN formulation of generic types for Java is an implementation of the same source language as GJ, albeit with fewer restrictions on program syntax. In fact, NEXTGEN and GJ were designed in concert with one another [15, 21] so that NEXTGEN would be a graceful extension of GJ. We call this common source language Generic Java. In essence, Generic Java is ordinary Java (JDK 1.3/1.4) generalized to allow class and method definitions to be parameterized by types.

4.1 Generic Classes

In Generic Java, class definitions may be parameterized by type variables and program text may use generic types in place of conventional types. A *generic type* consists of either a type variable or an application of a generic class/interface name to type arguments that may also be generic. Specifically, in a class definition (§8.1 of the JLS [33]), the syntax for the class name appearing in the header of a class definition is generalized from

Identifier

to

Identifier {<TypeParameters>}

where

```

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

```

and braces {} enclose optional phrases. For example, a vector class might have the header

```
class Vector<T>
```

Interface definitions are similarly generalized. In addition, the definition of `ReferenceType` (§4.3 of the JLS) is generalized from

```
ReferenceType  ⇨  ClassOrInterfaceType | ArrayType
```

to

```

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

```

Finally, the syntax for new operations (§15.8 of the JLS) is generalized to include the additional form

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

In essence, a generic type (`ReferenceType` above) can appear anywhere that a class or interface name can appear in ordinary Java, except as the superclass or superinterface of a class or interface definition. In these contexts, only a `ClassOrInterfaceType` can appear. This restriction means that a “naked” type variable cannot be used as a superclass or superinterface.

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

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

where `A` is the element type of the list and `B` is a singleton¹ ordering class for `A`. Static members of a generic class create “holes” in the scope of all the enclosing type

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

abstractions. A static nested class can be generic but all of the type variables of the class must be explicitly introduced in the definition of the class.

In a generic type application, a type parameter may be instantiated as any reference type. If the bound for a type parameter is omitted, the universal reference type `Object` is assumed.

4.2 Polymorphic Methods

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

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

where `Type` can be `void` as well as a conventional type. The scope of the type variables introduced in the type parameter list (`TypeParameters` above) is the header and body of the method. When a polymorphic method is invoked, no type instantiation information is required in most cases. For most polymorphic method applications, Generic Java can infer the values of the type arguments from the types of the argument values in the invocation.² Generic Java also provides a syntax for explicitly binding the type arguments for a polymorphic method invocation, but none of the current compilers (GJ, JSR-14, and NEXTGEN) support this syntax yet.

4.3 The GJ Implementation Scheme

The GJ implementation scheme developed by Odersky and Wadler [15, 42] supports Generic Java through type erasure. For each parametric class `C<T>`,³ GJ generates a single erased base class `C`; all of the methods of `C<T>` are implemented by methods of `C` with erased type signatures. Similarly, for each polymorphic method `m<T>`, GJ generates a single erased method `m`.

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

³For the sake of brevity, we will write meta-references to generic type instantiations (e.g., `C<T>`) as if they include only one type argument, so long as it is obvious how to extend the discussion to cases where there are multiple type arguments.

The erasure of a parametric type τ is obtained as follows. If τ is a parametric type instantiation of some class C , then the erasure of τ is C . If τ is a type variable T , the erasure of τ is the (erased) declared upper bound of T . For each program expression with erased type μ appearing in a context with erased type η that is not a supertype of μ , GJ automatically generates a cast to type η .

4.4 Implications of Type Erasure in GJ

The combination of type erasure and inheritance creates an interesting technical complication: the erased signature of a method inherited by a subclass of a fully instantiated generic type (*e.g.*, `Set<Integer>`) may not match the erasure of its signature in the subclass. For example, consider the following generic class:

```
class Set<T> {
    public Set<T> adjoin(T newElement) {...}
    ...
}
```

The compilation process erases the types in this class to form the following base class:

```
class Set {
    public Set adjoin(Object newElement) {...}
    ...
}
```

Now suppose that a programmer defines a subclass of `Set<Integer>`, and overrides the `adjoin` method:

```
class MySet extends Set<Integer> {
    public Set<Integer> adjoin(Integer newElement) {...}
    ...
}
```

which erases to the base class:

```
class MySet extends Set {
    public Set adjoin(Integer newElement) {...}
    ...
}
```

The type of the `newElement` parameter to `adjoin` in the base class `MySet` does not match its type in the base class `Set`.

GJ addresses this problem by inserting additional methods called *bridge methods* into the subclasses of instantiated classes. These bridge methods match the erased signature of the method in the superclass, overloading the program-defined method of the same name. Bridge methods simply forward their calls to the program-defined method, casting the arguments as necessary. In our example above, GJ would insert the following bridge method into the base class `MySet`:

```
public Set adjoin(Object newElement) {
    return adjoin((Integer)newElement);
}
```

Polymorphic static type-checking guarantees that the inserted casts will always succeed. Of course, this strategy fails if the programmer happens to define an overloaded method with the same signature as a generated bridge method. As a result, Generic Java prohibits method overloading when it conflicts with the generation of bridge methods.

4.5 Restrictions on Generic Java Imposed by GJ

Because the GJ implementation of Generic Java erases all parametric type information, GJ restricts the use of generic operations as described above in Section 1. In essence, no operations that depend on run-time generic type information are allowed.

4.6 The NEXTGEN Implementation Scheme

The NEXTGEN implementation of Generic Java eliminates the restrictions on type dependent operations imposed by GJ. In addition, the implementation architecture of NEXTGEN can support several natural extensions to Generic Java, including per-type static fields⁴ in generic classes and interfaces, co-variant subtyping of generic classes

⁴Since a per-type static field is attached to a specific instantiation of a generic class, the definition of such a field does not create a "hole" in the scope of enclosing type abstractions.

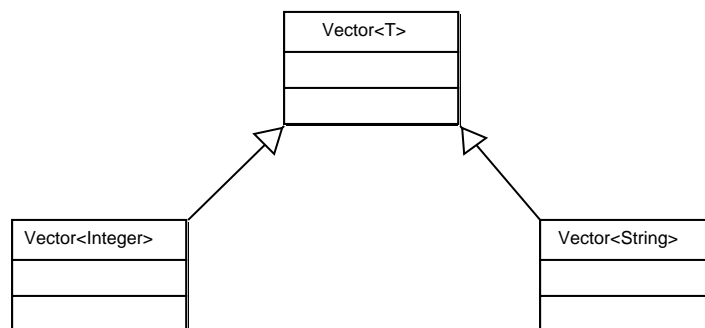


Figure 4.1 : Naive Implementation of Generic Types Over the Existing Java Class Structure

in select contexts, and mixins. In particular, extending NEXTGEN to support mixins is the critical addition necessary to support first-class genericity.

4.7 NEXTGEN Architecture

Roughly speaking, NEXTGEN enhances the GJ implementation scheme by making the erased base class C abstract and extending C by classes representing the various instantiations of the generic class $C<T>$, *e.g.*, $C<Integer>$, that occur during the execution of a given program. These subclasses are called instantiation classes. Each instantiation class $C<E>$ includes forwarding constructors for the constructors of C and code for the type dependent operations $C<E>$. In the base class C , the type dependent operations of $C<T>$ are replaced by calls on synthesized abstract methods called snippet methods [21]. These snippet methods are overridden by appropriate type specific code in each instantiation class $C<E>$ extending C . The content of these snippet methods in the instantiation classes is discussed later in this section.

4.7.1 Modeling Generic Types in a Class Hierarchy

The actual implementation of instantiation classes is a bit more complex than the informal description given above. Figure 4.1 shows the hierarchy of Java classes we would expect the NEXTGEN compiler to create, according to the above description,

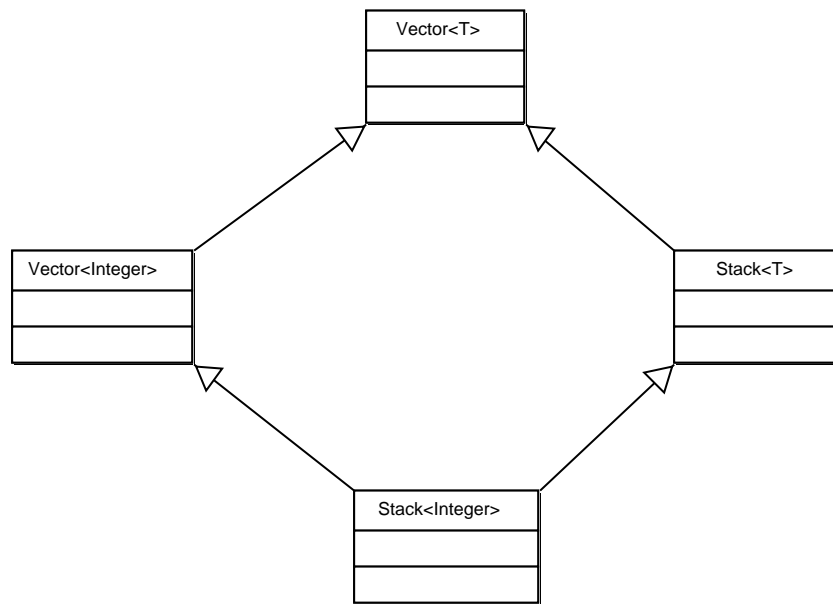


Figure 4.2 : Illegal Class Hierarchy in Naive JVM Class Representation

when compiling the generic type `Vector<T>` and the instantiations `Vector<Integer>` and `Vector<String>`.

When one generic class extends another, the simple JVM class hierarchy given in Figure 4.1 cannot represent the necessary subtyping relationships. For example, consider a generic class `Stack<T>` that extends a generic class `Vector<T>`. Any instantiation `Stack<E>` of `Stack<T>` must inherit code from the base class `Stack` which inherits code from the base class `Vector`. In addition, the type `Stack<E>` must be a subtype of `Vector<E>`. Hence, the instantiation class for `Stack<E>` must be a subclass of two different superclasses: the base class `Stack` and the instantiation class for `Vector<E>`. This class hierarchy is illegal in Java because Java does not support multiple class inheritance. Figure 4.2 shows this illegal hierarchy.

Fortunately, Cartwright and Steele showed how we can exploit multiple interface inheritance to solve this problem [21]. The Java type corresponding to a class instantiation `C<E>` can be represented by an empty instantiation interface `C<E>$` which is

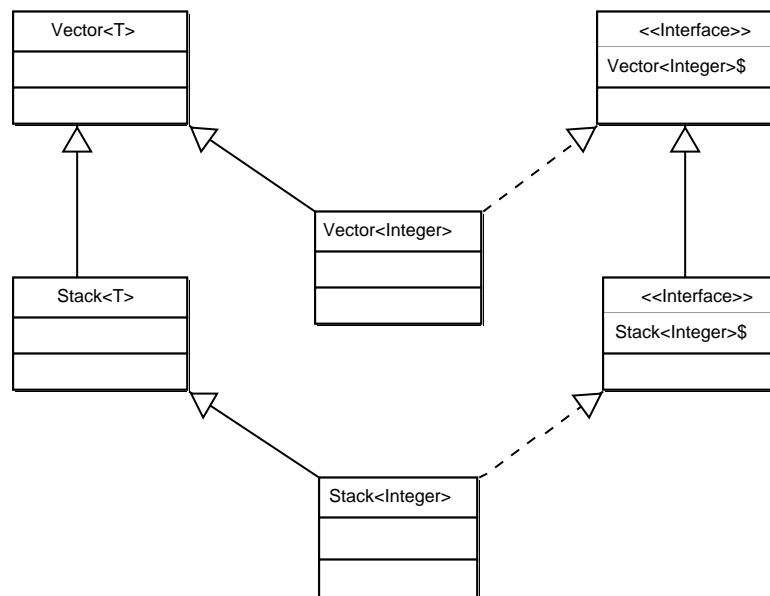


Figure 4.3 : Simple Parametric Type Hierarchy and its JVM Class Representation

implemented by the class $C\langle E \rangle$. The $\$$ at the end of the interface name distinguishes it from the name of corresponding instantiation class and the names of other classes or interfaces (assuming source programs follow the convention of never using $\$$ in identifiers). Since a Java class can implement an interface (actually an unlimited number of them) as well as extend a class, the multiple inheritance problem disappears. Also, since these interfaces are empty, their construction does not appreciably affect program code size. Figure 4.3 represents the same type structure as Figure 4.2 while conforming to the restriction of single class inheritance.

The following rules precisely describe how the NEXTGEN implementation translates generic classes to ordinary Java classes. For each generic class $C\langle T \rangle$:

1. Generate an abstract snippet method in $C\langle T \rangle$ for each application of a type dependent operation.
2. Replace each such application with an application of the new snippet method, passing the appropriate arguments.
3. Erase all types in the transformed class $C\langle T \rangle$ to produce the base class C for $C\langle T \rangle$.
4. For every instantiation $C\langle E \rangle$ of $C\langle T \rangle$ encountered during program execution, generate an instantiation *interface* for $C\langle E \rangle$ and all superclasses and superinterfaces of $C\langle E \rangle$ in which any of the type parameters of $C\langle T \rangle$ occur.
5. For every instantiation $C\langle E \rangle$ of $C\langle T \rangle$ encountered during program execution, generate an instantiation *class* for $C\langle E \rangle$ and all superclasses of $C\langle E \rangle$ in which any of the type parameters of $C\langle T \rangle$ occur.
6. Insert the appropriate forwarding constructors and concrete snippet methods into each instantiation class $C\langle E \rangle$. The concrete snippet methods override the inherited abstract snippet with code that performs the appropriate type de-

pendent operation. The forwarding constructors simply invoke `super` on the constructor arguments.

Generic interfaces are processed similarly, but only steps three and four above are applicable. The naive translation shown in Figures 4.1 and 4.2 suffices for generic interfaces; no supplementary classes or interfaces are required because Java supports multiple interface inheritance.

The most interesting parts in this process are steps four and five. One might think that the compiler could determine an upper bound Φ on the set of possible generic types in a program and generate class files for each instantiation in Φ .⁵ However, early in the process of building a compiler for NEXTGEN, we discovered that the set of all possible generic types across all possible program executions is infinite for some programs. These infinite sets of instantiations are possible because Generic Java permits cycles in the type application graph, *i.e.*, a generic class $C\langle T \rangle$ may refer to non-ground type applications of itself (or type application chains leading to itself) other than $C\langle T \rangle$. For example, consider the following parametric class:

```
class C<T> {
  public Object nest(int n) {
    if (n == 0) return this;
    else return new C<C<T>>().nest(n - 1);
  }
}
```

Consider a program including class $C\langle T \rangle$ that reads a sequence of integer values from the console specifying the arguments for calls on the method `nest` for a receiver object of type $C\langle \text{String} \rangle$. Clearly, the set of possible instantiations across all possible input sequences is infinite.

We solved this problem by deferring the instantiation of generic classes until runtime. NEXTGEN relies on a customized class loader that constructs instantiation classes from a template class file as they are demanded by the class loading process.

⁵Cartwright and Steele proposed this scheme in [21].

The customized class loader searches the class path to locate these template files as needed, and uses them to generate loaded class instantiations. A template class file looks exactly like a class file for a corresponding instantiation class except that the constant pool may contain some references to type variables. The class loader replaces these references (using string substitution) to form instantiation classes. To reduce the overhead of loading instantiation classes on demand, the customized class loader maintains a cache of the template class files that have been read already.

4.7.2 Snippet Methods

As mentioned above, expressions involving type dependent operations are replaced with calls to abstract snippet methods, which are overridden in each instantiation class. The snippet methods in each instantiation class `C<E>` must perform the type dependent operations determined by the types `E`. For `new` operations and `catch` operations, the generation of the appropriate type dependent code is straightforward. But a small complication arises in the case of casts and `instanceof` tests on `C<E>`. In a naive implementation, the body of a snippet method corresponding to a cast or `instanceof` test of type `C<E>` would simply perform the operation on its argument using the instantiation class for `C<E>`. But this implementation fails in some cases because of subtyping: the subclasses of `C<E>` are not necessarily subtypes of the instantiation class `C<E>`. (Recall the example depicted in Fig. 4.3.)

The solution to this problem is to perform the cast or `instanceof` test on the instantiation interface for `C<E>`, since all subtypes of `C<E>` implement it. In the case of a cast, still more processing beyond a cast to the interface for `C<E>` is necessary because the instantiation interface is empty! The result of the cast must be recast to the base class `C`. Casting only to the base class `C` is incorrect because every instantiation of the generic type `C<T>` (such as `Vector<Double>`) is a subtype of `C`.

4.7.3 Extensions of Generic Classes

If a generic class D extends another generic type N where N is not a ground type⁶, a `NEXTGEN` compiler must include concrete snippets for the type dependent operations of N in instantiation classes for D . These added snippets are necessary because the base class for N is the superclass of the base class for D . The requisite snippets are identical to the snippets in the template class for N , specialized with any type bindings established in the definition of D .

4.7.4 Polymorphic Methods

At first glance, polymorphic methods look easy to implement on top of generic classes: they can be translated to generic inner classes containing a single execute method [42]. Each invocation of a polymorphic method can create a generic instance of the associated inner class and invoke an execute method on the arguments to the polymorphic method call. Unfortunately, this translation does not work in general because polymorphic methods can be overridden in subclasses but inner classes cannot. In addition, the overhead of creating a new object on every invocation of a polymorphic operation could adversely impact program performance if polymorphic method calls are frequently executed.

In `NEXTGEN`, the implementation of polymorphic methods is a challenging problem because the type arguments in a polymorphic method invocation come from two different sources: the call site and the receiver type. The call site information is static, while the receiver type information is dynamic. The snippets in the method body can depend on both sources of information.

Our solution to this problem relies on using a heterogeneous translation [42] for polymorphic methods within generic classes. In other words, if the polymorphic method is defined within a generic class, we create a separate copy of the method definition in each instantiation class for the containing generic class. Hence, each

⁶If N is a ground type then D simply extends the instantiation class representing N .

receiver class of a polymorphic method call has a distinct implementation of the method, thereby accommodating method overriding.

The list of type arguments from the call site is passed to the receiver using a special class object whose name encodes the type arguments. If the polymorphic method body involves operations that depend on type parameters from the call site, then it explicitly loads an instantiation of a template class for a snippet environment containing snippet methods for the type dependent operations. The loaded environment class is a singleton containing only snippet code and a static field bound to the only instance of the class.

The overhead of loading snippet environments on every call can be completely eliminated if the JIT compiler performs the appropriate method specialization within the JVM. The JIT merely has to create a separate version of the method for each distinct collection of polymorphic method (call-site) arguments. This specialization is always safe; no analysis is required. We cannot perform this optimization in the class loader because new specialized methods must be created on demand as new classes are loaded; the class loader does not know what specializations will be required by classes that have not yet been loaded.

In the absence of JIT support, we can impose a modest restriction on polymorphic methods to ensure that the overhead of carrying run-time information is negligible. The restriction simply prohibits dynamic polymorphic methods from performing operations that depend on polymorphic method (call-site) type arguments. In other words, snippets in a dynamic polymorphic method can only depend on the type parameters of the receiver, not the type parameters from the call-site. The most common uses of method-level polymorphism conform to this restriction because the method type parameters are only used to support the precise typing of the method. For example, in coding the visitor pattern, the `visit` method in the classes in the composite hierarchy corresponding to the visitor is parameterized by the return type of the operation implemented by the visitor.

In cases where snippets involving polymorphic method (call-site) type arguments are essential, there is a reasonably satisfactory workaround when polymorphic method inheritance is not required. The workaround simply makes the polymorphic method `static` and passes `this` explicitly. For example, the `zip` method on lists, which requires a snippet of the form `new Pair<A,B>`, where `B` is a method type parameter can be implemented as a static method with two type parameters `<A,B>`. Static polymorphic methods can be supported in full generality without much overhead because there is no receiver and hence no dependence on the parametric type information from the receiver. A snippet environment corresponding to the particular instantiation at the call site can be passed simply as an extra parameter to the method.

Of course, this discussion of a workaround is moot if the Java standard embraces NEXTGEN-style genericity. The general implementation described early in this section is very efficient when JIT support (generating type-specializations for polymorphic methods) is available.

Chapter 5

Design Complications

The preceding description of the NEXTGEN architecture neglects three subtle problems that arise in the context of the Java run-time environment: (i) access to private types passed across package boundaries, (ii) access to private generic class constructors from the corresponding instantiation class constructors, (iii) the compatible generalization of the Java collection classes to generic form.¹

Cross-Package Instantiation

The outline of the NEXTGEN architecture given in Chapter 4 does not specify where the instantiation classes of a generic class $C<T>$ are placed in the Java name space. Of course, the natural place to put them is in the same package as the base class C , which is what NEXTGEN does.² But this placement raises an interesting problem when a private type is “passed” across a package boundary [42].

Consider the case where a class D in package Q uses the instantiation $C<E>$ of class $C<T>$ in package P where E is private in Q . If the body of class $C<T>$ contains type dependent operations, then the snippet bodies generated for instantiation class $C<E>$ will fail because they cannot access class E .

The simplest solution to the problem of cross-package instantiation 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 in another package. Although this approach raises security concerns, such widening has a precedent in Java. When an inner class refers

¹The GJ and JSR-14 compiler systems include a version of the Java libraries containing generic type signatures for the Java collection classes. The bytecode for the classes is unchanged.

²In any other package, private classes from the package containing C are inaccessible to the instantiation classes.

to the private members of the enclosing class, the Java compiler widens the visibility of these private members by generating getters and setters with package visibility [33]. Although more secure (and expensive) implementations of inner classes are possible, the Java language designers chose to sacrifice some visibility protection for the sake of performance. This loss of visibility security has not been a significant issue in practice because most Java applications are assembled from trusted components.

The current NEXTGEN compiler uses this simple, expedient solution to the private type visibility problem. It simply widens the visibility of private classes to public when necessary and generates a warning message to the programmer. Nevertheless, NEXTGEN can be implemented without compromising class visibility or execution efficiency.

One solution, as laid out in [21], is for the client class accessing an instantiation $C\langle E \rangle$ of a generic class $C\langle T \rangle$ to pass a snippet environment to a synthesized initializer method in the instantiation class. This environment is an object containing all of the snippets in the instantiation $C\langle E \rangle$. The snippet methods defined in $C\langle E \rangle$ simply delegate snippet calls to the snippet environment. But this solution requires an initialization protocol for instantiation classes that is tedious to manage in the context of separate class compilation. Type arguments can be passed from one generic class to another, implying that the composition of a snippet environment for a given generic type instantiation depends on all the classes reachable from the caller in the type application call graph. The protocol described in [21] assumes that the initialization classes are generated statically, which, as we observed earlier, cannot be done in the presence of polymorphic recursion. This protocol can be patched to load snippet environment classes dynamically from template class files. But any changes to a generic class can force the recompilation of all generic classes that can reach the changed class in the type application call graph.

A better solution to the private type visibility problem is for the class loader to construct a separate singleton class for every snippet where the mangled name of the

class identifies the specific operation implemented by the snippet. The NEXTGEN compiler already uses a similar name mangling scheme to name snippet methods in instantiation classes, eliminating the possibility of generating multiple snippet methods that implement exactly the same operation. In essence, the class-per-snippet scheme replaces a single snippet environment containing many snippets by many snippet environments each containing a single method. The advantage of this scheme is that name mangling can uniquely specify what operation must be implemented, enabling the class loader to generate the requisite public snippet classes on demand (without reading snippet environment class files) and place them in the same package as the type argument to the snippet. The compiler does not have to keep track of the type application call graph because snippets are dynamically generated as the graph is traversed during program execution. To prevent unauthorized access to private classes via a mangled snippet class name, the class loader only resolves references to such a class if (i) the accessing class is in the same package as all of the types embedded in the mangled name or (ii) the accessing class is an instantiation class where each “inaccessible” type in the mangled snippet class name appears as a type argument in the instantiation class. In essence, type arguments are “capabilities” passed from one generic class to another.

The same issue arises in the invocation of polymorphic methods across package boundaries but the class loader can enforce a similar “capability” based policy in resolving references to snippet environment instantiation classes.

One direction for continuing research is to modify the existing NEXTGEN compiler to use per-snippet classes to implement snippets and determine how the performance of this implementation compares with the current, less secure implementation. Per-snippet classes require an extra static method call for each snippet invocation. Again, with explicit JIT support, this overhead could be completely eliminated.

Handling Private Generic Constructors

Since we implement a generic class as an erased base class augmented by instantiation subclasses, private constructors in generic class definitions cannot be implemented by private constructors in the erased base classes. The corresponding constructors in instantiation classes must perform `super` calls on their counterparts in the base class. Consequently, the NEXTGEN compiler relaxes the protection on private constructors in generic base classes. This relaxation does not breach security because generic base classes in NextGen are abstract. The only way to access the constructors of a generic base class is by performing `super` calls in the constructors of immediate subclasses. The class loader can enforce a policy of not loading any immediate subclasses for a generic base class other than instantiation classes.

Extending the Java Collection Classes

One of the most obvious applications of generic types for Java is the definition of generic versions of the Java collection classes. GJ supports such an extension of the Java libraries by simply associating generic signatures with the existing JDK collection classes. To accommodate interoperability with legacy code, GJ allows breaches in the type system of Generic Java. In particular, GJ accepts programs that use erased types in source program text and supports automatic conversion between generic types and their erased counterparts. Using this mechanism, Generic Java programs can interoperate with legacy code that uses erased versions of generic classes, *e.g.*, the collection classes in the existing JDK 1.3 and 1.4 libraries. But this interoperability is bought at the price of breaking the soundness of polymorphic type-checking.³

NEXTGEN cannot support the same strategy because generic objects carry run-time type information. An object of generic type is distinguishable from an object of the corresponding base class type.

³The type system supported by the GJ compiler includes raw (erased) types. When an object of raw type is used in a context requiring a parametric type, the GJ type-checker flags an unchecked operation error, indicating that the program violates polymorphic type-checking rules. GJ still compiles the program to valid byte code, but the casts inserted by the GJ compiler can fail at run-time.

In a new edition of Java supporting NEXTGEN, the collection classes could be rewritten in generic form so that the base classes have the same signatures (except for the addition of synthesized snippet methods) as the existing collections classes in Java 1.4. The base classes for the generic collection classes would extend the corresponding existing (non-generic) collection classes. Given such a library, generic collection objects can be used in place of corresponding “raw” objects in many contexts (specifically those in which there are no writes to parametric fields). Similarly, raw objects can be used in place of generic objects in a few contexts (those in which there are no reads from parametric fields). In some cases, explicit conversion between raw and generic objects will be required, for both run-time correctness and static type correctness. To facilitate these conversions, the new generic collection classes would include methods to perform such conversions.

Because of the distinction between objects of parametric type and objects of raw type, the integration of legacy code with NEXTGEN programs requires more care than the integration of legacy code with GJ programs. But the extra care has a major payoff: the soundness of polymorphic type-checking is preserved.⁴

⁴To preserve type soundness, raw types must be treated more carefully than they are in GJ. In particular, the raw type `C` corresponding to a generic type `C<T>` must be interpreted as the existential type $\exists T \text{ C<T>}$. Hence, any operation with `T` as a result type yields type `Object`. Similarly, any method in a generic class with an argument whose type depends on `T` is illegal. Raw types have been analyzed formally in [36].

Chapter 6

NextGen Compiler Implementation

The NEXTGEN compiler is implemented as an extension of the GJ compiler written by Martin Odersky. The GJ compiler is organized as a series of passes that transform a parsed AST to byte code. We have extended this compiler to support NEXTGEN by inserting an extra pass that detects type dependent operations in the code, encapsulates them as snippet methods in the enclosing generic classes, and generates template classes and interfaces for each generic class. The names assigned to these snippets are guaranteed not to clash with identifiers in the source program nor with the synthesized names for inner classes, because they include the character sequence `$$`, which by convention never appears in Java source code or the mangled names of inner classes. We have also modified the GJ code generation pass to accept these newly generated class names even though there is no corresponding class definition.

The added pass destructively modifies the AST for generic classes by adding the requisite abstract snippet methods and replacing dependent operations with snippet invocations. It also generates the template classes that schematically define the instantiation classes corresponding to generic classes.

A template class file looks like a conventional class file except that some of the strings in the constant pool contain embedded references to type parameters of the class instantiation. These references are of the form `{0}`, `{1}`, `...`. The class loader replaces these embedded references by the corresponding actual type parameters (represented as mangled strings) to generate instantiation classes corresponding to the template.

Both the NEXTGEN compiler and class loader rely on a name-mangling scheme

to generate ordinary Java class names for instantiation classes and interfaces.

The NEXTGEN name-mangling scheme encodes ground generic types as flattened class names by converting:

- Left angle bracket to `$$L`.
- Right angle bracket to `$$R`.
- Comma to `$$C`.
- Period (dot) to `$$D`.

Periods can occur within class instantiations because the full name of a class (*e.g.*, `java.util.List`) typically includes periods. For example, the instantiation class

```
Pair<Integer, java.util.List>
```

is encoded as:

```
Pair$$Ljava$$Dlang$$DInteger$$Cjava$$Dutil$$DList$$R
```

By using `$$D` instead of `$` for the periods in full class names, we avoid possible collisions with inner class names.

6.1 The NEXTGEN Class Loader

When a NEXTGEN class refers to a generic type within a type dependent operation, the corresponding class file refers to a mangled name encoding the generic type. Since we defer the generation of instantiation classes and interfaces until run-time, no actual class file exists for a mangled name encoding a generic type. Our custom class loader intercepts requests to load classes (and interfaces) with mangled names and uses the corresponding template class file to generate the requested class (or interface). A template class file looks exactly like a conventional class file except that the constant pool may contain references to unbound type variables. The references to unbound

type variables are written in de Bruijn notation: the strings $\{0\}$, $\{1\}$, \dots , refer to the first, second, \dots , type variables, respectively. Since the characters $\{$ and $\}$ cannot appear in Java class names, type variable references can be embedded in the middle of mangled class names in the constant pool.

Roughly speaking, the class loader generates a particular instantiation class (interface) by reading the corresponding template class file and replacing each reference tag in the constant pool by the corresponding actual type name in the mangled name for the class. The actual replacement process is slightly more complicated than this rough description because the code may need the base class, interface, or actual type corresponding to an actual type parameter. The precise replacement rules are:

- Replace a constant pool entry of the form $\{n\}$ (where n is an integer) by the name of the class or interface bound to parameter n .
- Replace a constant pool entry of the form $\{n\}$ \$ (where n is an integer) by the name of the interface corresponding to the class or interface bound to parameter n . This form of replacement is used in the snippet code for casts and `instanceof` tests.
- Replace a constant pool entry of the form $\{n\}$ B (where n is an integer) by the base type corresponding to the type bound to the parameter n . (If the type bound to n is not generic, then the base type is identical to the argument type.)
- Process a constant pool entry of the form *prefix*\$\$\$*Lcontents*\$\$\$*Rsuffix* where *contents* contains one or more substrings of the form $\{n\}$ (where n is an integer) as follows. Each substring $\{n\}$ inside *contents* is replaced with the name of the class bound to parameter n , substituting \$\$\$ for each occurrence of “.” (period).

After this replacement, the class file denotes a valid Java class.

Chapter 7

NextGen Performance

The NEXTGEN compiler is maintained and extended as a production software system by the Rice JavaPLT research team. Some features, such as polymorphic methods, have been postponed because we expect that JIT support may be necessary to achieve acceptable performance. Nevertheless, the functionality currently supported is sufficient to measure the general overhead incurred by maintaining run-time support for

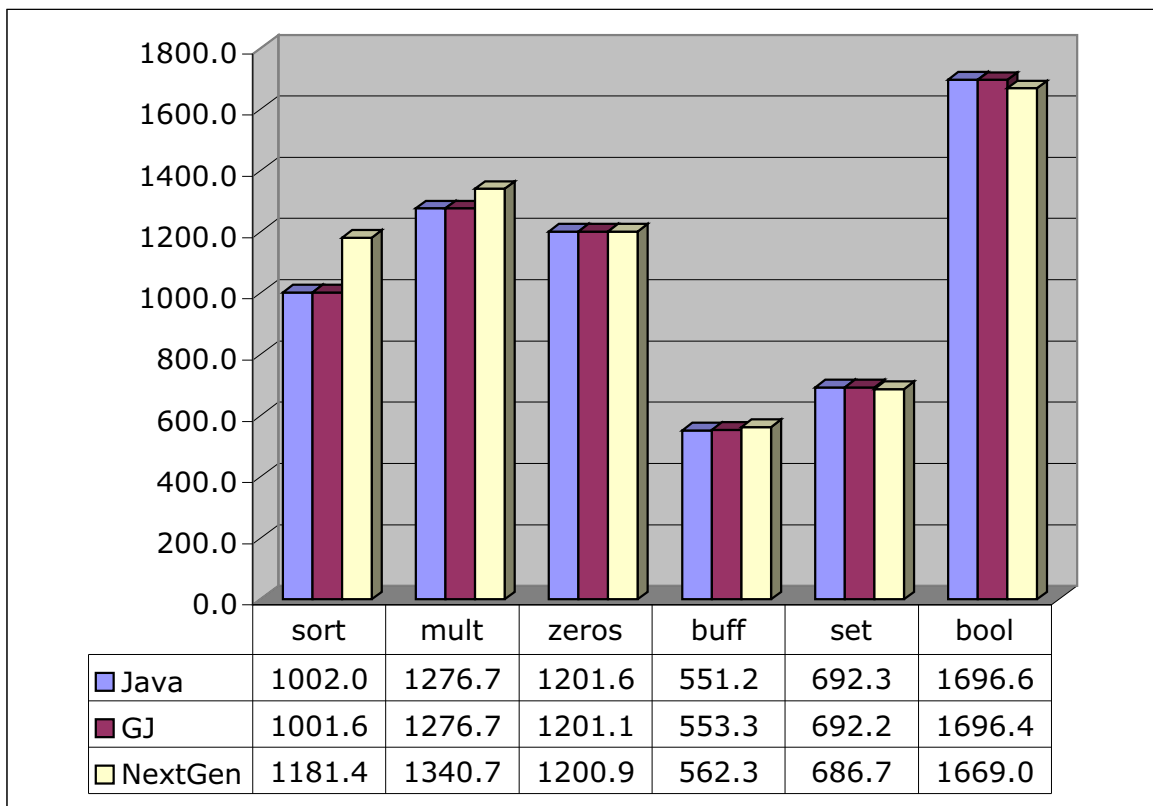


Figure 7.1 : Performance Results for Sun 1.3 Client (in milliseconds)

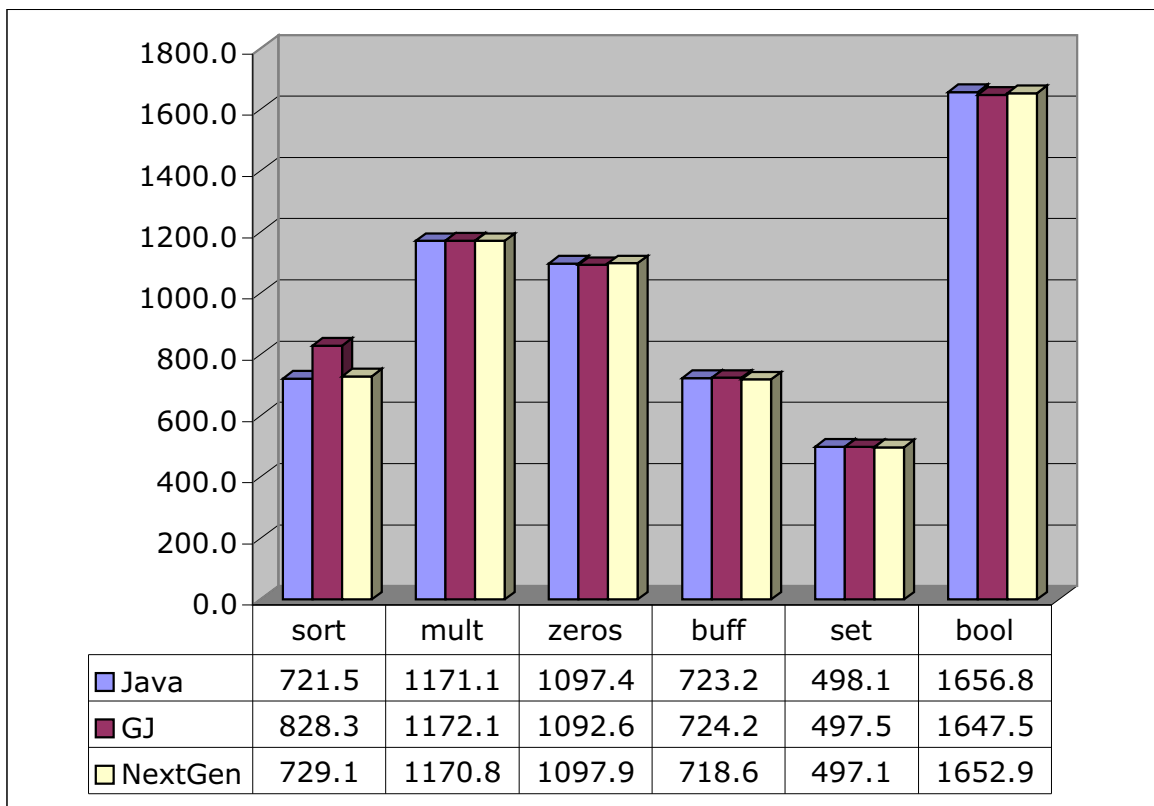


Figure 7.2 : Performance Results for Sun 1.3 Server (in milliseconds)

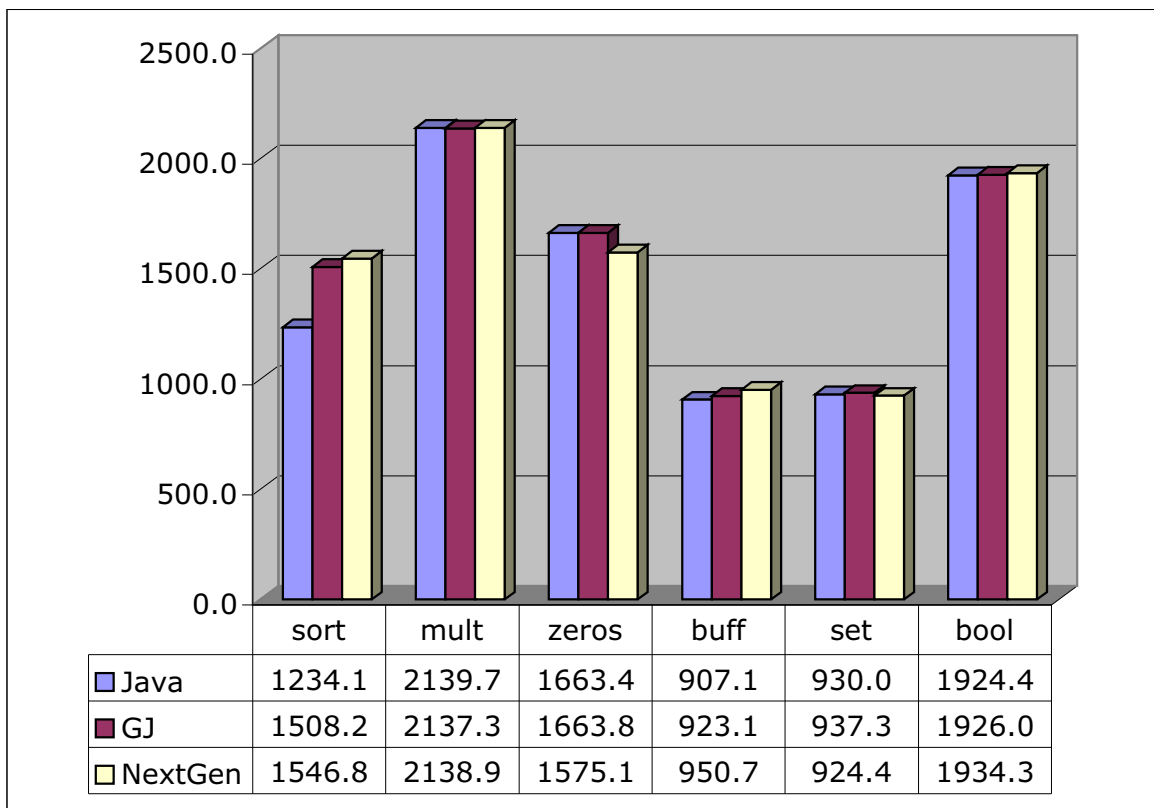


Figure 7.3 : Performance Results for Sun 1.4 Client (in milliseconds)

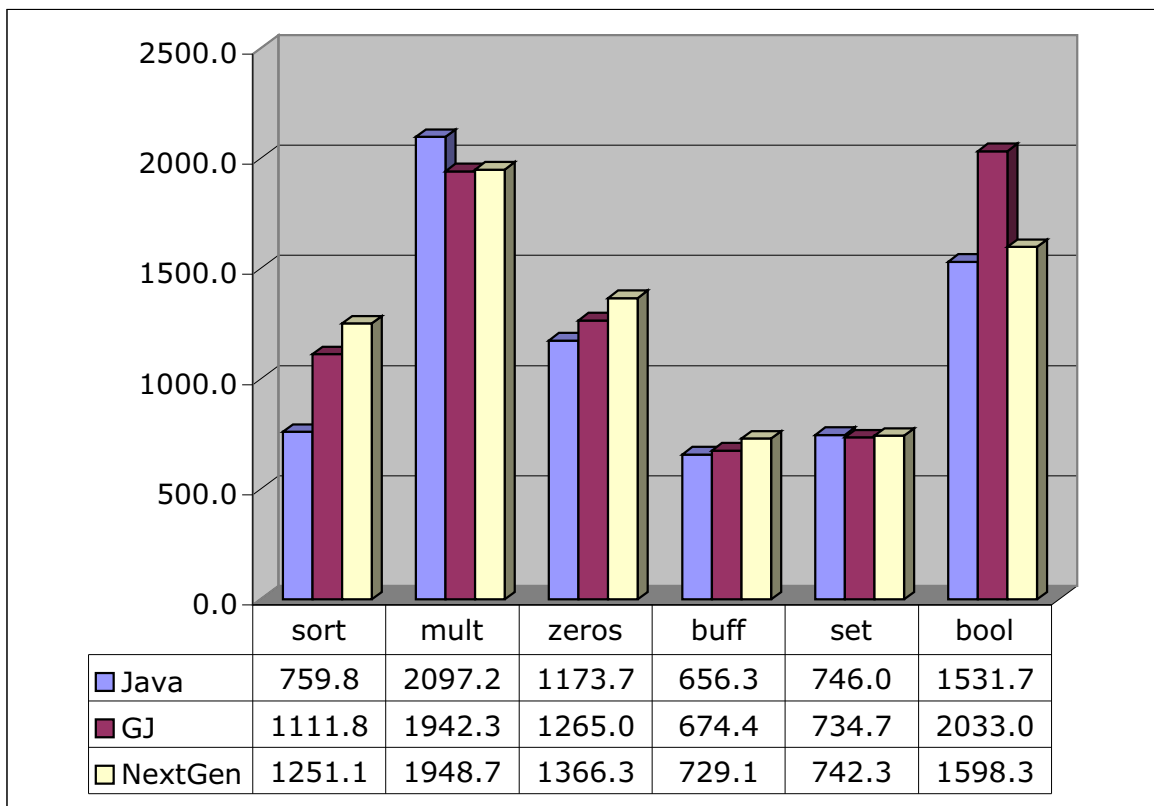


Figure 7.4 : Performance Results for Sun 1.4 Server (in milliseconds)

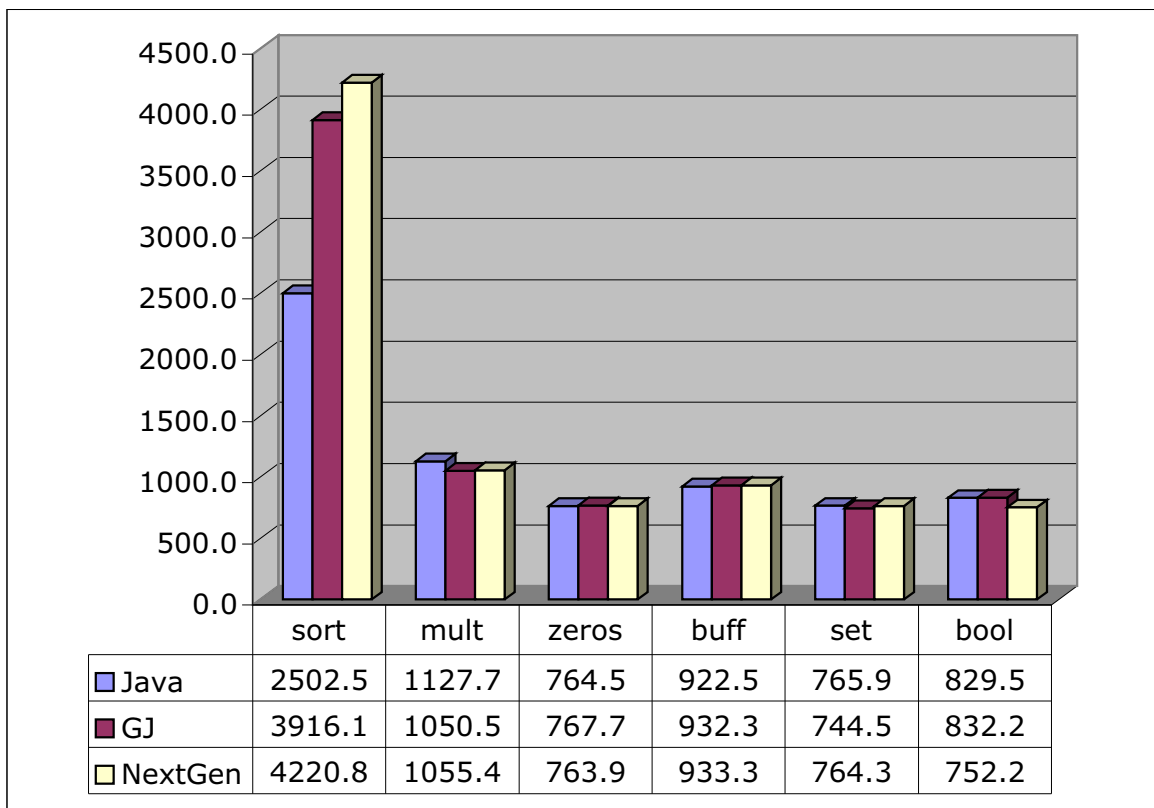


Figure 7.5 : Performance Results for IBM 1.3 (in milliseconds)

generic types. In this chapter, we will show the results of such a measurement.

Because no established benchmark suite for Generic Java exists, we had to construct our own benchmark suite to measure the performance of NEXTGEN. On existing benchmark suites for ordinary Java like JavaSpecMark [2], the performance of NEXTGEN is identical to that of the GJ and JSR-14 compilers, because they all generate the same class files. Our benchmark suite consists of the following programs, which all involve generic types:

- **Sort:** An implementation of the quicksort algorithm on generically typed linked lists, where quicksort is parameterized by the ordering relation for the sort. This benchmark consists of 769 lines of code in 13 classes. 7 of these classes make heavy use of generics.
- **Mult:** A visitor over generically typed binary trees of integers that multiplies the values of the nodes. This benchmark consists of 428 lines of code in 16 classes. 7 of these classes make heavy use of generics.
- **Zeros:** A visitor over generically typed binary trees that determines whether there is any child-parent pair in which both hold the value 0. This benchmark consists of 552 lines of code and 14 classes. 8 of these classes make heavy use of generics.
- **Buff:** An implementation of `java.util.Iterator` over a `BufferedReader`. This benchmark constructs a large, buffered `StringReader`, and then iterates over the elements. This benchmark consists of 305 lines of code in 7 classes. 2 of these classes make heavy use of generics.
- **Bool:** A simplifier of Boolean expressions. This program reads a large number of Boolean expressions from a file, parses them, and simplifies them. The simplification process is organized as a series of passes, each implemented 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.

- **Set:** An implementation of generically typed multi-sets, and set-theoretic operations on them. This program constructs large multi-sets and compares them as they are built. This benchmark consists of 316 lines of code in 6 classes. 2 of these classes make heavy use of generics.

The benchmarks were written in Generic Java specifically to take advantage of the added type checking provided by Generic Java. To facilitate comparison with the GJ compiler and the JSR-14 update of the GJ compiler, all of the benchmarks conform to the restrictions imposed by the GJ implementation of Generic Java.¹

The source code for each benchmark was manually translated to equivalent Java source code. Manual modification was necessary because the source transformation performed by the GJ compiler does not necessarily yield valid ordinary Java code. The GJ compiler performs its own code generation for this reason.² Nevertheless, this manual modification consisted merely of inserting casts and bridge methods as necessary; it had no effect on the number of classes or lines of code. The original and converted source code were both compiled using the JSR-14 compiler. The NEXTGEN compiler was applied to exactly the same source code as the JSR-14 compiler.

The results of these benchmarks for Java, GJ, and NEXTGEN under five separate JVMs are illustrated in Figs. 7.2-7.5. These results were obtained by running each benchmark twenty-one times, for each JVM listed, on a 2.0 GHz Pentium 4 with 512 MB RAM running Red Hat Linux 7.2. Because the results of the first run for each JVM/compiler combination exhibited significant variance, the results of the first run were uniformly dropped. We attribute this variance to the overhead of JVM startup and initial JIT (“just-in-time”) compilation of the code, neither of which is relevant

¹Some type dependent operations in NextGen are not type dependent in GJ, because GJ erases all parametric type information. In particular, all `new` operations on generic types are type dependent in NextGen but not in GJ.

²The bridge methods generated by GJ may rely on the result type for static overloading resolution; the JVM supports this generalization of Java overloading.

to what our experiment is intended to measure. Once the first run was dropped, the variance in the duration of the individual runs for each benchmark was less than 10%.

The results for the JSR-14 compiler also apply to the GJ compiler, because the class files generated by these compilers are functionally identical. The only differences are that (i) JSR-14 inserts an additional entry into the constant pool, and (ii) JSR-14 by default sets the class file version to 46.0 (the new Java 1.4 version tag). Neither of these differences should have any impact on performance.

The most striking feature of these results is that overhead of supporting run-time generic types is insignificant, even for programs that make heavy use of it. In fact, even the small overhead that NEXTGEN exhibits for some benchmarks is dwarfed by the significant range in performance results across JVMs and the potential tuning of JIT compilers to optimize the code for generics.

The small overhead in some of the benchmarks can be explained by considering what costs are incurred by keeping the run-time type information. Once an instantiation of a template class is loaded into memory, the only overhead of genericity is the extra method call involved in invoking a snippet. Because most of the operations in an ordinary program are not type dependent operations, this small cost is amortized over a large number of instructions.³

On advanced JVMs that perform dynamic code optimization, even type dependent operations incur little overhead because many of the snippet operations are inlined, eliminating the extra snippet call. If NEXTGEN were adopted as the Java standard, we anticipate that dynamic code optimization would be tuned to eliminate essentially all snippet call overhead. Moreover, run-time generic types provide explicit guidance on where code specialization is likely to be profitable—potentially making NEXTGEN code *more efficient* than either GJ (JSR-14) or ordinary Java code.

Our benchmark suite was specifically designed to make heavy use of generic types,

³There is one anomaly in the benchmark results that we do not understand, namely the slow running times for the code generated by the JSR-14 and NextGen compilers for Generic Java source on the Sort benchmark on the IBM 1.3 JVM. Perhaps the fact that the JSR-14 and NextGen bytecode relies on result types for static overload resolution interferes with some code optimization in the JIT in this particular JVM.

and yet, even in this context, supporting run-time generic type operations added little performance overhead. Therefore, we are confident that the performance impact of supporting run-time generic types for typical Generic Java programs will be negligible. On the other hand, the robustness and maintainability of many programs would be greatly enhanced in comparison with ordinary Java. In addition, the absence of consistent overhead across JVMs for any of the benchmarks suggests that code optimization sensitive to the performance demands of NEXTGEN could completely eliminate the overhead.

Chapter 8

Implementing First-Class Genericity

We now turn our attention to the question of how an object-oriented type system can be extended to support first-class genericity. Although we are focusing our attention on adding first-class generic types to Java, essentially the same design issues arise in supporting first-class genericity in any strongly-typed object-oriented language with nominal subtyping. We have designed the MIXGEN programming language as an extension of the NEXTGEN formulation of Generic Java with support for first-class generic types. Although the NEXTGEN language defined by Cartwright and Steele explicitly excludes mixins,¹ its implementation architecture can be gracefully extended to support them. But before we describe the syntax and semantics of MIXGEN, we need to discuss a minor extension to NEXTGEN that addresses a weakness in the design of Generic Java.

8.1 Preliminaries

In Generic Java [21], the definition of a generic class implicitly imposes some restrictions on the type arguments that may be used to instantiate the class. In particular, the use of a `new` operation on a naked type variable in a generic class forces the corresponding type argument to be a concrete class and to provide a constructor with a matching signature. In the spirit of safe instantiation, these restrictions can and should be made explicit.

The simplest solution is to allow only zeroary constructors to be invoked by `new` operations on a naked type variable and force classes bound to type parameters on

¹NEXTGEN was designed to address the requirements codified in JSR-14, which does not include mixins.

which naked `new` operations are performed to include a zeroary constructors. Generic C# follows this approach.

A more powerful solution is to support an optional `with` clause in the declaration of a type variable that specifies a list of constructor signatures. The `with` clause restricts the binding of the type variable to concrete classes with the specified constructors. For example, the generic class header

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

requires that any argument support a zeroary constructor. A `new` operation can be performed on a naked type variable only if the declaration of the type variable includes a `with` clause with a matching constructor.

8.2 MIXGEN Extensions

MIXGEN is a proper extension of this new version of NEXTGEN, with one exception: *covariant method return types are not supported*. This exception is necessary for type soundness. For example, suppose we have a mixin class `M<T>` where the bound on the mixin parent `T` includes an abstract method `m` with return type `Object`. Then suppose that some class `E` that satisfies the declared bounds on `T` defines `m` with return type `String`. `M` itself overrides `m` with return type `Object`. Then the mixin instantiation `M<E>` will not be a valid class file because it would include a contravariant return type for method `m`!

All existing NEXTGEN programs that do not include covariant method return types are valid MIXGEN programs, with the same semantics. MIXGEN extends NEXTGEN as follows:

1. The superclass specified for a generic class may be a type variable. When the superclass of a generic class is a type variable, then the declaration of the type variable must include a `with` clause.

2. A `with` clause may include `abstract` and `final` declarations for some of the methods in the bounding type `I` for the type variable. A type `T` may be bound to such a type variable only if it is a subtype of `I` and its `abstract`/`final` methods are a subset of those declared as `abstract`/`final` in the `with` clause.

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

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

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

8.3 Cyclic and Infinite Class Hierarchies

Both ordinary Java and Generic Java force the type hierarchy for a program to form a DAG (directed acyclic graph) under the subtyping relation. In Generic Java, all parameterization is erased from class definitions in forming this hierarchy—reducing generic types to the corresponding base (erased) types. This simplification is justified by the following observation: if a collection of generic types has a cycle, then the collection of corresponding base classes also has a cycle.²

²This fact is implicitly assumed in [34].

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

Example 1. A class extending a mixin application can extend itself:

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

Example 2. Classes constructed from mixin applications can form arbitrarily long cycles, as the following example demonstrates:

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

This program creates the cycle:

$$D_0 <: C\langle D_{k-1} \rangle <: D_{k-1} <: \dots <: D_1 <: C\langle D_0 \rangle <: D_0$$

Example 3. Recursion in class definitions involving mixins raises the possibility of infinite class hierarchies. The following program creates an infinite class hierarchy:

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

consisting of the types

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

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

One simple way to enforce the DAG constraint in the context of mixins is to prohibit generic classes from parametrically extending mixin classes, *i.e.*, prohibit class definitions such as `class D<X>` below:

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

In the section on type soundness, we show that this restriction guarantees that the DAG property holds.

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

```
class D<X> extends C1<C2<...<Ck<X>...>>
```

where C_1, C_2, \dots, C_k are generic classes that have already been identified as (primitive or secondary) mixins,³ and iterating this process until no new secondary mixins are identified.

The DAG checking algorithm simply inspects all of the class headers

³This notation ignores extra type arguments that are not part of the of superclass chain for `D<X...>`

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

in the program⁴ and applies two different reductions to this collection of headers.

- First, it erases all type arguments other than the superclass arguments of mixins and secondary mixins.
- Second, it erases all mixin applications, all secondary mixin applications, and eliminates mixin definitions from the list of class headers.

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

The first reduction in the preceding algorithm is motivated by the fact that type arguments other than superclass arguments do not affect the structure of the type hierarchy of a program. This is the same observation used to justify erasing all type parameters from Generic Java programs before checking for cycles in the type hierarchy.

The second reduction is based on the observation that mixins simply provide a richer mechanism than conventional class definitions for extending a superclass by a subclass. The classes constructed by mixins and secondary mixins can safely be merged with their superclasses in a class hierarchy graph without changing the status of the hierarchy as a DAG or non-DAG. From an intuitive perspective, mixins simply annotate classes with additional members.

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

To illustrate how this explicit DAG checking algorithm works, consider the three examples at the beginning of this subsection of class definitions involving mixins that

⁴We are ignoring the implemented interfaces, because they must meet exactly the same constraints as they do for Generic Java.

can create class hierarchies that are not DAG's. In the first example, applying the analysis to the `extends` clause for class `D` produces the reduced class header `class D extends D`, which is rejected as cyclic. In the second example, the algorithm produces the reduced collection of class headers `class D0 extends Dk-1, ..., class D2 extends D1, class D1 extends D0`, which is rejected as cyclic. In the third example, the algorithm reduces the class headers to the header `class C<X> extends C<C<X>>`, which is rejected as cyclic.

8.4 Accidental Overriding

For a given mixin

```
class M<T extends N with {...}> extends T
```

`M` can be instantiated with many different superclasses, but any potential instantiation must satisfy the declared bounds on `T`. Unfortunately, these declared bounds cannot prevent unintended interference between a mixin instantiation `M<A>` and its superclass `A`. In particular, `M<A>` may *accidentally override* a method of `A` that is not a member of the bounding class `N`—breaking the superclass.

Accidental method overriding is difficult to detect because the superclass of a mixin is not known when the mixin is compiled and locally type checked. When a generic class is instantiated somewhere in a program, each type argument can potentially flow anywhere in the program through type application. Because the parent of a mixin may be instantiated with a type variable (or another mixin that is not ground), it is impossible to determine via local analysis when a particular mixin instantiation will result in an accidental override at run-time. For example, consider Figure 8.1.

In this example, each mixin definition is type correct in isolation. Moreover, the method invocation expression `new DFactory<C<Object>>().create()` is type correct given the headers of generic classes `C`, `D`, `DFactory`, and `E`. However, the expression `new DFactory<C<Object>>().create()`, generates the mixin instantiation

```

interface I {
    Object f();
}

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

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

class DFactory<T extends I with {init()}> {
    D<T> create() { return new D<T>(); }
}

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

```

Figure 8.1 : A mixin with accidental overriding

`D<C<Object>>`, in which method `m()` in `D` accidentally overrides method `m()` in `C`. When the program is executed, loading class `D<C<Object>>` will raise an error because the method return types for `m()` in `C` and `D` are inconsistent.

In languages supporting separate class compilation, such as (Generic) Java and C#, we cannot detect all such accidental overrides during compilation. Class `DFactory` may be compiled separately from the method call

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

Because a generic class can pass its type arguments to other generic classes, only a whole program analysis can detect all accidental overrides. In the context of separate class compilation, the best that we can hope to do is detect some accidental method overridings at load time. Furthermore, since the error messages could involve code in classes for which the source is not available, developers would be unable even to anticipate when such errors would occur. Thus, accidental overriding is incompatible with the Principle of Safe Instantiation.

8.5 Hygienic Mixins

From the preceding discussion, it is clear that we cannot support the local type checking of mixins (formulated as generic classes) if accidental method overriding occurs. But accidental method overriding is unavoidable if the semantics of mixins is defined by simple syntactic expansion. Fortunately, there is alternative semantics for mixins, developed by Flatt, Krishnamurthi, and Felleisen, that prevents accidental method overriding [28]. In essence, this semantics changes the protocol for method lookup so that accidental method overridings are ignored.

This semantics has not been previously been explicated in the context of mixins as generic classes, so we must develop some new machinery to apply it to this context. As a prelude, let us step back and consider how static type checking meshes with dynamic dispatch in nominally subtyped object-oriented languages. When a method invocation is statically type-checked, the invoked method's signature is resolved based on the static type T of the receiver. In Java, the byte code generated for the method call refers to this signature and the type T . The actual method code invoked at run-time conforms to this type signature because nominally subtyped object-oriented languages force overriding methods to have a compatible signature. Hence, we can model dynamic dispatch as a lookup process that starts in the class hierarchy at the static type of the receiver and walks *down* the path toward the receiver's class to find the method with a matching signature that is closest to the receiver's run-time

class. Alternatively, we could start the search in the receiver's class and walk *up* the superclass chain in the type hierarchy to find the first method definition with a signature matching the signature specified in the method call. The soundness of the type system ensures that these two lookup procedures are observably equivalent.

In MIXGEN, the class hierarchy is not resolved until run-time, so a more sophisticated semantics must be used. To find the method matching signature S in type T , the semantic model starts with the definition provided by the static type T and searches down the hierarchy toward the run-time type for valid overrides of S . If a mixin class on this path does not include S in its superclass interface, then the search stops *because the method is hidden in the mixin instantiation and all of its instantiated subclasses*. The resolved method is the last matching method encountered before reaching the mixin class that hides the method. Of course, if no hiding mixin class is encountered in this traversal, the resolved method is the same as it is in conventional Java. For example, in Figure 8.1, in method

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

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

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

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

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

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

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

8.5.1 MIXGEN and MIXEDJAVA

Our formulation of a hygienic semantics, that equates views with static types, is significantly different than the formulation in MIXEDJAVA, where views are carried with values at run-time. The tagging of values with types significantly affects the semantics of the language. A naive translation of a MIXEDJAVA program into MIXGEN will not generally preserve the meaning of the original program. Consider the following MIXGEN code fragment:

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

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

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

has no effect, because `m` is *overridden* in each application of the mixin `C` in `C<C<A>>`. The invoked method is the code for `m` in the second mixin application. Hence, if `m`

subsequently invokes the method `n`, the static type of `this` is `C<C<A>>` implying that the version of `n` introduced in `C<C<A>>` will be invoked.

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

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

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

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

MIXEDJAVA cannot name the type `T` or `M<T>`. As a result, MIXEDJAVA does not support the precise typing of polymorphic recursion or other programming patterns that introduce cycles in the mixin type application graph.

8.6 Implementing MIXGEN in the JVM

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

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

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

To add mixins to NEXTGEN, we must extend the NEXTGEN compiler to translate mixins to template class files and modify the class loader to enforce mixin hygiene. If hygiene were not an issue, the existing NEXTGEN class loader would suffice for this purpose. But this naive approach does not prevent accidental overriding. The class loader must systematically rename class methods to enforce hygiene. We describe how renaming can be done in Section 8.6.1 below.

8.6.1 Enforcing Hygienic Method Invocation

Our hygienic semantics for MIXGEN formulates method invocation as a downward search from the static type of the receiver, allowing method overriding in a mixin instantiation only when the method is included in the bounding type of the superclass. This search is more elaborate than the standard dynamic dispatching mechanism employed in the run-time systems for mainstream object-oriented languages (Java, C#, and C++) which effectively searches up the class hierarchy from the class type of the receiver.⁵ However, we can efficiently implement our “downward search” semantics using conventional dynamic dispatch—provided that we use a customized class loader similar to the one already present in NEXTGEN. We will show how we can use such a class loader to systematically rename method names to prevent accidental overriding.

⁵The method table in each class object reduces this search to a table lookup.

The renaming of methods in mixin classes in the JVM is a subtle issue. The class loader obviously must rename the *new* methods introduced in mixins to avoid accidental overriding. In fact, all *new* methods introduced in classes must be renamed to avoid accidental overriding because a subclass of a mixin instantiation can reintroduce a method that was hidden by the mixin.⁶ Similarly, any references to a renamed method including valid overrides must be renamed to reflect the new name. For each method invocation based on a class type, the class loader can determine the class on the ancestor hierarchy where that method was introduced and perform the appropriate renaming. But this transformation does not work for method invocations based on interface types. Such a method invocation may resolve to several different renamed methods depending on the class of the receiver.

We can solve the problem of interface-based method dispatch by adding a forwarding method for each method introduced in a class that implements an interface method.⁷ The forwarding method maps `invokeinterface` calls on the (renamed) interface method to the corresponding method introduced in the class.

But there is an additional subtlety here⁸ because a class constructed using a series of mixins can implement multiple instantiations of the same generic interface. Each mixin application can only implement one instantiation of a generic interface,⁹ but a sequence of mixins can collectively implement several such instantiations (see the program fragment given below). Both `NEXTGEN` and `MIXGEN` use erased method signatures at run-time to maximize code sharing. Essentially all of the code for a generic class is located in a shared base class that is extended by each instantiation of the generic class. If a method in the generic interface has the same internal name

⁶Even the methods in classes that are not subclasses of a mixin instantiation must be renamed to avoid colliding with method dispatches based on interface types.

⁷If a method introduced in a class implements more than one interface method (which can happen when the same method signature appears in different interfaces), then a forwarding method must be generated for each such interface method.

⁸Detected by Martin Odersky in an earlier draft of this paper.

⁹This is a restriction of Generic Java that could be eliminated by a more heterogeneous implementation strategy, but it does not appear to be an issue in practice.

(after renaming) in different various instantiations supported by the same class, then the forwarding methods for different instantiations of the same generic interface will collide.¹⁰

To eliminate this problem, we can view generic classes as introducing new methods in each instantiation class and distinctly rename the methods in these instantiation classes. This elaboration of our renaming scheme requires a modest revision to the NEXTGEN implementation architecture. The instantiation interfaces previously used only for subtyping purposes (to support generic type casts and `instanceof` tests) must be augmented by the methods introduced in the generic instantiation. The instantiation classes corresponding to the instantiation interfaces must implement these methods by forwarding them to the corresponding methods in the base class.

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

The simplest renaming scheme that satisfies all of above constraints is to prefix the name of every new method introduced in any class (or interface) `C` by a qualifier consisting of the full name of the class `C` followed by a `$` sign. If `C` is generic, the name of `C` in the prefix must be mangled in names introduced in instantiation classes and interfaces.¹¹ If the mangled name includes free type parameters, then the complete mangled name will not be resolved until an instantiation of the class is loaded. The class loader processes every method descriptor for a class based dispatch (`invokevirtual`) in the constant pool mapping the method name to its prefixed form.

¹⁰Collisions can still happen in more heterogeneous implementations without erasure because the signature of a method may not mention the type parameters.

¹¹The new method names in the base class of a generic class (in addition to the instantiation classes) must also be prefixed by the class name.

```

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

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

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

```

Figure 8.2 : An example of a mixin implementing two instantiations of the same interface.

Method descriptors for interface based dispatches are also prefixed by the fully qualified name of the interface. If the class is a mixin instantiation, then a forwarding method must be generated for each *new* method (introduced in the mixin) mapping its external name to its prefixed name.

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

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

In this manner, method invocation *behaves* as if it were performed by a downward

search from the static type of the receiver. But it is implemented using the standard JVM protocol.

8.6.2 Compiling with clauses

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

```
class M<T extends N with {...}> extends T { ... }
```

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

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

Chapter 9

Core MixGen

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

9.1 Definitions

The design of CORE MIXGEN is based on the Featherweight GJ core language for GJ [34]. In the remainder of this thesis, we will refer to these two languages as CMG and FGJ respectively. In developing CMG, we augmented FGJ with the essential features required to support first-class genericity, but nothing more. These features included the following:

- **with** clauses in type parameter declarations. Since FGJ and CMG do not include abstract classes or interfaces, **with** clauses contain only constructor signatures. A **with** clause consists of a sequence of constructor signatures terminated by semicolons and enclosed in braces. For example, **with** {`init(); init(Object x);`} specifies that a type variable contains two constructors: one zeroary constructor and one constructor that takes a single argument of type `Object`.

- Relaxed restrictions on the use of naked type variables. In CMG (as in MIXGEN), all generic types including type variables are first-class and can appear in casts, `new` expressions, and `extends` clauses of class definitions.
- Multiple constructors in a class definition. In FGJ, each class has a standard constructor that takes an initial value for each field as an argument. In CMG, we relax this restriction and permit multiple constructors with arbitrary signatures. This feature allows a class to implement multiple `with` clauses. Without this feature, all classes matching a given `with` clause would have to contain exactly the same collection of fields, crippling the language’s expressiveness.

All CMG programs are valid MIXGEN programs.¹ In addition, all FGJ programs *that do not include covariant subtyping of method return types*² are valid CMG programs, modulo two trivial modifications: (1) all type parameter declarations must be annotated with empty `with` clauses, and (2) the arguments in a constructor call must include casts so that they match the parameter types exactly. The former modification is required for the sake of syntactic simplicity; all CMG type parameter declarations must contain `with` clauses. The latter modification is required because CORE MIXGEN allows multiple constructors. In order to keep the resolution of constructor calls simple, an exact match of the static types of constructor arguments to a constructor signature is required. Like FGJ, CMG is a functional language. The body of each method consists of a single `return` statement.

9.1.1 Syntax

The syntax of CORE MIXGEN is given in Figure 9.1. Throughout all formal rules of the language, the following meta-variables are used over the following domains:

¹Some invalid casts that cause errors at run-time in CMG would be detected statically in MIXGEN. Because it is rarely possible in the context of first-class genericity to detect casts that are guaranteed to fail, CMG simply allows all casts to pass static checking. For this reason, the complications arising from “stupid casts” in Featherweight Java do not arise in CMG. The relationship between CMG and MIXGEN parallels the relationship between FGJ and GJ.

²Like MIXGEN, CMG does not support covariant subtyping of return types, which interferes with first-class genericity. See Chapter 8.

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

Table 9.1 : Core MixGen Syntax

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

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

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

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

$\Delta \vdash T <: T \text{ [S-REFLEX]}$ $\frac{\Delta \vdash S <: T \quad \Delta \vdash T <: U}{\Delta \vdash S <: U} \text{ [S-TRANS]}$ $\Delta \vdash X <: \Delta(X) \text{ [S-BOUND]}$ $\frac{CT(C) = \text{class } C < \bar{X} \text{ extends } \bar{N} \text{ with } \overline{\{T\}} > \text{ extends } T \{ \dots \}}{\Delta \vdash C < \bar{S} > <: [\bar{X} \mapsto \bar{S}]T} \text{ [S-CLASS]}$ <hr style="border: 0.5px solid black;"/> $\text{bound}_{\Delta}(X) = \Delta(X) \quad \text{bound}_{\Delta}(N) = N$

Table 9.2 : Subtyping and Type Bounds

9.1.2 Subtyping and Valid Class Tables

Rules for subtyping appear in Figure 9.2. The subtyping relation is represented with the symbol $<: .$ Subtyping is reflexive and transitive, and mixin instantiations are subtypes of the instantiations of their parent types.

A class table CT is a mapping from class names to definitions. A program is a fixed class table with a single expression e . Executing a program consists of evaluating e . As in FGJ, a valid class table must satisfy several constraints: (1) for every C in $\text{dom}(CT)$, $CT(C) = \text{class } C \dots$, (2) $\text{Object} \notin \text{dom}(CT)$, (3) every class name appearing in CT is in $\text{dom}(CT)$, (4) the subtype relation induced by CT is antisymmetric, and (5) the sequence of ancestors of every instantiation type is finite. These last two properties, which are trivial to check in FGJ and Java, are actually quite subtle in CMG, as they are in MIXGEN. CMG avoids both of these complications (cycles and infinite class hierarchies) by placing the following two constraints on class tables:

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

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

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

9.1.3 Type Checking

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

Table 9.3 : Well-formed Constructs

The typing rules of CMG include three environments:

- A type environment Γ mapping program variables to their static types. Syntactically, these mappings have the form $\bar{x} : \bar{T}$.
- A bounds environment Δ mapping type variables to their upper bounds. Syntactically, these mappings have the form $\bar{X} \triangleleft \bar{N}$. The bound of a type variable is always a non-variable type. The bound of a non-variable type N is N .
- A `with` environment Φ mapping type variables to the lower bounds on the set of constructors that they provide. This information is given in `with` clauses and complements the information in the bounds environment. Since CMG does not include abstract or final methods, constraints on the set of allowed abstract and final methods do not appear in `with` clauses. Syntactically, `with` environments have the form $\bar{X} \bowtie \overline{\{\bar{T}\}}$, where $\overline{\{\bar{T}\}}$ denotes the set of constructor signatures specified in a `with` clause.

When multiple environments are relevant to a typing judgment, they appear together, separated by semicolons. Empty environments are denoted with the symbol \emptyset . In the interest of brevity, we often omit empty environments from typing judgments. For example, the judgment $\emptyset; \emptyset; \emptyset \vdash e \in \text{Object}$ is abbreviated as $\vdash e \in \text{Object}$. The extension of an environment E with environment E' is written as $E + E'$. We use the notation $[\bar{X} \mapsto \bar{Y}]e$ to signify the safe substitution of all free occurrences of \bar{X} for \bar{Y} in e .

9.1.4 Well-formed Types and Class Definitions

The rules for well-formed constructs appear in Figure 9.3. A type instantiation is well-formed in environments $\Phi; \Delta$ if all instantiations of type parameters (1) are subtypes of their formal types in Δ , and (2) contain all constructors specified in Φ .³ Method

³If a type variable is instantiated with another type variable, Φ is checked to ensure that the sets of specified constructor signatures are compatible.

definitions are checked for well-formedness in the context of the class definition in which they appear. A method `m` appearing in class `C` is well-formed in the body of `C` if the constituent types are well-formed, the type of the body in Δ is a subtype of the declared type, and `m` is a valid override of any method of the same name in the static type of the parent of `C`.

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

Class definitions are well-formed if the constituent elements are well-formed, none of the fields known statically to occur in ancestors are shadowed,⁴ and every constructor has a distinct signature.

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

9.1.5 Constructors and Methods

The rules for method and constructor inclusion, method typing, method lookup, and valid overrides, appear in Figure 9.4. Method types are determined by searching upward from the static type for the first match. The type of the class in which a method occurs is prepended to method types. These prepended classes are used to annotate receiver expressions in the typing rule for method invocations. As explained in section 9.1.10, the annotated type of a receiver of an application of method `m` is reduced to a more specific type when the more specific type includes `m` (with a compatible method signature) in the static type of its parent. Once the annotated type of a receiver is reduced to the most specific type possible, lookup of `m` starts at

⁴Notice that this constraint alone does not prevent accidental shadowing in mixin instantiations.

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

Table 9.4 : Constructors and Methods

the reduced annotated type.

9.1.6 Expression Typing

$\Phi; \Delta; \Gamma \vdash x \in \Gamma(x) \text{ [GT-VAR]} \quad \frac{\Phi; \Delta \vdash T \text{ ok} \quad \Phi; \Delta; \Gamma \vdash e \in S}{\Phi; \Delta; \Gamma \vdash (T)e \in T} \text{ [GT-CAST]}$
$\frac{\Phi \vdash T \text{ includes init}(\bar{S}) \quad \Phi; \Delta; \Gamma \vdash \bar{e} \in \bar{S} \quad \Phi; \Delta \vdash T \text{ ok}}{\Phi; \Delta; \Gamma \vdash \text{new } T(\bar{e}) \in T \text{ annotate } [\bar{e} :: \bar{S}]} \text{ [GT-NEW]}$
$\frac{\begin{array}{l} \text{fields}(N) = \bar{T} \bar{f} \\ \Phi; \Delta; \Gamma \vdash e \in T \quad \Delta \vdash T <: N \\ \Delta \vdash P <: N \text{ and } f_i \in \text{fields}(P) \text{ implies } P = N \end{array}}{\Phi; \Delta; \Gamma \vdash e.f_i \in T_i \text{ annotate } [e :: N]} \text{ [GT-FIELD]}$
$\frac{\begin{array}{l} \Phi; \Delta \vdash \bar{T} \text{ ok} \quad \Phi; \Delta; \Gamma \vdash e_0 \in T_0 \quad \Phi; \Delta; \Gamma \vdash \bar{e} \in \bar{R} \\ \text{mtype}(m, \text{bound}_\Delta(T_0)) = P. <\bar{X} \text{ extends } \bar{N} \text{ with } \{\bar{T}\}> S m(\bar{U} \bar{x}) \\ \Delta \vdash \bar{T} <: [\bar{X} \mapsto \bar{T}]\bar{N} \quad \Phi \vdash \bar{T} \text{ includes } [\bar{X} \mapsto \bar{T}]\{\bar{T}\} \quad \Delta \vdash \bar{R} <: [\bar{X} \mapsto \bar{T}]\bar{U} \end{array}}{\Phi; \Delta; \Gamma \vdash e_0.m<\bar{T}>(\bar{e}) \in [\bar{X} \mapsto \bar{T}]S \text{ annotate } [e_0 \in P]} \text{ [GT-INVK]}$
<hr style="border-top: 3px double #000;"/> $\frac{\begin{array}{l} \Delta \vdash \bar{R} <: \bar{S} \quad \Phi; \Delta \vdash T \text{ ok} \quad \Phi; \Delta \vdash \bar{S} \text{ ok} \\ \Phi \vdash T \text{ includes init}(\bar{S}) \quad \Phi; \Delta; \Gamma \vdash \bar{e} \in \bar{R} \end{array}}{\Phi; \Delta; \Gamma \vdash \text{new } T(\bar{e} :: \bar{S}) \in T} \text{ [GT-ANN-NEW]}$
$\frac{\begin{array}{l} \text{fields}(N) = \bar{T} \bar{f} \quad \Phi; \Delta \vdash N \text{ ok} \\ \Phi; \Delta; \Gamma \vdash e \in T \quad \Delta \vdash T <: N \end{array}}{\Phi; \Delta; \Gamma \vdash [e :: N].f_i \in T_i} \text{ [GT-ANN-FIELD]}$
$\frac{\begin{array}{l} \Phi; \Delta \vdash \bar{T} \text{ ok} \quad \Phi; \Delta; \Gamma \vdash e_0 \in T_0 \quad \Phi; \Delta; \Gamma \vdash \bar{e} \in \bar{R} \\ \text{mtype}(m, 0) = P. <\bar{X} \text{ extends } \bar{N} \text{ with } \{\bar{T}\}> S m(\bar{U} \bar{x}) \\ \Delta \vdash \bar{T} <: [\bar{X} \mapsto \bar{T}]\bar{N} \quad \Phi \vdash \bar{T} \text{ includes } [\bar{X} \mapsto \bar{T}]\{\bar{T}\} \quad \Delta \vdash \bar{R} <: [\bar{X} \mapsto \bar{T}]\bar{U} \end{array}}{\Phi; \Delta; \Gamma \vdash [e_0 \circ 0].m<\bar{T}>(\bar{e}) \in [\bar{X} \mapsto \bar{T}]S} \text{ [GT-ANN-INVK]}$

Table 9.5 : Expression Typing

The rules for expression typing are given in Figure 9.5. Naked type variables may occur in `new` expressions and casts. When checking `new` expressions of naked type, the `with` environment is checked to ensure that it includes an appropriate constructor signature.

The expression typing rules annotate the receiver expressions for method invocations and field lookups with a static type. In the case of a field lookup, this static type is used to disambiguate the field reference in the presence of accidental shadowing. Although classes are statically prevented from shadowing the known fields of their ancestors, a mixin instantiation may accidentally shadow a field contained in its parent.⁵ 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 Chapter 9. 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 known to exist. During computation, the annotated type is reduced whenever possible, modeling the downward search semantics of hygienic mixin method overriding.

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

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

⁵One pathological case of such accidental shadowing occurs when a mixin instantiation extends another instantiation of itself. Then all fields in the parent class are shadowed with fields of incompatible type.

9.1.7 Explicit Polymorphism

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

9.1.8 Fields and Field Values

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

Table 9.6 : Fields and Field Values

The rules for the retrieval of the field names and values of an object (used by the typing and computation rules on field lookup) are given in Figure 9.6. The presence of multiple constructors for a class, where constructor signatures do not directly match the field types of a class, makes field value lookup more complex than in FGJ. It is important that a `new` expression is matched to the constructor of the appropriate signature. As a result, unlike FGJ, the mapping *fields* only retrieves those fields directly defined in a class definition. Additionally, a mapping *field-vals* is needed to find the field values of a given object. A static type is passed to *field-vals* to allow for field disambiguation in the presence of accidental shadowing.

9.1.9 Constructor Call Resolution

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

9.1.10 Computation

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

Table 9.7 : Computation

The CORE MIXGEN computation rules are defined in Figure 9.7. As in FGJ, computation is specified via a small-step semantics. Because the static type of a receiver is used to resolve method applications and field lookups, static types must be preserved during computation as annotations on receiver expressions. When computing the application of a method, the appropriate method body is found according to the mapping *mbody*. The application is then reduced to the body of the method, substituting all parameters with their instantiations, and `this` with the receiver. Because it is important that a method application is not reduced until the most specific matching type annotation of the receiver is found, two separate forms are used for type annotations. The original type annotation marks the receiver with an annotation of the form $\in T$. This form of annotation is kept until no further reduction of the static type is possible. At that point, the form of the annotation is switched to $:: T$. Because the computation rules dictate that methods can be applied only on receivers whose annotations are of the latter form, we're ensured that no further reduction is possible when a method is applied. The symbol \circ is used to designate contexts where either form of annotation is applicable.

9.2 Proof of Type Soundness

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

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

ments in which the trailing expression of a program is typed are empty. We refer to expressions that are well-typed in the set of empty environments as *ground*. Because the evaluation of a program consists of reducing a expression, and because ground expressions always reduce to other ground expressions, it suffices to prove subject reduction solely for ground expressions. We formalize the notion of groundedness with the following definitions.

9.2.1 Ground Expressions

Definition 1 (Ground Types) *A type T is **ground** iff $\vdash \mathsf{T}$ ok.*

Definition 2 (Ground Expressions) *An expression e is **ground** iff $\vdash e \in \mathsf{T}$.*

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

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

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

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

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

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

Case $\overline{\mathsf{X}} \bowtie \overline{\{\mathsf{T}\}} \vdash \mathsf{x}_i$ includes I_k : We're given that $\vdash \overline{\mathsf{T}}$ includes $[\overline{\mathsf{X}} \mapsto \overline{\mathsf{T}}]\overline{\{\mathsf{T}\}}$. But $[\overline{\mathsf{X}} \mapsto \overline{\mathsf{T}}]\mathsf{x}_i = \mathsf{T}_i$, finishing the case.

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

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

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

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

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

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

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

implies

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

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

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

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

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

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

Case S-Reflex: Trivial.

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

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

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

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

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

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

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

Case WF-Object: Trivial.

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

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

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

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

Case S-Reflex: Trivial.

Case S-Trans: Immediate from the induction hypothesis.

Case S-Bound: Impossible since \mathbb{N} is ground.

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

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

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

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

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

Case GT-Ann-Field: $\vdash e = [e_0 :: \mathbb{N}].f_i \in \mathbb{T}_i$. By Lemma 1, \mathbb{N} is ground. Then by Lemma 3, \mathbb{T}_i is ground.

Case GT-Ann-Invk: Let $e = [e_0 :: \mathbb{T}_0].m\langle\bar{\mathbf{R}}\rangle(\bar{\mathbf{e}})$, $mtype(m, \mathbb{T}_0) = \mathcal{C}\langle\bar{\mathbf{U}}\rangle.\langle\bar{\mathbf{Y}} \text{ extends } \bar{\mathbf{N}} \text{ with } \overline{\{\bar{\mathbf{I}}\}} \rangle \mathbb{T}' m(\bar{\mathbf{S}} \bar{\mathbf{x}})$. Let $CT(\mathcal{C}) = \text{class } \mathcal{C}\langle\bar{\mathbf{X}}\rangle \text{ extends } \bar{\mathbf{N}}' \text{ with } \overline{\{\bar{\mathbf{I}}'\}} \rangle \text{ extends } \mathbb{V} \{ \dots \}$. Then $\mathbb{T}' = [\bar{\mathbf{X}} \mapsto \bar{\mathbf{U}}]\mathbb{T}''$ where \mathbb{T}'' is the return type of m in the original method definition in \mathcal{C} . In that case, the type \mathbb{T} of expression e is $[\bar{\mathbf{Y}} \mapsto \bar{\mathbf{R}}]\mathbb{T}'$ by [GT-ANN-INVK], so we must show that $\vdash [\bar{\mathbf{Y}} \mapsto \bar{\mathbf{R}}]\mathbb{T}'$ ok. But, by [GT-METHOD], $\bar{\mathbf{X}} \bowtie \overline{\{\bar{\mathbf{I}}'\}} + \bar{\mathbf{Y}} \bowtie \overline{\{\bar{\mathbf{I}}\}}; \bar{\mathbf{X}} \triangleleft \bar{\mathbf{N}}' + \bar{\mathbf{Y}} \triangleleft \bar{\mathbf{N}} \vdash \mathbb{T}'$ ok. Then by Lemma 6, we have $\vdash [\bar{\mathbf{X}} \mapsto \bar{\mathbf{U}}][\bar{\mathbf{Y}} \mapsto \bar{\mathbf{R}}]\mathbb{T}'$ ok, \square

9.2.2 Class Hierarchies

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

Lemma 9 (Compactness) *For a given ground type $C\langle\bar{N}\rangle$, there is a finite chain of ground types P_0, \dots, P_N s.t. for all i s.t. $1 \leq i \leq N$, $\vdash P_{i-1} <: P_i$ and $P_N = \text{Object}$.*

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

Lemma 10 (Antisymmetry) *For ground types $C\langle\bar{N}\rangle$, $D\langle\bar{P}\rangle$, if $\vdash C\langle\bar{N}\rangle <: D\langle\bar{P}\rangle$ then either $\not\vdash D\langle\bar{P}\rangle <: C\langle\bar{N}\rangle$ or $C\langle\bar{N}\rangle = D\langle\bar{P}\rangle$.*

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

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

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

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

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

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

Lemma 11 (Uniqueness) *For a given ground type $C\langle\bar{N}\rangle$, there is exactly one type $P \neq C\langle\bar{N}\rangle$ (i.e., the declared parent instantiation) s.t. both of the following conditions hold:*

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

Proof Let P be the declared parent instantiation of $C\langle\bar{N}\rangle$. Suppose for a contradiction that there exists a type $0 \neq P$ s.t. $\vdash C\langle\bar{N}\rangle <: 0$, $C\langle\bar{N}\rangle \neq 0$, and $\vdash 0 <: P$. Then there is some finite derivation of $\vdash C\langle\bar{N}\rangle <: 0$. Consider a shortest derivation of $\vdash C\langle\bar{N}\rangle <: 0$, i.e., a derivation employing no more rule applications than any other derivation. Such a derivation can't conclude with [S-REFLEX] because $C\langle\bar{N}\rangle \neq 0$. Also, it can't

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

9.2.3 Preservation

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

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

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

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

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

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

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

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

There are two cases:

Subcase 0 = P: Then by Lemma 4,

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

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

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

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

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

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

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

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

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

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

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

Case GR-Cast: Immediate from Lemma 1.

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

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

Case GR-Invk: $[\text{new } C \langle \bar{S} \rangle (\bar{e} :: P) :: D \langle \bar{T} \rangle].m \langle \bar{V} \rangle (\bar{d}) \rightarrow [\bar{x} \mapsto \bar{d}][\text{this} \mapsto \text{new } C \langle \bar{S} \rangle (\bar{e} :: \bar{P})]e_0$. Then $\text{mbody}(m \langle \bar{V} \rangle, D \langle \bar{T} \rangle) = (x, e_0)$. Let $CT(D) =$

class $D\langle\bar{X}\rangle$ extends \bar{N} with $\{\bar{T}\}$ extends $U \quad \{\dots\}$ and $mtype(m, D\langle\bar{T}\rangle) = \langle\bar{Y}\rangle$ extends \bar{N}' with $\{\bar{T}'\}$ extends $U' \quad m(\bar{R} \ \bar{x})$. Then $e_0 = [\bar{Y} \mapsto \bar{V}][\bar{X} \mapsto \bar{T}]e'_0$ where e'_0 is the body of m appearing in the definition of D . By [GT-METHOD], $\bar{X} \bowtie \{\bar{T}\} + \bar{Y} \bowtie \{\bar{N}'\}; \bar{X} \triangleleft \bar{N} + \bar{Y} \triangleleft \bar{N}'; \bar{x} : \bar{R} \vdash e'_0 \in U''$ for some U'' s.t. $\bar{X} \triangleleft \bar{N} + \bar{Y} \triangleleft \bar{N}' \vdash U'' \triangleleft : U'$. The only potential free variables in e'_0 are \bar{X} , \bar{Y} , and \bar{x} . But, by Lemma 1, \bar{T} , \bar{D} , and \bar{d} are ground, so by Lemmas 12 and 13, $[\bar{x} \mapsto \bar{d}][\text{this} \mapsto \text{new } C\langle\bar{S}\rangle(\bar{e} :: \bar{P})]e_0$ is ground.

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

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

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

Theorem 1 (Preservation of Types Under Subject Reduction) *For ground expression e , and ground type T , if $\vdash e \in T$ and $e \rightarrow e'$ then $\vdash e' \in S$ where $\vdash S \triangleleft : T$.*

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

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

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

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

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

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

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

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

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

By [GT-ANN-INVK], $\vdash e \in [\bar{\mathbf{Y}} \mapsto \bar{\mathbf{V}}][\bar{\mathbf{X}} \mapsto \bar{\mathbf{R}}]\mathbf{S}$. Let $\Phi = \bar{\mathbf{X}} \bowtie \overline{\{\bar{\mathbf{I}}\}} + \bar{\mathbf{Y}} \bowtie \overline{\{\bar{\mathbf{I}}\}}$, $\Delta = \bar{\mathbf{X}} \triangleleft \bar{\mathbf{T}}' + \bar{\mathbf{Y}} \triangleleft \bar{\mathbf{N}}$, and $\Gamma = \bar{\mathbf{x}} : \bar{\mathbf{U}} + \text{this} : \mathbf{D} \langle \bar{\mathbf{R}} \rangle$. By [GT-METHOD], $\Phi; \Delta; \Gamma \vdash e_0 \in \mathbf{S}'$ where $\Delta \vdash \mathbf{S}' <: \mathbf{S}$. By Lemma 12, $\Gamma \vdash [\bar{\mathbf{Y}} \mapsto \bar{\mathbf{V}}][\bar{\mathbf{X}} \mapsto \bar{\mathbf{R}}]e_0 \in [\bar{\mathbf{Y}} \mapsto \bar{\mathbf{V}}][\bar{\mathbf{X}} \mapsto \bar{\mathbf{R}}]\mathbf{S}'$ and by Lemma 5, $\vdash [\bar{\mathbf{Y}} \mapsto \bar{\mathbf{V}}][\bar{\mathbf{X}} \mapsto \bar{\mathbf{R}}]\mathbf{S}' <: [\bar{\mathbf{Y}} \mapsto \bar{\mathbf{V}}][\bar{\mathbf{X}} \mapsto \bar{\mathbf{R}}]\mathbf{S}$. Also, by [GT-ANN-INVK], $\vdash \bar{\mathbf{e}} \in \bar{\mathbf{U}}'$ where $\vdash \bar{\mathbf{U}}' <: [\bar{\mathbf{Y}} \mapsto \bar{\mathbf{V}}][\bar{\mathbf{X}} \mapsto \bar{\mathbf{R}}]\bar{\mathbf{U}}$. Then by Lemma 13, $\vdash [\bar{\mathbf{x}} \mapsto \bar{\mathbf{e}}][\text{this} \mapsto \text{new } \mathbf{C} \langle \bar{\mathbf{S}} \rangle (\bar{\mathbf{e}})][\bar{\mathbf{Y}} \mapsto \bar{\mathbf{V}}][\bar{\mathbf{X}} \mapsto \bar{\mathbf{R}}]e_0 \in \mathbf{S}''$, where $\vdash \mathbf{S}'' <: [\bar{\mathbf{Y}} \mapsto \bar{\mathbf{V}}][\bar{\mathbf{X}} \mapsto \bar{\mathbf{R}}]\mathbf{S}'$. Finally, by transitivity of subtyping, $\vdash \mathbf{S}'' <: [\bar{\mathbf{Y}} \mapsto \bar{\mathbf{V}}][\bar{\mathbf{X}} \mapsto \bar{\mathbf{R}}]\mathbf{S}'$, finishing the case.

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

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

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

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

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

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

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

Notice that the preservation theorem above (as well as the supporting lemmas) establish preservation for annotated terms. But since terms are not annotated

until type checking, it is important to establish that the types of the annotated terms match their types before annotation. This property is established with the following two lemmas (the first is merely a small supporting lemma for the second).

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

Proof Trivial induction on the derivation of

$$mtype(\mathbf{m}, \mathbb{N}) = \mathbb{O}.\langle \bar{\mathbf{x}} \rangle \text{ extends } \bar{\mathbf{P}} \text{ with } \{\bar{\mathbf{I}}\} \triangleright \mathbb{T} \mathbf{m}(\bar{\mathbf{S}} \bar{\mathbf{x}}). \quad \square$$

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

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

Proof We consider each part of the theorem in turn.

1. $\Phi; \Delta; \Gamma \vdash \mathbf{e}.\mathbf{f}_i \in \mathbb{T}$. The only distinction between the antecedents of [GT-FIELD] and [GT-ANN-FIELD] is that the type \mathbb{N} in which the accessed field is contained is explicitly determined in [GT-FIELD] by the annotated type of the receiver. But, by Lemma 11, the condition that there is no proper subtype \mathbb{P} of \mathbb{N} s.t. $fields(\mathbb{P})$ includes \mathbf{f}_i ensures that \mathbb{N} is unique. Because this unique type is the annotation assigned to the receiver, it is the same type referred to in [GT-ANN-FIELD], so $\Phi; \Delta; \Gamma \vdash [\mathbf{e} :: \mathbb{N}].\mathbf{f}_i \in \mathbb{T}$.
2. $\Phi; \Delta; \Gamma \vdash \mathbf{new} \mathbf{R}(\bar{\mathbf{e}}) \in \mathbb{T}$. Because none of the argument expressions have been reduced, their static types will match the annotated types exactly and $\vdash \bar{\mathbf{e}} \in \bar{\mathbf{N}}$. The other antecedents of [GT-NEW] match antecedents of [GT-ANN-NEW] exactly.

3. $\Phi; \Delta; \Gamma \vdash \mathbf{e}.\mathbf{m}\langle\bar{\mathbf{v}}\rangle(\bar{\mathbf{e}}) \in \mathbf{T}$. Again, no reduction has occurred, so by Lemma 15 the annotated type $\mathbf{0}$ will match the closest supertype of the bound of the static type \mathbf{T}_0 of the receiver that contains \mathbf{m} . Then $mtype(\mathbf{m}, \mathbf{0}) = mtype(\mathbf{m}, bound_{\Delta}(\mathbf{T}_0))$ and the case is finished by [GT-ANN-INVK].

□

9.2.4 Progress

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

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

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

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

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

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

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

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

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

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

Proof Trivial induction over the derivation of

$$mtype(m, N) = R.\langle \bar{X} \text{ extends } \bar{P} \text{ with } \{\bar{T}\} \rangle T m(\bar{U} \bar{x}). \quad \square$$

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

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

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

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

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

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

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

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

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

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

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

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

□

9.2.5 Type Soundness

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

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

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

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

Chapter 10

Conclusion

Adding support for generic types is critical for precise typing in object-oriented programming languages. The research community has put forth many thoughtful proposals on how best to add these types. However, these proposals have largely overlooked the essential role that first-class genericity can play in effective software development. We have presented numerous examples of how a lack of support for first-class genericity inhibits the application of many widely practiced design patterns and practices. Without first-class genericity, programs in a generic type system will be more difficult to develop, maintain, and debug.

As the NEXTGEN compiler demonstrates, we can achieve essentially the same level of compatibility with existing legacy code as GJ while adding support for type dependent operations. Furthermore, we have shown that our implementation of this design incurs no serious performance penalties compared to the existing Java and GJ compilers. Finally, we have provided a sound, compatible extension of Java, and similar nominally subtyped object-oriented languages, that supports true first-class genericity.

Adding first-class genericity to a nominally subtyped object-oriented language produces a surprisingly powerful language that supports precisely typed mixins as well as conventional generic classes. We have shown how the resulting language can be safely type checked and how it can be efficiently implemented on existing run-time systems such as the JVM. We believe that these results provide a convincing argument for adding first-class genericity to the next generation of nominally subtyped object-oriented languages. We conclude with some directions for future research.

10.1 Future Research

Our analysis of first-class genericity suggests several directions for further inquiry. We briefly mention three such directions.

10.1.1 Performance of Hygienic Method Lookup

Adding support for full first-class genericity to NEXTGEN introduces more forwarding methods and interface-based method dispatches into the implementation, which could conceivably add some overhead to programs that use generic types. We do not believe that this overhead is likely to be significant in practice—even in programs that make heavy use of generics—because the extra forwarding methods are typically inlined by JIT compilers. Nevertheless, this expectation can be checked empirically by implementing the scheme described in Chapter 8 and testing it against a suite of representative benchmarks. Because our scheme inserts forwarding methods into all classes (not just mixins), its performance could be measured against existing Generic Java compilers by leaving mixins out of the benchmark suite entirely. In this way, each Generic Java compiler could be tested on code identical to the others.

10.1.2 Selective Covariance of Generic Types

A simple but potentially useful extension of MIXGEN, would be to allow type parameters to be declared as covariant so that $C<S>$ is a subtype of $C<T>$ if S is a subtype of T . Extending the MIXGEN compiler design to support this feature is straightforward. It involves (i) trivially modifying the parser to support syntax for covariant type variable declarations, and (ii) extending the customized class loader to support covariant instantiation classes by adding all of the interfaces corresponding to the supertypes of the instantiation to the list of implemented interfaces. However, we expect that we would have to impose severe restrictions on the use of covariant type parameters to maintain type soundness. In particular, we would have to ensure that no legal mixin instantiation would produce incompatible overriding method signatures. We expect

that problems similar to those discussed in Chapter 8 for covariant method return types may arise.

10.1.3 Integration of Raw Types

As mentioned in Chapter 1, raw types play an important role in integrating generic code with non-generic legacy code. However, we have not included raw types in our formal analysis of first-class genericity. We expect that many subtle issues arise when the two features are combined. Igarashi and Viroli have formally analyzed raw types (a.k.a. variance types) and have determined a set of contexts where they can be used safely [36]. We would like to integrate their analysis with our analysis of first-class genericity and produce a sound language that supports both.

Appendix A

The NextGen Implementation Code Structure

This appendix describes the high-level architecture and code layout of the NEXTGEN prototype compiler. The NEXTGEN compiler was developed as an extension to the GJ compiler under special license from Sun Microsystems. This same compiler was extended independently by Sun Microsystems to form the JSR-14 prototype compiler, scheduled for inclusion in J2SE 1.5. In the process of developing NEXTGEN, we have refactored the original GJ compiler substantially, and no attempt has been made to maintain compatibility with the JSR-14 source code. In addition, we have retrofitted our compiler with a substantial array of unit tests, boosting the sheer sized of the code base from 35,000 lines to 50,000 lines. Nevertheless, the reader may find that some of the architectural features of NEXTGEN described here are helpful when deciphering the JSR-14 code base (modulo class, package, and variable name changes).

The GJ compiler, as well as the NEXTGEN and JSR-14 compilers derived from it, are written in Generic Java.¹ As a result, we enjoy much more precise type checking in the source code of these compilers than we would have with ordinary Java. But we are also unable to use many of the powerful development tools available for standard Java.²

Throughout this appendix, it is assumed that the reader has access to a working JavaPLT development environment, is familiar with basic CVS commands, with JUnit, with Ant, and with DrJava. Instructions on setting up a JavaPLT environment

¹”Generic Java” is a specification for a family of languages that add generic types to Java. This family includes both GJ and NEXTGEN.

²Two very useful tools that are compatible with Generic Java are the CODEGUIDE and DRJAVA IDEs. All NEXTGEN developers are encouraged to leverage both tools. DRJAVA is particularly useful for writing new unit tests; CODEGUIDE provides support for automated refactoring and incremental compilation.

are available at

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

A short tutorial on CVS is available at <http://www.cvshome.org/docs/manual>. A tutorial on JUnit is available at <http://www.junit.org>. A tutorial on Ant is available at <http://www.jakarta.org>. DrJava documentation is available at <http://drjava.sf.net>.

A.1 The NEXTGEN CVS Repository

The NEXTGEN source code is maintained under the `javaplt` CVS repository. To checkout the NextGen source code, go to your `javaplt` directory and checkout the module `nextgen`. Because NEXTGEN is written in JSR-14, you will need to prepend JSR-14 to the bootclasspath when starting Ant. The best way to do that is to set your `ANT_OPTS` environment variable to the location of the JSR-14 jar. For example, on Unix systems, set it to:

```
-Xbootclasspath/p:'javaplt-home'/packages/jsr14_adding_generics-1_3-ea/javac.jar
```

Of course, on Windows/Cygwin, your path will look different.³

Once you have checked out a copy of the code, the first thing you should do is `cd` to the new `nextgen` directory and type

```
$ ant all
```

Doing so will run all Ant targets in the project. An *essential* invariant of the NEXTGEN project is that `ant all` should succeed when invoked on a clean checkout, on all supported platforms (Linux, Solaris, OS X, and Windows XP/Cygwin) with a proper JavaPLT development environment. If it doesn't, be sure to correct any problems with your environment before continuing.

³Because we must set this variable before Ant starts, this is one place where Ant's ability to automatically convert path names doesn't save us.

A.2 The NEXTGEN Project Directory Structure

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

- `build.xml`. This file contains the XML source code for all Ant targets associated with the NextGen project.
- `anttools`. This directory contains the Java source code for custom Ant tasks associated with NEXTGEN. In particular, the `NextGenCompilerTask` and `NextGenJUnitRunner` source code is contained here. Because all Java source code in the NEXTGEN project is placed in various subpackages of `edu.rice.cs.nextgen`, the files in this directory (and all other directories containing Java source code) are all placed in subdirectories of `edu/rice/cs/nextgen`.

`NextGenCompilerTask` is used in the `ant compile` target in `build.xml`. Although the NEXTGEN compiler could be invoked within Ant simply as a Java program, the `NextGenCompilerTask` is much more convenient because it allows the compiler to be invoked on Ant filesets and other compound structures instead of just an explicit list of command-line arguments.

`NextGenJUnitRunner` is used for NEXTGEN acceptance testing. It is called by the `simpletests` target. This task uses the NEXTGEN classloader to invoke JUnit on `TestCases` compiled with the `NextGenCompiler`. Note that such `TestCases` can't be invoked by JUnit with the default class loader because some of them may be template classfiles.

- `benchmarks`. Contains the source code for the NEXTGEN benchmark suite. There are three versions of this suite: `edu.rice.cs.nextgen.javabenchmarks`, `edu.rice.cs.nextgen.gjbenchmarks`, `edu.rice.cs.nextgen.nextgenbenchmarks`, corresponding to the three compilers tested in NEXTGEN benchmarking. All of these suites are compiled by the Ant

target `compile-benchmarks`. A bash script is provided for running the benchmarks in the `bin` subdirectory, discussed below. Note that the benchmarks can't be run (easily) from within Ant because an entire run of the benchmarks involves invoking several different JVMs with different classloaders. Also, to prevent skewing performance results, we don't want to use up resources by running an Ant process during benchmarking.

- `bin`. Contains various bash scripts useful for performing tasks associated with the NEXTGEN project. Ultimately, we'd like to turn all of these scripts into platform-independent Ant targets, but in some cases, we haven't yet found a good way to do that.
- `doc`. Contains all documentation associated with the NEXTGEN project, including the sources for the NextGen Developer's Guide, which this appendix is based on. The NextGen documentation available online is a checked out copy of this directory. To make changes to the live webpage, you must have write access to `/home/javaplt`. After committing your changes, go to `/home/javaplt/public.html` and execute `cvs update`. That command will update both the `public.html` module and the copy of `nextgen/doc`. To update only the NEXTGEN documentation, simply perform a `cvs update` in the `/home/javaplt/public.html/nextgen` directory.
- `lib`. Contains all library classes needed by the Ant targets, including JSR-14, JUnit, and all `bootclasses` for Linux, Solaris, OS X, and Windows XP. Note that the bootclasses are included inside this project subdirectory to maintain the invariant that `ant all` always succeeds on a clean checkout. All bootclasses must be accessible not just to Ant, but also to the NEXTGEN compiler, to prevent it from signaling errors when compiling sources with references to standard library classes. There is no standardization of the placement of bootclasses

across JDKs⁴ and we want to decouple any quirks of the bootclasspath on a platform from the ability of the Ant targets to work. By putting all bootclasses in a well-defined location, we can ensure that we can always find them.

- **manifests.** Contains the manifest files used when bundling the NEXTGEN compiler and classloader into jar files.
- **jars.** Contains the jar files for the compiler and classloader, constructed with target `ant jar`.
- **simpletests.** Contains all acceptance tests for the NEXTGEN compiler, in the form of JUnit `TestCases` written in NEXTGEN. Running an acceptance test consists of compiling it under the NEXTGEN compiler and then invoking JUnit on it with the NEXTGEN classloader. By running JUnit on the compiled code, we check that the semantics of that code is as expected, providing a powerful check on NEXTGEN's correctness. The acceptance tests can be run by invoking the Ant target `simpletests`.
- **src.** Contains all source code for the NEXTGEN compiler and classloader, in packages `edu.rice.cs.nextgen.compiler` and `edu.rice.cs.nextgen.classloader`, respectively. In the section on NEXTGEN package design, we will discuss the various subpackages in this directory.
- **built.** This directory is empty on a clean checkout. It's used by Ant to store the class files for all Generic Java sources associated with NEXTGEN.

A.3 Ant Targets in the NEXTGEN Project

For a complete list of targets, always refer to the `build.xml` file. Below are some of the most common targets you will invoke.

⁴Non-Sun JDKs such as the Mac OS X JDK do not always follow the file layout of the Sun JDKs, and Sun hasn't made JDK file layout part of any standard specification.

- `clean`. Deletes all compiled classfiles.
- `compile`. Compiles the compiler and classloader.
- `test`. Runs all unit tests on the compiler and classloader.
- `simpletests`. Compiles the compiler and classloader and runs all acceptance tests.
- `testall`. Compiles the compiler and classloader and runs all unit tests and acceptance tests.
- `compile-tests`. Compiles all acceptance tests with the most recently compiled version of NEXTGEN.
- `update`. Updates this checked out copy of NEXTGEN with the CVS repository.
- `commit`. Synchronizes with the CVS repository, compiles, ensures that all unit tests and acceptance tests still pass, and, if so, creates new jar files and commits the newly synchronized version to the repository. Run this often! At least once a day, if not once an hour when actively working with the code. The more often you run it, the easier it will be to diagnose new bugs.
- `jar`. Constructs jar files for the most recently compiled versions of the compiler and classloader.
- `all`. Runs all targets in the project. Useful for making sure that everything still works.
- `alloffline`. Runs all targets in the project except `commit`. Useful for making sure that everything testable works when you're not connected to a network.

A.4 Releasing a New Version of NEXTGEN

The NEXTGEN documentation and jar files are available online at <http://www.cs.rice.edu/javaplt/nextgen>. To release the latest committed version to the live website, simply go to `/home/javaplt/nextgen` on csh and do a `cvs update`. Concerning the documentation: be sure that the latest committed LaTeX file corresponds to the latest committed pdf file.⁵

WARNING: If you edit anything in the live directories, be sure to perform a `cvs commit` so your changes are included in the CVS repository. Better yet, never edit the live directories directly. Instead, always make changes to your local copy and update the live directory.

A.5 NEXTGEN Package Design

A.5.1 Classloader Package Design

All classes in the NEXTGEN classloader are contained in a single package: `edu.rice.cs.nextgen.classloader`. There are no subpackages. The main entry point is class `Runner`.

A.5.2 Compiler Package Design

The source code for the NEXTGEN compiler is divided into the following packages, found in `src/edu/rice/cs/nextgen/compiler`:

- **main**. Contains the main entry point for the compiler, in class `Main`, as well as supporting code. In particular, class `JavaCompiler` contains the code for invoking the various phases of the compiler.
- **parser**. Contains the code for scanning and parsing NEXTGEN source files.
- **tree**. Contains the code defining NEXTGEN abstract syntax trees.

⁵In the long run, the pdf file should be re-generated during `ant commit`. Doing so would require finding a portable L^AT_EX to pdf generator, putting it in the NEXTGEN lib directory, and calling it in the `commit` target.

- **comp**. Contains the code for computing most phases of compilation, including symbol table entry, type checking, data flow checking, and bytecode generation.
- **code**. Contains code for many of the datastructures utilized by the various phases of compilation. Unfortunately, there is no sharp conceptual distinction between the classes contained in this package and the classes contained in the **comp** package. A useful refactoring would be to move the classes in these two packages so as to define the roles of each package more precisely.
- **flatten**. Contains the visitors and support code that convert type dependent operations into snippet calls. No comparable package exists in the JSR-14 compiler. Parametric types are flattened according to the rules for NEXTGEN name mangling. Snippet methods are added to classes as necessary. Template classes are generated. Finally, the jump targets for **break** and **continue** statements are fixed after flattening.⁶
- **instrument**. Contains the code for pretty-printing all datastructures used by the compiler. No comparable package exists in the JSR-14 compiler. We originally added this code to facilitate diagnosis of errors when modifying the datastructures during NEXTGEN-specific phases of compilation. It has been an invaluable tool in retrofitting unit tests over **NextGen**.
- **util**. Contains many general-purpose datastructures, including parametric versions of many Java collections classes. These collections classes were written as part of the GJ compiler before the JSR-14 compiler was available. We've modified and rewritten many of them, particularly the **List** classes. Although JSR-14 provides parametric versions of the collections classes, our versions (particularly of **Lists**) have many advantages over those in JSR-14, so in most cases there is little (or negative) incentive to refactor them out of the codebase. One useful

⁶These targets are broken by flattening because the positions of the targets are changed when snippet methods are added.

feature that is currently missing from our hash tables is an iterator (however, a `map` facility is provided).

- `hist`. Contains some useful DrJava interactions histories. `ImportAll.hist` imports all `NextGen` source packages. `NextGenTestCase` sets up all variables initialized in class `edu.rice.cs.nextgen.compiler.main.NextGenTestCase`. `JavaCompilerTest.hist` performs the various tasks done in the `imports` and `setUp` method of `JavaCompilerTest`, i.e., it initializes a new `JavaCompiler` on a `NextGen` sourcefile:

```
> package edu.rice.cs.nextgen.compiler.main;
> import edu.rice.cs.nextgen.compiler.code.*;
> import edu.rice.cs.nextgen.compiler.comp.*;
> import edu.rice.cs.nextgen.compiler.instrument.*;
> import edu.rice.cs.nextgen.compiler.util.*;
> import edu.rice.cs.nextgen.compiler.tree.*;
> import java.io.IOException;
> import junit.framework.TestCase;
> import junit.framework.TestSuite;
>
> String[] args = new String[] {
    "-classpath",System.getProperty("sun.boot.class.path"),
    "simpletests/edu/rice/cs/nextgen/simpletests/InstanceofParameter.java"
};
> CompilerOptions options = new CompilerOptions();
> _fileNames = options.processArgs(args);
> _compiler = new JavaCompiler(options);
```

Be sure to add other useful interactions histories as you build them.

A.6 Unit Tests in NEXTGEN

In both the compiler code and the classloader code, unit tests are contained in files ending in `Test`. Each such file contains a public class that extends `junit.framework.TestCase`. Although there is a significant set of working unit tests over the source code, not all code is covered directly (yet). Nevertheless, skeleton test cases are provided for all code, making it easier to add new tests. Also, the

`PrintableObject` class in package `instrument`, and the `NextGenTestCase` class in package `main` make writing new unit tests easy over any part of the code base.

A.6.1 Class `PrintableObject`

`PrintableObject` provides a mechanism for pretty-printing complex datastructures. By extending class `PrintableObject`, a class inherits all the functionality needed to pretty-print itself. All that is needed is for the extending class to override method `print()` and use the inherited methods to specify how it should be printed. A `PrintableObject` decides which fields of itself to print. It can also tell its constituent fields that are `PrintableObjects` to print themselves selectively, passing a `Filter` object to handle cyclic references.⁷ Virtually all complex datastructures in `NEXTGEN` extend `PrintableObject`. See their `print` methods for examples of how to call the `PrintableObject` functionality for pretty-printing.

In addition to method `print`, which sends a pretty-printed representation of an object to `System.out`, every `PrintableObject` has a `longString()` method that takes no arguments and returns a (multiline) `String` representation of the object. `longString()` is extremely useful when writing unit tests. Reasonable `toString` methods are also written for most classes, but they contain much less detailed information. The `toString` methods can be useful for quickly checking the identity of an object.

To familiarize yourself with the various datastructures used in `NEXTGEN`, it is recommended that you experiment with printing out instances of them in the DrJava interactions window while reading through their descriptions in this document. Another great way to learn how these datastructures work is by writing new unit tests over them.

⁷See classes `edu.rice.cs.nextgen.compiler.code.Scope` and `edu.rice.cs.nextgen.compiler.code.Symbol.ClassSymbol` for two examples of cyclic structures printed with `Filters`.

A.6.2 Class `NextGenTestCase`

`NextGenTestCase` initializes default values for the most commonly used NEXTGEN datastructures, allowing you to quickly build complex datastructures in a test case. It also defines the following convenience methods:

- `String removeWhiteSpace(String)`. This method takes a `String` and returns a new `String` with all whitespace removed. It's useful for checking that a multiline `String` representation of an `Object` is as expected (without breaking everytime that the whitespace in the representation is changed).
- `String removeNewlines(String)`. Like `removeWhiteSpace`, with similar motivation, but only `newlines` are removed.
- `String wrapInQuotes(String)`. Often in a test method, you will want to check that the `String` representation of an object is as expected. To specify what's expected in source code, you have to wrap the `String` representation inside quotes. For multiline `String` representations, this task can be extremely tedious. `wrapInQuotes` is a static method that takes a `String` and returns a new `String` with each line wrapped in quotes. This method can be conveniently called in the DrJava interactions window while constructing new test methods. In the long run, `wrapInQuotes` should become functionality provided by DrJava on selected blocks of text in the editor. Until that functionality is added, this method will save you time.
- `Parser makeParser(String)`. Takes a `String` representing source code and returns an instance of `edu.rice.cs.nextgen.parser.Parser` to read that source code.
- `Tree.TopLevel makeClassTree(String)`. Takes a `String` representing the source code of a complete Java source file, parses it and returns an instance of

`Tree.TopLevel`.⁸

- `Tree.Import` `makeImport(String)`. Takes a `String` representing an import declaration, parses it and returns an instance of `Tree.Import`.

Other `make` methods should be added to this class as they become useful in the unit tests. It is recommended that all new NEXTGEN test case classes extend `NextGenTestCase`.

A.7 Phases of The NEXTGEN Compiler

When class `Main` is invoked with a sequence of keyword arguments and files, it stores all keyword arguments in a new `CompilerOptions` object named `options`, creates a new instance of `JavaCompiler` named `compiler`, with a field reference to `options`, and calls `compiler.compile()` on a `List` of all source files to compile. The phases of compilation corresponding precisely to the sequence of method calls in method `JavaCompiler.compile()`. Those phases are as follows.

```

List<Tree> trees = parseFiles(filenamees);
List<Environment<AnalyzerContext>> envs = enterClasses(trees);
checkTypes(envs);
envs = flatten(envs);
patchSnippets();
eraseTypes(envs);
flattenGroundedSuperClasses();
List<ClassSymbol> classSyms = generateByteCode(envs);

```

We will now provide a high-level overview of the entire compilation process and then delve into the datastructures manipulated during each phase.

During parsing (Stage I), the initial `List<String>` of filenames is converted into a `List<Tree>`. Next, (Stage II), the classes and packages in this tree are entered into a `SymbolTable` object, and `Symbols` for various lexical items are filled into the `Trees`. A `List<Environment<AnalyzerContext>>` (basically the original `List<Tree>`

⁸For more information on `Trees`, see the section “Phases of the NextGen Compiler”.

with each `Tree` wrapped in an `Environment`) is returned from this phase. The `List<Environment<AnalyzerContext>>` is named `envs` and is the primary datastructure passed through subsequent phases.

After symbols are entered, `envs` is type checked (Stage III). The type of each `Tree` is recorded in a field in the `Tree`. After type-checking comes NEXTGEN-specific processing. Type names are flattened, snippets are added and template classes and interfaces are generated as static nested classes in the corresponding base class (Stage IV). Afterward, various source-position-specific targets must be patched in the base class (Stage V), since the locations of items will move after adding snippet methods.

After snippets are patched, types are erased, as in GJ (Stage VI).⁹ Then the superclasses of grounded types are flattened (Stage VII).¹⁰ Finally, bytecode is generated from `envs` (Stage VIII) and returned as a `List<ClassSymbol>`. These `ClassSymbols` are then written out to disk to form the corresponding classfiles.

Notice that not all stages take all datastructures they depend on as arguments. In particular, `patchSnippets` and `flattenGroundedSuperClasses` do not take `envs` as an argument. They operate on `envs` via field references set to it. Obviously, this situation is undesirable; an important refactoring is to finish decoupling all field references to `envs` and to always pass it as an argument.¹¹

Every major phase of compilation after parsing¹² is implemented as a walk over the parsed `Trees`. Consequently, each phase is associated with a subclass of class `Tree.Visitor`. For example, symbol entry is handled by visitor `SymbolEnterer`. Static checking is handled primarily by visitor `StaticAnalyzer`, which is assisted by visitor `TypeChecker`. Flattening of parametric type references is handled by visitor

⁹Notice that flattened types won't be erased, as they are no longer parametric.

¹⁰This part of flattening must be deferred because Stage IV depends on the unflattened representation. In addition, Stage II as currently written depends on this deferral because it tests whether or not the superclass is parametric and ground to determine whether the supercall for a constructor must be modified.

¹¹Originally, there were many other phases that kept field references to `envs`. The remaining phases are the most difficult to refactor.

¹²Snippet patching is not performed by a visitor because the targets to fix are accumulated during type flattening. Class `SnippetPatcher` simply walks over the accumulated list.

`TypeFlattener`. Type erasure is handled by visitor `TypeEraser`¹³. Code generation is handled by visitor `CodeGenerator`. We will now examine the various datastructures manipulated by these phases. We will start with `Trees` because the first internal representation of the source code constructed is a `List<Tree>`, and all other datastructures built depend on them.

A.7.1 Trees

```
Tree ::= TopLevel | Import | ClassDef | MethodDef | VarDef | Block
      | DoLoop | WhileLoop | ForLoop | Labelled | Switch | Case
      | Synchronized | Try | Catch | Conditional | Exec | Break
      | Continue | Return | Throw | Apply | NewInstance | NewArray
      | Assign | Assignop | Operation | TypeCast | TypeTest
      | Indexed | Select | Ident | Literal | TypeIdent | TypeArray
      | TypeApply | TypeParameter | Erroneous
```

Figure A.1 : Abstract Syntax Trees

`Trees` in `NEXTGEN` represent abstract syntax trees. `Trees` form a composite class hierarchy rooted at class `edu.rice.cs.nextgen.compiler.tree.Tree`. All subclasses in this composite hierarchy are static nested classes in class `Tree`. Every syntactic construct in `NEXTGEN` corresponds to a subclass of class `Tree`. For example, source files are parsed to `TopLevels` and class definitions are parsed to `ClassDefs`. The entire composite hierarchy is represented in BNF notation in Figure A.7.1.

A.7.2 Trees and PrintableObjects

Class `edu.rice.cs.nextgen.compiler.tree.Tree` is a subclass of class `edu.rice.cs.nextgen.compiler.instrument.PrintableObject`, allowing instances to be pretty-printed easily in the DrJava interactions pane. For example, suppose we want to pretty-print the `Tree` for this simple `NEXTGEN` source file:

¹³`TypeEraser` is a subtype of `TreeTranslator`, which is a subtype of `Tree.Visitor`.

```
package example;
public class C<T> {}
```

We can print it in the DrJava interactions window as follows:

```
> package edu.rice.cs.nextgen.compiler.main;

> NextGenTestCase.makeClassTree(
    "package example; \n" +
    "public class C<T> { } \n"
).longString()
```

Resulting in:

```
"{TopLevel for null
 packageId: {Ident example} packageSymbol: null
 starImportScope (except for java.lang): null
 namedImportScope: null
 =====
 {ClassDef C<{TypeParameter T extends null implements null}>
   extends null implements ()
   flags: 1
   null}}
"
```

This result is a pretty-printed representation of a `Tree.TopLevel`. The first line indicates the source file corresponding to the `TopLevel`. In this case it is `null` because this `TopLevel` was parsed directly from a `String` instead of a file. If the `TopLevel` were constructed from a file `f`, the first line would read `TopLevel for f`. The fields printed represent the package, the imports, and the list of `ClassDefs`. Notice the use of `null` values as defaults for many fields.

Underneath the horizontal bar are listed the constituent class definitions in the source file. The pretty-printed `ClassDef` follows the structure of the original source code. The `flags` field of a `ClassDef` is an `int` that stores various attributes of the class, such as the included visibility modifiers. Underneath the header of the `ClassDef` is its contained `ClassSymbol`, initialized to `null` and filled in during symbol entry.

If class `C` contained any members, the `Trees` for those members would be printed out underneath the `ClassSymbol`. For example, below is the source for acceptance test `ReturnIntegerTest`:

```
package edu.rice.cs.nextgen.simpletests;

class C<T> {
    Object m(Integer i) {
        return i;
    }
}
```

This source will parse to

```
{TopLevel for simpletests/edu/rice/cs/nextgen/simpletests/ReturnIntegerTest.java
  packageId:
  {Select {Select {Select {Select {Ident edu}.rice}.cs}.nextgen}.simpletests}
    packageSymbol: null
    starImportScope (except for java.lang): null
    namedImportScope: null
    =====
    {ClassDef C<{TypeParameter T extends null implements null}>
      extends null implements ()
      flags: 0
      null
      {MethodDef <> {Ident Object} m({VarDef {Ident Integer} i}) throws flags: 0
        null
        {Block {Return {Ident i}}}}}}}
```

The headers of method definitions include the type parameters (between angle brackets), the return type, name, parameter types, `throws` clause, and the visibility flags. Beneath the header is a `MethodSymbol` (filled in during symbol entry). Finally, beneath the `MethodSymbol` is the parsed body of the method.

The generated bytecodes of a method are stored in a `MethodSymbol`. After bytecode generation, the printed `MethodSymbol` will display these bytecodes. For example, here is the same `TopLevel` as above after all phases including bytecode generation:

```
{TopLevel for
  simpletests/edu/rice/cs/nextgen/simpletests/ReturnIntegerTest.java
  packageId:
```

```

{Select {Select {Select {Select {Ident edu}.rice}.cs}.nextgen}.simpletests}
  packageSymbol: [PackageSymbol edu.rice.cs.nextgen.simpletests]
  starImportScope (except for java.lang): null
  namedImportScope:
    [Scope: [ClassSymbol  fullname: edu.rice.cs.nextgen.simpletests.C
      flatname: edu.rice.cs.nextgen.simpletests.C flags: 1024
      members (cyclic references have been filtered):
        [Scope: [MethodSymbol m
          [Code: aload_1; -80; ]]
          [MethodSymbol <init>
            [Code: aload_0; -73; nop; aconst_null; -79; ]]]]]
=====
{ClassDef C<{TypeParameter T extends null implements null}>
  extends null implements ()
  flags: 0
  [ClassSymbol  fullname: edu.rice.cs.nextgen.simpletests.C
    flatname: edu.rice.cs.nextgen.simpletests.C flags: 1024
    members (cyclic references have been filtered):
      [Scope: [MethodSymbol m
        [Code: aload_1; -80; ]]
        [MethodSymbol <init>
          [Code: aload_0; -73; nop; aconst_null; -79; ]]]]
  {MethodDef <> null <init>() throws  flags: 0
    [MethodSymbol <init>
      [Code: aload_0; -73; nop; aconst_null; -79; ]]
    {Block {ExpressionStatement {Apply {Ident super}()}}}}
  {MethodDef <> {Ident Object}
    m({VarDef {Ident Integer} i}) throws  flags: 0
    [MethodSymbol m
      [Code: aload_1; -80; ]]
    {Block {Return {Ident i}}}}}}

```

MethodSymbols inside ClassSymbols contain bytecode, as well as the the MethodSymbols attached to MethodDefs in Trees (which are actually the same as (== to) the Symbols contained in the ClassSymbol. Incidentally, notice that a default constructor has been added to class C.

A.7.3 Environments

A `List<Environment<AnalyzerContext>>` is returned from Stage II. An `Environment` is a structure associated with a `Tree`; each `Environment` contains its associated `Tree`

as a field. The `List` returned from Stage II contains one environment for each `Tree` in the `List<Tree>` passed to it.

In addition to referencing its associated `Tree`, each `Environment` contains a reference to its enclosing `Environment`. For example, if an `Environment` E is associated with an immediate subtree T of a enclosing `Tree` T' , and `Environment` E' is associated with T' , then E contains a reference to E' . Each element of the `List<Environment<AnalyzerContext>>` resulting from the parsing phase of compilation is associated with a `TopLevel` representing a file in the original list of filenames. Contained `Environments` can be constructed from enclosing `Environments` by method `spawn(Tree)`.

`Environments` store information relevant to their associated `Tree`. They are used during two phases of compilation: static analysis and code generation. The information stored during these two phases is significantly different. In order to decouple the general-purpose functionality of `Environments` from the code for storing the information relevant to a particular usage, `Environments` are parameterized by a type parameter `A`. A field `context` of type `A` contains the information stored at each level of an `Environment`. So, during static analysis, we make use of a `List<Environment<AnalyzerContext>>` where each `Environment<AnalyzerContext>` in the `List` contains a `context` field of type `AnalyzerContext`.

A.7.4 AnalyzerContexts

`AnalyzerContexts` contain many fields relevant to static analysis, but most importantly they contain the following three fields:

1. An `expectedType` field of type `Type`. `Types` represent the results of static checking. This field is initialized to `NoType.ONLY` and filled in during static checking.
2. An `expectedKind` field of type `int`. `Kinds` are very high-level descriptions of the expected syntactic structure of a `Tree` that are filled in during static checking. The possible `Kinds` of a `Tree` are as follows: `NO_KIND`, `PACKAGE_KIND`, `TYPE_KIND`,

`VAR_KIND`, `VAL_OR_VAR_KIND`, `METHOD_KIND`, `ALL_KINDS`. Each `Kind` is represented as an `int` in interface `edu.rice.cs.nextgen.compiler.code.Kinds`.¹⁴ The `expectedKind` of an `AnalyzerContext` is initialized to `NO_KIND` and filled in by subsequent analysis.

3. A `Scope`. Conceptually, `Scopes` are lists of identifiers introduced in a lexical context. We now turn our attention to their internal structure.

A.7.5 `Scopes` and `Entries`

Each `Scope` is owned by a `Symbol` corresponding to the lexical environment the `Scope` represents. For example, a `Scope` corresponding to fields in a class would be owned by the associated `ClassSymbol`. The first identifier appearing in a `Scope` is stored in an `Entry`. Each `Entry` contains a `Symbol` and a reference to a `sibling` in the `Scope`. So, we can iterate over the `Entries` in a `Scope` by starting with the contained `Entry` and following the `sibling` links. The final `Entry` contains `null` as its `sibling`.¹⁵

Each `Entry` also refers to the `Entry` it shadows in an enclosing `Scope`. The shadowed `Entry` is contained in field `shadowed`.

New `Symbols` may be destructively added to a `Scope` with method `addSymbol`. `Entries` in a `Scope` may be accessed with method `lookup` that takes a `Name` and returns the `Entry` in that `Scope` corresponding to the given `Name`. `Names` are identifiers; they can be constructed from `Strings` with the static method `Name.fromString(String)`. For more information on `Names`, see their unit tests in package `edu.rice.cs.nextgen.util`. For convenience, `lookup` is overridden with a method that takes a `String` and converts it to a `Name`.

¹⁴Representing a dataset of atomic elements as `int` fields in an interface is a common idiom in `NEXTGEN`, inherited from the GJ compiler. This idiom is a workaround for the lack of enumeration types in Java. By “implementing” an interface corresponding to a dataset, a class can refer to all of the elements of the dataset directly. Also, a class can iterate over the elements of the dataset with a simple `int` counter in a `for` loop. One serious disadvantage of this idiom is that values of the dataset must be deciphered when they’re printed out in an error message. Also, proper data abstraction is not enforced; there is nothing to prevent us from using an element of a dataset in a nonsensical arithmetic operation.

¹⁵Obviously, `Scopes` and `Entries` are rich with opportunities for refactoring. Because they represent list-like structures, it would be better to make them either subtypes of type `List`, or containers, each with a field of type `List`.

Each `Scope` refers to its enclosing `Scope`, in field `nextScope`. Consequently, `Scopes` mirror the structure of the `Environments` they're associated with.¹⁶

Environments and Scopes as PrintableObjects

Like `Trees`, `Environments` and their constituents can be printed to examine their contents. For example, here is a call on `longString()` of an empty `Scope` (constructed from a default `ClassSymbol` available in `NextGenTestCase`):

```
> new Scope(NextGenTestCase.CLASS_SYMBOL).longString()
[Scope: ]
```

Using `addSymbol` and `lookup`, we can build and examine `Scopes` in the DrJava interactions window:

```
> s = new Scope(NextGenTestCase.CLASS_SYMBOL)
edu.rice.cs.nextgen.compiler.code.Scope@7dfcf1
> s.addSymbol(NextGenTestCase.METHOD_SYMBOL);
> s.lookup("METHOD_SYMBOL")
edu.rice.cs.nextgen.compiler.code.Scope$Entry@7abaab
> s.lookup("METHOD_SYMBOL").longString()

"[MethodSymbol METHOD_SYMBOL]
"
```

We can also build and print `Environments`¹⁷:

```
> new Environment(null, null)
edu.rice.cs.nextgen.compiler.comp.Environment@53be7c
> new Environment(null, null).longString()

"[Environment
  null
  null]
"
```

¹⁶This mirrored structure is undesirable because it can result in “Rogue Data” where one of the two structures is modified but the other is not. A potential refactoring would be to eliminate the `nextScope` field and replace it with a method that accesses the scope contained in the enclosing `Environment`.

¹⁷Raw types are used in this example because, at the time of this writing, the DrJava interactions window does not yet support generic types. In some unit tests where the context is unimportant, `Environments` are instantiated as `Environment<VoidContext>`. Class `VoidContext` is never used outside the unit tests.

The two arguments to the constructor are the associated `Tree` and the associated context. Likewise, the `longString` representations of `Environments` include the associated context and tree. Here is a more complex example:

```
> Tree.TopLevel simpleTree = NextGenTestCase.makeClassTree("public class C {}");
  Environment env = new Environment(simpleTree, null);
> env.longString()

"[Environment
  null
  {TopLevel for null
    packageId: null packageSymbol: null
    starImportScope (except for java.lang): null
    namedImportScope: null
    =====
    {ClassDef C<> extends null implements () flags: 1
      null
      }]}
"
```

`Environments` can be spawned from other `Environments` as follows:

```
> e1 = new Environment(null, null);
> e2 = e1.spawn(null);
```

When printed, spawned `Environments` first print out their own contents, and then print the contents of their enclosing `Environment`:

```
> e2.longString()

"[Environment
  null
  null]
[Environment
  null
  null]
"
```

Of course, we can also spawn `Environments` containing non-null `Trees`:

```
> Tree.TopLevel innerTree = NextGenTestCase.makeClassTree("public class C {}");
  Environment innerEnv = new Environment(innerTree, null);
```

```

Tree.TopLevel outerTree = NextGenTestCase.makeClassTree("public class D {}");
Environment outerEnv = innerEnv.spawn(outerTree);
> outerEnv.longString()

"[Environment
  null
  {TopLevel for null
    packageId: null packageSymbol: null
    starImportScope (except for java.lang): null
    namedImportScope: null
    =====
    {ClassDef D<> extends null implements () flags: 1
      null
    }}]
[Environment
  null
  {TopLevel for null
    packageId: null packageSymbol: null
    starImportScope (except for java.lang): null
    namedImportScope: null
    =====
    {ClassDef C<> extends null implements () flags: 1
      null
    }}]
"
```

Notice that the original `Environment` is unaffected by the spawn.

A.7.6 Symbols

`Symbols` are containers for identifiers. They also hold relevant information such as the bytecode generated for an identifier. The various types of `Symbols` form a composite class hierarchy rooted at class `Symbol`. All subclasses are static nested classes of class `Symbol`. The composite hierarchy is represented in BNF notation in Figure A.7.6.¹⁸

Each `ClassSymbol` contains a `Scope` of its members. Because each method's bytecode is stored in its corresponding `MethodSymbol`, and because the `Scope` of a `ClassSymbol` contains all `Symbols` for methods (and other members) appearing in a

¹⁸Notice that concrete class `TypeSymbol` forms the root of a composite subhierarchy, and also represents naked type parameters! An important refactoring would be to make `TypeSymbol` abstract and add a new subclass for naked type parameters.

```

Symbol ::= TypeSymbol | VarSymbol | MethodSymbol | OperatorSymbol
TypeSymbol ::= (type variable) TypeSymbol | ClassSymbol | PackageSymbol

```

Figure A.2 : Symbols

class, all of the information needed to write out a classfile for a given class can be accessed in the class's `ClassSymbol` after bytecode generation.

A.7.7 SymbolTables and ClassReaders

During Phase II, a `SymbolTable` is constructed and kept as a field in `JavaCompiler` for use in subsequent phases. A `SymbolTable` contains a field `classReader` (of type `ClassReader`) that stores all constituent symbols. `ClassReaders` store their symbols in two `Hashtables`: one mapping package names to their corresponding `PackageSymbols`, and one mapping class names to their `ClassSymbols`. In addition to all classes explicitly referred to in a program, `SymbolTables` include all classes in `java.lang`. They also contain mappings for all Java infix operators. Because the `ClassSymbols` in `java.lang` and the infix operators are the same for all `SymbolTables`, they are omitted from the `longString` representation. To see precisely what is entered during `SymbolTable` initialization, refer to the source file `src/edu/rice/cs/nextgen/compiler/comp/SymbolTable.java`.

In the DrJava interactions session below, we create a new `SymbolTable` corresponding to NextGen acceptance test `InstanceofParameter.java`. The text of that test is:

```

package edu.rice.cs.nextgen.simpletests;

public class InstanceOfParameter<T> {
    public boolean is(Object o) {
        boolean ret = o instanceof T;
        return ret;
    }
}

```

Here is the interactions session:

```
> load JavaCompilerTestSetup.hist

> trees = _compiler.parseFiles(_fileNames);
> envs = _compiler.enterClasses(trees);
> _compiler.getSymbolTable().longString()

"[SymbolTable
  packages:
    edu.rice.cs.nextgen => [PackageSymbol edu.rice.cs.nextgen]
    edu.rice => [PackageSymbol edu.rice]
    edu.rice.cs.nextgen.simpletests =>
      [PackageSymbol edu.rice.cs.nextgen.simpletests]
    edu.rice.cs => [PackageSymbol edu.rice.cs]
    edu => [PackageSymbol edu]

  classes: edu.rice.cs.nextgen.simpletests.InstanceOfParameter =>
    [ClassSymbol
      fullname: edu.rice.cs.nextgen.simpletests.InstanceOfParameter
      flatname: edu.rice.cs.nextgen.simpletests.InstanceOfParameter
      flags: 4194305
      members (cyclic references have been filtered):
        [Scope: [MethodSymbol is]
          [MethodSymbol <init>]]]]]
"
```

A.7.8 CodeGenerators, GenContexts, and ClassWriters

Class `JavaCompiler` drives class generation by first using a `CodeGenerator` visitor to write bytecode to a `ClassSymbol`, and then passing that `ClassSymbol` to method `ClassWriter.writeClassFile()`, on an instance of `ClassWriter` stored in the `SymbolTable`.

When generating bytecode for a given `Tree`, `CodeGenerators` keep information for each subtree in an `Environment<GenContext>`. Class `GenContext` is a non-public class, written in source file `CodeGenerator.java`, that stores the following information for each subtree:

1. The `expectedType` of the subtree (a `Type`).

2. All unresolved `exitingJumps` out of the code represented by the subtree, represented as a `Chain`. `Chains` are described below.
3. All unresolved `continuingJumps` to the code represented by the subtree, also represented as a `Chain`.

Class `Chain` is a static nested class of class `Code` (contained in source file `Code.java`).¹⁹ Conceptually, `Chains` are lists of jumps. Each `Chain` contains three fields:

1. The `next Chain` in the list.
2. `pc` (an `int`). This field represents the position of the jump instruction.
3. `stackSize` (an `int`). The stack size after the jump instruction.²⁰

An important invariant of a chain is that all elements of the chain list have the same `stackSize`.

Writing to Disk

Once all bytecode has been generated and stored in the `ClassSymbols`, `ClassWriters` are responsible for writing class files to disk. The `ClassSymbols` to be written are not stored in class `ClassWriter`. Instead, each class component is passed as an argument to the appropriate method, and written to a file.

¹⁹Instances of `Code` represent all information that goes into a code attribute in a classfile.

²⁰When the locations of jumps are resolved, the compiler ensures as a sanity check that the `stackSize` is non-negative in each block of code. In an old (unreleased) version of the NEXTGEN compiler, jump targets were resolved incorrectly because snippet methods moved the locations of all code in a parametric class. As a result, the sanity check on `stackSizes` would fail during compilation of some classes.

Appendix B

A Sample Conversion of the NextGen Compiler

The following program is a short implementation of a list utility written in NextGen. It is followed by source code indicating the various conversions that the compiler would perform on this code. Each conversion is annotated with an inlined comment discussing why the conversion was done. Although the compiler would ordinarily output Java bytecode, a source code representation is presented to facilitate readability. Likewise, class names have not been mangled, as the name conversion process (described above) is straightforward. Each parameterized class name in the converted file should be understood to designate the corresponding mangled class name.

B.1 The Original Source Code

```
import java.util.*;

public class ListWrapper<T> {
    private final List _list;

    public ListWrapper(List list) throws IllegalArgumentException {
        ListIterator itor = list.listIterator();
        while (itor.hasNext()) {
            if (!(itor.next() instanceof T)) {
                throw new IllegalArgumentException("Input list contains +
                                                element not of type T!");
            }
        }
        _list = list;
    }

    public ListWrapper() { this(new ArrayList()); }

    public void add(T item) { _list.add(item); }
```

```
public WrappedIterator<T> getIterator() {
    return new WrappedIterator<T>(_list.listIterator());
}

public T[] toArray() {
    return (T[]) _list.toArray(new T[0]);
}

public static void main(String[] args) {
    ListWrapper<String> w = new ListWrapper<String>();
    w.add("a");
    w.add("b");
    w.add("c");

    WrappedIterator<String> itor = w.getIterator();
    while (itor.hasNext()) {
        String val = itor.next();
        System.out.println("item: " + val);
    }

    String[] sArray = w.toArray();
    System.out.println("Type of sArray: " +
        sArray.getClass().getName());

    LinkedList iList = new LinkedList();
    iList.add(new Integer(5));

    // This should throw an exception!
    w = new ListWrapper<String>(iList);
}

class WrappedIterator<T> {
    private final ListIterator _itor;

    public WrappedIterator(ListIterator itor) { _itor = itor; }

    public boolean hasNext() { return _itor.hasNext(); }

    public T next() { return (T)_itor.next(); }
}
```

B.2 The Transformed Source Code

```

import java.util.*;

/**
 * Base class for ListWrapper. Note it has no type
 * parameters.
 */
public class ListWrapper {
    private final List _list;

    public ListWrapper(List list) throws IllegalArgumentException {
        ListIterator itor = list.listIterator();
        while (itor.hasNext()) {
            if (! ListWrapper$$instanceof$T(itor.next()) ) {
                throw new IllegalArgumentException("Input list contains +
                    element not of type T!");
            }
        }
        _list = list;
    }

    public ListWrapper() {
        this(new ArrayList());
    }

    /** Argument type was erased. */
    public void add(Object item) {
        _list.add(item);
    }

    /** Return type was erased. */
    public WrappedIterator getIterator() {
        return ListWrapper$$newWrappedIterator$T$(_list.listIterator());
    }

    /** Return type was erased. */
    public Object[] toArray() {
        return (Object []) _list.toArray(ListWrapper$$newArray$T$1$0(0));
    }

    public static void main(String[] args) {
        ListWrapper<String> w = new ListWrapper<String>();
        w.add("a");
        w.add("b");
    }

```

```

w.add("c");

// Notice the return type of getIterator was erased.
WrappedIterator itor = w.getIterator();
while (itor.hasNext()) {
    // The return type of next() was erased, so we insert
    // a GJ-style cast that is guaranteed to succeed.
    String val = (String) itor.next();
    System.out.println("item: " + val);
}

// toArray's static type was erased to Object[].
// However, since the snippetized array creation uses
// the actual type parameter to create the array, this
// cast will succeed.
String[] sArray = (String[]) w.toArray();
System.out.println("Type of sArray: " +
    sArray.getClass().getName());
LinkedList iList = new LinkedList();
iList.add(new Integer(5));

// This should throw an exception!
w = new ListWrapper<String>(iList);
}

/* Abstract snippets. */
protected abstract Boolean ListWrapper$$instanceof$(Object o);
protected abstract WrappedIterator
    ListWrapper$$newWrappedIterator$(ListIterator itor);

/**
 * Abstract snippet for new T[]. The two numbers after
 * $$ refer to
 * the number of dimensions passed to the array creation
 * expression (1)
 * and the number of initial values passed to it (0).
 */
protected abstract Object[]
    ListWrapper$$newArray$(int len);
}

/**
 * Template interface for ListWrapper. A different copy of
 * this template, with T substituted with the actual type
 * parameter, will be loaded

```

```

    * for each instantiation.
    */
public interface ListWrapper<T>$ {}

/**
 * Template class for ListWrapper. A different copy of this
 * template, with T substituted with the actual type
 * parameter, will be loaded
 * for each instantiation.
 */
public class ListWrapper<T> extends ListWrapper
    implements ListWrapper<T>$
{
    /* Forwarding constructors. */
    public ListWrapper<T>(List list) throws IllegalArgumentException {
        super(list);
    }

    public ListWrapper<T>() throws IllegalArgumentException { super(); }

    /* Concrete snippets. */
    protected boolean ListWrapper$$instanceof$T(Object o) {
        // T$ will be the instantiation interface for T if T is
        // an instantiated parametric class. Otherwise T$ will
        // just be T.
        return o instanceof T$;
    }

    protected WrappedIterator
        ListWrapper$$newWrappedIterator$T$(ListIterator itor)
    {
        return new WrappedIterator<T>(itor);
    }

    protected Object[] ListWrapper$$newArray$T$1$0(0)(int len) {
        return new T[len];
    }
}

/**
 * Base class for WrappedIterator. Note it has no type
 * parameters.
 */
class WrappedIterator {
    private final ListIterator _itor;

```

```

public WrappedIterator(ListIterator itor) {_itor = itor;}

public boolean hasNext() {
    return _itor.hasNext();
}

/**
 * Notice that the return type was erased to Object.
 * However, we are assured we will only return an
 * instance of T due to
 * the snippetized cast.
 */
public Object next() {
    return WrappedIterator$$castTo$T(_itor.next());
}

/* abstract snippet. */
protected abstract Object WrappedIterator$$castTo$T(Object o);
}

/**
 * Template interface for WrapperIterator. A different copy
 * of this template, with T substituted with the actual
 * type parameter, will be
 * loaded for each instantiation.
 */
interface WrappedIterator<T>$ {}

/**
 * Template interface for WrapperIterator. A different copy
 * of this template, with T substituted with the actual
 * type parameter, will be
 * loaded for each instantiation.
 */
class WrappedIterator<T> extends WrappedIterator
    implements WrappedIterator<T>$
{
    /* Forwarding constructor. */
    public WrappedIterator<T>(ListIterator itor) {
        super(itor);
    }

    /* Concrete snippet. */
    protected Object WrappedIterator$$castTo$T(Object o) {

```

```
// TB will be the base class of T (the erased name) if
// T is an instantiated parametric class. Otherwise TB
// will just be T. T$ will be the instantiation
// interface for T if T is an instantiated parametric
// class. Otherwise T$ will just be T.
// The two casts are necessary here, each for its own
// reason:
// T$: This cast ensures that o is an instance of T
//     or one of its subclasses. If T is parametric,
//     we must check this via the instantiation
//     interface because a subclass of T may not
//     directly extend T! For example, Stack<String>
//     not Vector<String>. But Stack<String> does
//     implement Vector<String>$.
// TB: This cast ensures that o has the methods that T
//     has. If T is parametric, we must check this via
//     the base class of T, since that is where the
//     public interface of T actually resides.
//
// Note that if T is not parametric, these two casts
// will be the same, a harmless redundancy.
return (TB) (T$) o;
}
}
```

Bibliography

- [1] O. Agesen, D. Detlefs. *Mixed-mode Bytecode Execution*. Sun Microsystems Technical Report SMLI TR-2000-87, June, 2000.
- [2] O. Agesen, S. Freund, J. Mitchell. *Adding parameterized types to Java*. In OOPSLA 1997.
- [3] E. Allen, J. Bannet, R. Cartwright. *Mixins in Generic Java are Sound*. Technical Report. Rice University, 2003.
- [4] E. Allen, R. Cartwright. *The Case for Run-time Types in Generic Java*. Principles and Practice of Programming in Java, June 2002.
- [5] E. Allen, R. Cartwright. *Safe Instantiation in Generic Java*. To appear in Principles and Practices of Programming in Java, 2003.
- [6] E. Allen, R. Cartwright, B. Stoler. *Efficient Implementation of Run-time Generic Types for Java*. IFIP WG2.1 Working Conference on Generic Programming, July 2002.
- [7] D. Ancona, G. Lagorio, E. Zucca. *JAM-A Smooth Extension of Java with Mixins*. In ECOOP 2000, LNCS, Springer Verlag, 2000.
- [8] D. Ancona, E. Zucca. *A theory of mixin modules: algebraic laws and reduction semantics*. Universita di Geneva. 1999.
- [9] D. Ancona, E. Zucca. *A Theory of Mixin Modules: Basic and Derived Operators*. Mathematical Structures in Computer Science, 8(4):401–446. 1998.
- [10] P. Baldan, G. Ghelli, A. Raffaeta. *Basic theory of F-bounded quantification*. Information and Computation 153(1), p. 173-237. 1999.
- [11] J. Bloch, N. Gafter. *Personal communication*. 2002.
- [12] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. dissertation, Dept. of Computer Science, University of Utah 1992.
- [13] G. Bracha, W. Cook. *Mixin-based inheritance*. In OOPSLA 1990 .
- [14] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. *GJ Specification*. 1998.

- [15] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. *Making the future safe for the past: adding genericity to the Java programming language*. In OOPSLA 1998.
- [16] K. Bruce, G. Longo. *A modest model of records, inheritance, and bounded quantification*. Information and Computation v. 87, p. 196-240. 1990.
- [17] K. Bruce, A. Schuett, R. van Gent. *PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language*. In ECOOP 1995.
- [18] L. Cardelli. *A semantics of multiple inheritance*. Lecture Notes in Computer Science, v. 173 p. 51-69. Springer-Verlag. 1984.
- [19] L. Cardelli, S. Martini, J. Mitchell, A. Scedrov. *An extension of system F with subtyping*. Information and Computation 109(1-2), 4-56. 1994.
- [20] L. Cardelli, P. Wegner. *On understanding types, data abstraction, and polymorphism*. Computer Surveys 17(4), p. 471-522. 1985.
- [21] R. Cartwright, G. Steele. *Compatible genericity with run-time types for the Java programming language*. In OOPSLA 1998.
- [22] W. Cook. *A Denotational Semantics of Inheritance*. PhD dissertation, Brown University, 1989.
- [23] W. Cook. *A Proposal for Making Eiffel Type-safe*. In ECOOP 1989.
- [24] P. Cunning, W. Cook, W. Hill, W. Olthoff, J. Mitchell. *F-bounded Quantification for Object-Oriented Programming*. ACM FPCA. 1989.
- [25] P. Curien, G. Ghelli. *Coherence of subsumption: Minimum typing and type-checking in $F_{<}$* . Mathematical Structures in Computer Science v. 2, p. 55-91. 1992.
- [26] K. Fisher, J. Reppy. *Inheritance-Based Subtyping*. Seventh Workshop on Foundations of Object-Oriented Languages. 2000.
- [27] M. Flatt, S. Krishnamurthi, M. Felleisen. *Classes and Mixins*. In POPL 1998.
- [28] M. Flatt, S. Krishnamurthi, M. Felleisen. *A Programmer's Reduction Semantics for Classes and Mixins*. Formal Syntax and Semantics of Java, volume 1523, June 1999.
- [29] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.
- [30] G. Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD dissertation, Universita di Pisa. 1990.

- [31] G. Ghelli. *Recursive types are not conservative over $F_{<}$* . *Typed Lambda Calculi and Applications*, p. 146-162. Springer-Verlag, 1993.
- [32] J. Girard. *Interpretation fonctionnelle et elimination des coupures de l'arithmetic d'ordre superieur*, PhD dissertation, Universite Paris. 1972.
- [33] J. Gosling, B. Joy, G. Steele. *The Java Language Specification*. Addison-Wesley. Reading, Mass. 1996.
- [34] A. Igarashi, B. Pierce, P. Wadler. *Featherweight Java: A minimal core calculus for Java and GJ*. In OOPSLA 1999.
- [35] A. Igarashi, B. Pierce. *On Inner Classes*. In ECOOP 2000.
- [36] A. Igarashi, M. Viroli. *On Variance-Based Subtyping for Parametric Types*. In ECOOP 2002.
- [37] A. Kennedy, D. Syme. *Design and implementation of generics for the .NET Common Language Runtime*. In PLDI 2001.
- [38] S. McDirmid, M. Flatt, W. Hsieh. *Jiazzi: New Age Components for Old Fashioned Java*. In OOPSLA 2001.
- [39] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall. 1988.
- [40] D. Moon. *Object-oriented Programming with Flavors*. In OOPSLA 1986.
- [41] A. Myers, J. Bank, B. Liskov. *Parameterized Types for Java*. In POPL 1997.
- [42] M. Odersky, P. Wadler. *Pizza into Java: translating theory into practice*. In POPL 1997.
- [43] B. Pierce. *Bounded quantification is undecidable*. *Information and Computation* 112(1), p. 131-165. 1994.
- [44] B. Pierce. *Types and Programming Languages*. MIT Press. 2002.
- [45] J. Reynolds. *Towards a theory of type structure*. *Colloque sur la Programmation*, p. 408-425. 1974.
- [46] A. Snyder. *CommonObjects: An Overview*. SIGPLAN Workshop on Object-oriented Programming. 1986.
- [47] G. Steele. *Growing a Language*. Higher-Order and Symbolic Computation. 1999.
- [48] Sun Microsystems Inc. *JSR-14 v1.3 prototype compiler source code*. 2003.

- [49] Sun Microsystems, Inc. *JSR 14: Add Generic Types To The Java Programming Language*. Available at <http://www.jcp.org/jsr/detail/14.jsp>.
- [50] B. Stoustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.
- [51] M. Viroli, A. Natali. *Parametric Polymorphism in Java: an Approach to Translation Based on Reflective Features*. In OOPSLA 2000.
- [52] J. Wells. *Typability and type checking in the second-order λ -calculus are equivalent and undecidable*. Ninth Annual IEEE Symposium on Logic in Computer Science. 1994.