INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeaeb Road, Ann Arbor, Mi 48106-1346 USA
800-521-0600

UMI®
Efficient Implementation of Run-time
Generic Types for Java

by

Eric Ethan Allen

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED. THESIS COMMITTEE:

Robert Cartwright, Professor, Chair
Computer Science

Dan Wallach, Assistant Professor
Computer Science

Scott Rixner, Assistant Professor
Computer Science and
Electrical and Computer Engineering

HOUSTON, TEXAS
MAY, 2002
ABSTRACT

Efficient Implementation of Run-time

Generic Types for Java

by

Eric Ethan Allen

One of the most significant limitations of the Java Programming Language is its lack of support for generic types. This thesis describes an efficient compiler and run-time system for NextGen, a compatible extension of Java that supports generic types, including type dependent operations. The NextGen compiler is implemented as an extension to the existing compiler for GJ, a limited generic extension of Java that does not support type dependent operations. Our implementation is homogeneous: each instantiation of a generic class is represented by a distinct subclass of a common base class, which contains all code except for atomic operations that depend on the particular instantiation. To support polymorphic recursion, a customized class loader generates instantiations of generic classes on demand. A suite of benchmarks testing the Java, GJ, and NextGen compilers shows that supporting type dependent operations has little or no overhead compared to ordinary Java and GJ.
ACKNOWLEDGMENTS

I would like to thank Professor Robert Cartwright, designer and co-developer of the NextGen compiler, and graduate student Brian Stoler, implementer of the NextGen class loader, for their feedback and suggestions.

I would also like to thank my wife, Kori, for her support, encouragement, and constructive feedback.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Glossary</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Related Work</td>
<td>3</td>
</tr>
<tr>
<td>3 Motivation</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>The Singleton Pattern</td>
<td>6</td>
</tr>
<tr>
<td>Casting and Catching Generic Types</td>
<td>10</td>
</tr>
<tr>
<td>Functional Maps over Arrays</td>
<td>11</td>
</tr>
<tr>
<td>Legacy Classes and Interfaces</td>
<td>12</td>
</tr>
<tr>
<td>Transparent Debugging</td>
<td>13</td>
</tr>
<tr>
<td>4 Design Fundamentals</td>
<td>15</td>
</tr>
<tr>
<td>Bridge Methods</td>
<td>17</td>
</tr>
<tr>
<td>Instantiation Classes and Snippet Methods</td>
<td>20</td>
</tr>
<tr>
<td>Modeling Parametric Types in a Simply-Typed Class Hierarchy</td>
<td>21</td>
</tr>
<tr>
<td>Snippet Methods</td>
<td>26</td>
</tr>
<tr>
<td>Extensions of Parametric Classes</td>
<td>27</td>
</tr>
<tr>
<td>5 Design Complications</td>
<td>29</td>
</tr>
<tr>
<td>Cross-Package Instantiation</td>
<td>29</td>
</tr>
<tr>
<td>Security in the Presence of Instantiation Classes</td>
<td>31</td>
</tr>
<tr>
<td>Extending the Existing Java Collections Classes</td>
<td>32</td>
</tr>
<tr>
<td>6 Implementation</td>
<td>34</td>
</tr>
<tr>
<td>Flattening Parametric Names and Snippet Generation</td>
<td>34</td>
</tr>
<tr>
<td>The NextGen Classloader</td>
<td>38</td>
</tr>
</tbody>
</table>
GLOSSARY

**Base Class.** A simple class corresponding to a generic class where all parametric types have been erased and all type dependent operations have been replaced by calls on abstract snippet methods.

**Bridge Method.** An automatically generated forwarding method added to a class in order to satisfy a defined method signature in an instantiation class.

**Distinguished Subexpression.** A subexpression of a snippetized expression $E$, whose value must be passed as an argument to the snippet method generated for $E$ in order to preserve $E$’s behavior.

**Flattened Class Name.** The name of an instantiation type, encoded as a valid simple name in such a way as to prevent conflicts with Java class names.

**Ground Instantiation Type.** A simple type or an instantiation type whose arguments are all ground instantiation types.

**Homogeneous Implementation.** An implementation of generic classes in which all instantiations of a class share the code that implements the methods of that class. A *fully homogeneous* implementation of generic types relies on type erasure to map all instantiations of a generic type into a single base class.
**Instantiation Class.** A synthetically generated class whose instances represent the members of a particular ground instantiation type. Each instantiation class extends a base class and an instantiation interface.

**Instantiation Interface.** An empty interface, used to represent a particular ground instantiation type. Instantiation interfaces are used in the multiple inheritance architecture necessary to implement NextGen in Java.

**Instantiation Type.** An application of a parametric type, binding the various type parameters to arguments.

**Naked Snippet.** A snippet that performs a type dependent operation directly on the parameter of a generic class, e.g., a snippet for operation `new T();` in a generic class `C<T>.

**Naked Type.** A parametric type consisting of a single type variable, e.g., `T` in class `C<T>.

**Name Mangling.** The process by which the NextGen compiler converts the name of an instantiation type into a collection of names for the corresponding instantiation class and interface. This process is guaranteed to prevent conflicts with Java class names.

**Parametric Type (a.k.a. Generic Type).** A class or interface parameterized by one or more type variables. These variables are bound in the body of the corresponding class or interface definition.

**Simple Name.** The name of a simple type.
Simple Type. A class or interface without any parameters, corresponding to an ordinary Java class or interface.

Snippet Environment. A special class containing naked snippets for type dependent operations on package-private classes, to be used in contexts outside of the corresponding package.

Snippet Method. An automatically generated method that performs a type dependent operation. Snippet methods are declared as abstract in the base class of a parametric type, and defined appropriately for each instantiation class.

Template Class File. A special class file used by the NextGen customized class loader to generate instantiation classes on demand.

Type Bound. A type expression bounding the instantiation of a type parameter. Type bounds are not permitted to be naked types.

Type Dependent Operation. An operation that may be performed only in the presence of run-time generic type information.

Type Erasure. A translation of a potentially parametric type expression to a simple type expression, performed by eliminating all type variables from the instantiation. A naked type is erased to the erasure of its bound.
INTRODUCTION

Adding generic types to the Java programming language would greatly benefit Java programmers in their efforts to develop, maintain, and debug robust software. Generic types enable a programmer to parameterize classes and methods with respect to type, identifying important abstractions that otherwise cannot be stated in a statically typed language. Generic type declarations allow a type checker to analyze these abstractions and perform far more precise static checking than is possible in a simply typed language such as Java [6]. In fact, much of the casting done in Java is the direct consequence of not having generic types. In the absence of generic types, a Java programmer is forced to rely on a clumsy idiom to simulate parametric polymorphism: The universal type Object (or suitable bounding type) is used in place of a type parameter T, and casts are inserted where necessary to convert expressions of the erased type Object to a particular instantiation type. This idiom obscures the type abstractions in the program, clutters the program with casting operations, and significantly degrades the precision of static type checking.

Despite the obvious advantages of adding generic types to Java, such an extension would be of questionable value if it meant sacrificing compatibility either with the Java Virtual Machine (JVM) or the wealth of Java legacy code. Fortunately, as the Pizza [9], GJ [2], and Sun JSR-14 compilers have shown, it is possible to compile Java with generic types into bytecode for the existing JVM. However, the source languages supported by these compilers all impose significant restrictions on the use of genericity. In particular, program
operations that depend on generic type information are forbidden. The prohibited operations include casts, instanceof tests, and new operations of naked type such as new T () and new T [], and per-instantiation static field accesses (a language feature introduced in NextGen). We call such operations type dependent.

The Pizza, GJ, and JSR-14 compilers do not support type dependent operations because they all perform type erasure to translate Java with generics to ordinary Java bytecode. In essence, these languages implement generic types using the programming idiom described above. At the source level, the awkwardness of the idiom is largely hidden; the only observable effect is the prohibition against type dependent operations. These operations cannot be supported because the requisite generic type information is not available at run-time; the compiler has erased it. NextGen overcomes the limitations of Pizza, GJ, and JSR-14 by introducing a separate Java class for each distinct instantiation of a generic type; all parametric type information is preserved by the compiler and is available at run-time. Hence, type dependent operations are fully supported by NextGen. On the other hand, NextGen retains essentially the same level of compatibility with legacy code as GJ and JSR-14. For these reasons, we believe that NextGen provides an important step forward in the evolution of the Java programming language.
Chapter 2

RELATED WORK

The first generic Java compiler to support type dependent operations was an experimental compiler developed by Agesen, Freund, and Mitchell that relied on a purely heterogeneous implementation of generic classes: a complete, independent copy of a generic class was generated for each instantiation. In their implementation, a customized class loader generated complete class instantiations from template class files in much the same way that C++ expands templates [1].

The Pizza programming language, designed and implemented by Odersky and Wadler, was an ambitious extension of Java that supported not only parameterized classes and methods, but also algebraic data types and first-class closures. All of these features were implemented in such a way as to retain compatibility with the existing Java Virtual Machine. As mentioned in chapter 1, Pizza accomplishes this feat through type erasure, transforming generic type references into simple type references [9]. Consequently, type dependent operations are forbidden in GJ.

Odersky and Wadler adapted Pizza into GJ, a more restrained extension of Java that adds only generic types. By narrowing the set of extensions they added to Java, and by implementing GJ via type erasure, they were able to provide complete forward and backward compatibility of Java bytecode. In particular, references to generic versions of the Java collections classes were transformed into references to the existing, unparameterized base classes [9]. Consequently, old and new classes refer to the same collections classes. However,
the GJ compiler pays heavy penalties for this approach. These shortcomings will be discussed in detail in chapter 4.

Recently, Sun Microsystems has released the JSR-14 prototype compiler for Java extended with generic types. The language implemented by this compiler is identical to GJ. In fact, the JSR-14 compiler is a refactored version of the original GJ compiler. Sun improved the robustness of that compiler, and added a substantial amount of documentation. But JSR-14 relies on the same type erasure strategy as Pizza and GJ, and, therefore, it cannot support type dependent operations. In the remainder of this thesis, we will use GJ as the representative of this type erasure strategy. Implicitly, discussions of the limitations of GJ should be understood to apply to Pizza and JSR-14 as well.

Another related implementation of generic types is the PolyJ extension of Java developed at MIT [8]. The PolyJ website suggests that PolyJ is similar to NextGen in some respects, but the only published paper on PolyJ describes a very different approach to implementing genericity. In particular, their approach involves modification of 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. The language design includes a second notion of interface that uses a more flexible matching scheme than Java interfaces. 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 generic type implementation that most closely resembles NextGen is the extension of the .NET common runtime by Kennedy and Syme to support generic types in C# [7]. They follow the same homogeneous approach to implementing genericity described
in the NextGen design [6]. 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. To handle polymorphic recursion, 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 concerns than ours. They have not yet addressed the problem of supporting polymorphic methods.

Unlike C#, Java language evolution runs a significant risk of not supporting type dependent operations. In the next chapter, we will see why this lack of support has many very negative consequences for effective software development. Furthermore, subsequent chapters will explain how Java could be extended to support type dependent operations, describe an implementation of such an extension, and demonstrate that this implementation incurs little to no performance penalty when compared to Java and GJ.
Chapter 3

MOTIVATION

The NextGen programming language adds generic types to Java in such a way as to support type dependent operations. As I 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 first-class types; they can be used anywhere that conventional types can. The NextGen compiler employs 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 Java programmers to write cleaner, more reliable code.

The Singleton Pattern

One of the most widely applicable design patterns in programming practice is the Singleton Pattern. 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 the field. To enforce this practice, the class constructors can be declared private.
For example, consider the following class hierarchy for representing binary trees:

```java
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`:

```java
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 GJ, a static field of generic class is shared across all instantiations of the class. This design choice is dictated by the fact that GJ uses type erasure to implement generic types: a generic class is translated to a conventional class — called a base class — by converting all generic type references to their bounds (which are conventional types).

Hence, there can only be one static ONLY field 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 disjoint. For this reason, the GJ compiler prohibits the type.

The only way to support the Singleton Pattern in the GJ 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.

In contrast, NextGen, which implements 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. If a static field declaration includes the new modifier \texttt{generic}, then each instantiation of the generic type contains its own copy of this generic field. In the case of class Leaf<T>, this allows us to apply the Singleton Pattern in the natural way:

```java
    class Leaf<T> extends Tree<T> {

        static <T> final Leaf<T> ONLY = new Leaf<T>();
    }
```

1 Because we have focused development on implementing features necessary to measure performance benchmarks across platforms, we have deferred implementation of per-instantiation static fields.

2 NextGen is a proper extension of GJ, so static fields without the \texttt{generic} modifier are shared in NextGen just as they are in GJ.
Each instantiation of class Leaf will then contain its own field ONLY of the appropriate type.

**Casting and Catching Generic Types**

In GJ, casting an expression $M$ to a generic type $C<E>$ is prohibited unless parametric type information in the static type of $M$ matches $E$ — 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 is 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 any interesting 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 the generic type $C<E>$ of the cloned object, which is prohibited in GJ 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.
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 resultant elements. Generic types allow us to perform such transformations with far less casting. For example, consider the following parametric Command interface:

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

Suppose we wanted 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 this in NextGen as follows:

```java
public class ArrayMapper {
    <A,B> public static B[] mapArray<A,B> (Mapper<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 GJ does not support run-time generic type operations such as `new B[]`, the preceding code is invalid in GJ. 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.

**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. Furthermore, it must interoperate with existing compiled binaries. GJ accomplishes this goal 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 GJ compiler can be fooled into compiling generic class references into references to existing Java classes *if* the class path is set during compilation to include generic “stub” classes with the same signatures as the erasures of the non-generic classes they spoof. This feature enables GJ to re-interpret existing “naturally generic” library classes or other binaries as classes with generic signatures.

In addition, GJ 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 GJ program can pass type-checking, yet generate a run-time type error (a `ClassCastException`) at a point where there is no cast in the source program! [2,6]
The run-time error is produced by a cast, automatically inserted by the GJ 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-versa.\(^3\) and perform the conversions whenever "naturally generic" data is passed between legacy code and new code.\(^4\)

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.

**Transparent Debugging**

In the GJ 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

---

\(^1\) 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.
can only report the erased types of data objects, not their generic types. For example, an empty `List<Integer>` will simply be reported 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 completely consistent with the source code.

Generic types hold great promise for simplifying the application of many design patterns and coding practices in Java. But, as the preceding examples have shown, their full value cannot be realized without support for run-time generic types. If such operations that depend on run-time generic types are excluded from the language, programmers will find Java 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 example, support for run-time type operations is critical for type-safe compatibility with Java legacy code and the transparent debugging of generic Java code.

In the next chapter, we will discuss the design of the NextGen compiler, and how it manages to support type dependent operations without sacrificing compatibility.
Chapter 4

DESIGN FUNDAMENTALS

The NextGen source language is a proper extension of the GJ formulation of generic Java\textsuperscript{5}. In fact, NextGen and GJ were designed in concert with one another so that the two languages would have this property. In this common formulation, Java class definitions may be augmented by including parameters in the specification of class names, and referring to these parameters in the body of class definitions. The modified syntax for class names is as follows:

\[
\text{ClassDec ::= SimpleClassName} \\
| \quad \text{SimpleClassName<VarDec*}} \\
\]

\[
\text{VarDec ::= Var} \\
| \quad \text{Var extends ClassOrVarName}} \\
\]

\[
\text{ClassOrVarName ::= Var} \\
| \quad \text{Name} \\
\]

\[
\text{Name ::= SimpleClassName} \\
| \quad \text{SimpleClassName<ClassOrVarName*}} \\
\]

\textsuperscript{5} This upward compatibility does not include reflective operations.
where \texttt{Var} denotes the set of valid Java variable names, and \texttt{SimpleClassName} denotes the set of simple names (\textit{i.e.}, names without parameterization). \texttt{ClassDec} denotes the (possibly parameterized) class name that follows the keyword \texttt{class} in a class definition. The productions of \texttt{ClassOrVarName} can appear in any context where a class name can appear in ordinary Java, except as the superclass of a class definition or the superinterface of an interface definition. In these positions, only a production of \texttt{ClassName} can appear.

The scope of the type variables occurring in a \texttt{ClassDec} is the body of the corresponding class definition, including the bindings of the variable definitions themselves. However, type variables outside of \texttt{ClassNames} may not appear in the \texttt{extends} or \texttt{implements} clauses of a class definition.

Because NextGen was designed as a proper extension of GJ, understanding the NextGen compiler requires first understanding the GJ compiler. GJ supports parameterized classes and methods through type erasure. For each parametric class \texttt{C<T>}, GJ generates a single erased base class \texttt{C}; all of the methods of \texttt{C<T>} are implemented by methods of \texttt{C} with erased type signatures. The erasure of any parametric type \texttt{T} is obtained by replacing each type parameter in \texttt{T} by its bound (typically \texttt{Object}). For each program expression with erased type \texttt{T} appearing in a context with erased type \texttt{U} that is not a subtype of \texttt{T}, GJ automatically generates a cast to type \texttt{T}.
Bridge Methods

The erasure of parametric types creates a few complications, most notably the need for bridge methods when a class extends an instantiated generic class [1]. Because the type erasure process will reduce all types to their bounds, the signature of any method with an argument of generic type will be erased. If any class were to extend an instantiated class and override such a method, the type erased program would not pass typing checking; the signature of the overriding method would not match the method signature in the erased base class. For example, consider the following generic class:

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

The types in this class would be erased to form the following base class:

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

Now suppose that a programmer were to subclass an instantiation of Set<T>, with, say, Set<Integer>, and override \texttt{adjoin}:

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

The type of the parameter to \texttt{adjoin} in class \texttt{MySet} is correct, but it does not match the erased parameter type in the base class. Thus, the generated class file for class \texttt{MySet} will not pass bytecode verification.

GJ addresses this problem by inserting bridge methods into the subclasses of instantiated classes. These bridge methods match the erased signature of the method in the parent class, overloading the programmer defined method of the same name. Bridge methods simply forward their calls to the programmer-defined method, casting the arguments as necessary. In our example above, GJ would insert a bridge method into class \texttt{MySet} as follows:
class MySet extends Set<Integer> {
    public Set<Integer> adjoin(Object newElement) {
        return adjoin((Integer)newElement);
    }

    public Set<Integer> adjoin(Integer newElement) {
        ...
    }

    ...

}  

Then the types in this class would be erased as follows, producing a valid Java subclass of the base class Set:

class MySet extends Set {
    public Set adjoin(Object newElement) {
        return adjoin((Integer)newElement);
    }

    public Set adjoin(Integer newElement) {
        ...
    }

    ...

}  

Static type-checking guarantees that the inserted casts will always succeed. Of course, this strategy would not work if the programmer were to define an overloaded a method with the same signature as a generated bridge method. Therefore, overloading methods with arguments of generic type is restricted by the type checker.
Bridge methods are also necessary in some contexts where extensions of instantiation classes implement interfaces. Suppose that class \( C \) extends an instantiation class \( D \) and implements some interface \( I \). Furthermore, suppose that \( I \) includes a method \( M \) that refers to one of the type parameter instantiations defined in \( D \). Then \( C \) might inherit an implementation of \( M \) whose instantiated signature in \( C \) matches that of \( M \) in \( I \). If an instance of \( C \) is passed as an argument to a method that takes a parameter of type \( I \), the JVM will expect to find a method \( M \) of the appropriate signature in class \( C \). But the erased method signature in the generic ancestor won't match; a bridge method is needed.

**Instantiation Classes and Snippet Methods**

NextGen enhances the GJ implementation architecture by making the erased base class \( C \) abstract and extending \( C \) by the various instantiations of the generic class \( C<T> \), e.g., \( C<Integer> \), that occur during execution of a given program. These subclasses are referred to as instantiation classes. Each instantiation class includes forwarding constructors for the non-private constructors of \( C \). The type dependent operations of \( C<T> \) are replaced by calls on synthesized abstract methods called snippet methods [3], and these snippet methods are overridden by appropriate type specific code in the instantiation classes extending \( C \). The content of these snippet methods in the instantiation classes is discussed later in this chapter.
Modeling Parametric Types in a Simply-Typed Class Hierarchy

The most interesting feature of the NextGen architecture is the machinery that it uses to map generic Java to conventional Java in such a way that generic operations can be performed almost as efficiently as their simple counterparts. Figure 4.1 shows the hierarchy of Java classes used to implement the generic type Vector<T> and the instantiations Vector<Integer> and Vector<Double>. The overhead involved in executing a type dependent operation in the generic class Vector<Double> is an extra method call on the snippet that implements the type dependent operation.

When one generic class extends another, the simple JVM class hierarchy given in Figure 4.1 cannot represent the subtyping relationships that class instantiations must obey. For example, consider a generic class Stack<T> that extends a generic class Vector<T>. Any instantiation Stack<E> of the generic class Stack<T> must inherit code from the abstract base class Stack which inherits code from the abstract base class Vector. In addition, the type Stack<E> must be a subtype of Vector<E>. Hence, an instantiation class implementing the type 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 where the instantiation class Vector<E> has two superclasses.

Fortunately, we can exploit multiple interface inheritance to solve this problem. The Java type corresponding to a generic class instantiation C<E> can be represented by an empty instantiation interface C<E>$ which is implemented by the class C<E> (a dollar is appended to signify that the name of this generated interface must not clash with those of any existing classes or interfaces). Because a Java class can implement an interface (actually an unlimited number of them) as well as extend a class, the multiple inheritance problem goes away. Also, because these interfaces are empty, their construction has little impact on the code footprint. Figure 4.3 represents the same type structure as Figure 4.2 while conforming to the restriction of single class inheritance.

Therefore, the transformation of a generic class C<T> into a corresponding, infinite, set of GJ classes can be described as follows:
1. Generate an abstract snippet method for each expression \( E \) in \( C<T> \) involving a type dependent operation.

2. Replace each such expression \( E \) in \( C<T> \) with an expression that invokes the new snippet method with the appropriate arguments.

3. Erase all types in the transformed class \( C<T> \) to produce the base class \( C \) for \( C<T> \).

4. For every instantiation of \( C \) of ground instantiation type that occurs during program execution, generate an instantiation interface for \( C \) and all of its superclasses and superinterfaces in which any of the type parameters of \( C \) occur.

5. For every instantiation of \( C \) of ground instantiation type that occurs during program execution, generate an instantiation class for \( C \) and all of its superclasses in which any of the type parameters of \( C \) occur.
6. Insert the appropriate forwarding constructors and concrete snippet methods into each instantiation class of $C$. The concrete snippet methods override the inherited abstract snippet with a method that performs the appropriate type dependent operation. The forwarding constructors simply invoke `super` on the constructor arguments.

Much of the complexity of this process is a result of steps four and five. One might think that it would be possible to statically determine a bound $U$ on the set of possible instantiations of each parametric class in a program, and then generate class files corresponding to each instantiation in $U$. Each of these instantiation classes could then extend the appropriately erased base class and interface as described by Cartwright and Steele [3]. The bodies of these classes would consist solely of constructors and snippet method definitions, overriding the abstract declarations in the parent class with specific type
dependent operations. Because all of the types in instantiation classes are ground (i.e., contain no type variables), the definition for each snippet method in a snippet class is trivial.

However, early in the process of building an implementation of the NextGen compiler, we discovered that the collection of all potential instantiation classes across all possible program executions is infinite for some programs. Infinite collections of potential instantiations are possible because generic Java permits polymorphic recursion, i.e., a method may call itself recursively with arguments of type specified recursively in terms of the input types. For example, consider the following parametric class:

```java
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<T> that reads a sequence of integer values from console input specifying the arguments for calls on the method nest for a receiver object of type C<String>. For any finite input sequence, the set of instantiations of C during program execution is finite, but an infinite input sequence may require infinitely many
instantiations. Moreover, 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. The customized class loader searches the class path to locate these template files as needed, and uses them to generate loaded class instantiations. To reduce the overhead of generating instantiation classes, the customized class loader maintains a cache of the template class files that have already been read.

**Snippet Methods**

As mentioned above, expressions involving type dependent operations in a NextGen program are replaced with calls to abstract snippet methods, defined in a base class, and overridden in each instantiation class. The snippet method in each instantiation class \( C \) must perform the type dependent operation appropriately for the particular ground instantiation type corresponding to \( C \). For most type dependent operations, e.g., new expressions and per-instantiation static field accesses, generation of the appropriate type dependent operation is straightforward. But a small complication arises in the case of casts and instanceof tests. In a naïve implementation, the code in the snippet method performs the cast or instanceof test on the specified instantiation type class. This implementation fails because some subtypes of \( C<E> \) are not subclasses of the instantiation class for \( C<E> \). For example, consider the class hierarchy for Vectors and Stacks depicted in
Fig. 4.3. A cast to `Vector<Integer>` must succeed when applied to an instance of `Stack<Integer>`, but the instantiation class for `Stack<Integer>` does not inherit from the instantiation class for `Vector<Integer>`.

The solution to this problem is to perform the type dependent operation on the instantiation interface, since all subtypes of the ground instantiation type will inherit from it. In the case of `instanceof` tests, that's all that must be done. But, for casts, it's still not enough. Although a successful cast to the instantiation interface will prove that the cast object is of the appropriate type, the JVM will not permit any method invocations or field accesses on the object, because the instantiation interface is empty. Therefore, it is necessary to cast to the base class corresponding to the instantiation class as well. Casting only to the base class would not be enough because the base class will have valid subclasses that are not subtypes of the cast type (such as `Vector<Double>`). It is necessary to include both casts in the snippet body for completeness.

**Extensions of Parametric Classes**

An interesting complication arises in the NextGen class generation architecture when a generic class `C` extends another generic class `D`. If `D` is a ground instantiation type, this case can be handled simply by modifying `C` to extend the instantiation class corresponding to `D`. But if `D` is not a ground instantiation type, the situation gets more complicated. The base class of `C` must extend that of `D`, requiring any instantiation class of `C` to implement the abstract snippets contained in the base class of `D`. Furthermore, the bodies of these snippet implementations must respect the bindings of the type variables.
of $D$ to the type expressions assigned to them in the extends clause of $C$. With multiple implementation inheritance, we could handle this case by defining each instantiation class $C_I$ of $C$ to extend both the base class of $C$ and an instantiation class $D_I$ of $D$ that binds the type variables of $D$ based on the type bindings in $C_I$. Such a strategy is illustrated in Figure 4.4.

Because multiple implementation inheritance is not available in Java, we directly implement the snippet methods of $D$ in each instantiation class of $C$. Thus, the template class file of $C$ must include these snippet methods. We can construct the appropriate snippet bodies for this class file by examining the template class file for $D$ and copying the template snippet bodies, expanding each reference to a type parameter to its binding in terms of the type parameters of $C$. 

Figure 4.4. Illegal class hierarchy in naive extension of parametric class $D$ by $C$, with instantiation class $C'$. 
Chapter 5

DESIGN COMPLICATIONS

The fundamental design for NextGen described above is complicated by several problems that emerge when additional aspects of the Java run-time environment are taken into account. The solutions to these problems, as laid out in [3], will be described presently.

Cross-Package Instantiation

The above design does not discuss the packages in which the base and instantiation classes of a generic class C<T> are placed. Of course, the simplest place to put them is in the same package as C, and, in fact, this is what NextGen does. But doing so raises some problems.

Consider the case where a generic class C<T>, defined in package P, is instantiated in the body of class D, defined in package Q, as class C<E>, where E is a package-private class in package Q. Furthermore, suppose that the body of class C contains type dependent operations. Then the snippet bodies generated for instantiation class C<E> will not be able to access class E. Specifically, they will not be able to perform type dependent operations on E. Let us refer to snippets that perform type dependent operations directly on the instantiation of a type parameter as naked snippets.

The simplest solution to the problem of cross-package instantiation is to define NextGen to automatically widen package-private classes to public visibility when they are
used to instantiate a generic type in another package. Although one might argue that the automatic widening of a language construct is an undesirable feature, such widening has precedent in Java: When an inner class refers to the private members of its enclosing class, Java widens the visibility of the private members of the enclosing class by generating getters and setters with package visibility [5]. This feature is solely a consequence of the way that inner classes are implemented in Java: They are compiled to top-level class files. These class files would not have access to the private members of their enclosing classes if getters and setters were not added. Although more secure (and expensive) implementations are possible, the Java language designers chose to sacrifice some visibility protection for the sake of performance. This decision was well justified; in practice, the loss of visibility security has not proven to be a significant issue. Therefore, we propose that the visibility issues concerned with generic class instantiation be solved in a similar manner.

Nevertheless, it is possible to implement NextGen in such a way that class visibility security is preserved. One solution, as laid out by Cartwright and Steele, is to pass snippet environments to the constructors of the instantiation classes of $C$ [3]. This environment is an object containing the naked snippet methods for class $E$. Then the snippet methods defined in $C<E>$ can forward calls on these type dependent operations to the snippet environment. But this solution requires that the constructors laid out in the template class of $C$ take snippet environments as arguments.

The requisite bookkeeping can be done by the compiler and recorded in the compiled class files, for use by the class loader, which determines the necessary form of snippet environment when an instantiation class is loaded and generates an appropriate singleton class to represent the environment. The overhead in this approach is the extra
complexity in the compiler and class loader, the extra work that the class loader must perform, and the extra level of dispatching on the snippet environment to execute snippets.

A simple alternative to snippet environments would be 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 implementing the same operation. In essence, this 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 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. The class-per-snippet scheme is not as secure as the snippet environment scheme because the snippets (in public snippet classes) generated during program execution can be executed by any program class.

Security in the Presence of Instantiation Classes

Given the NextGen design described above, one security issue is how to prevent an attacker from spoofing the instantiation classes of a generic class C<T>. To prevent such spoofing, we could include in each client class of C a special snippet method, declared to
be final, that would use reflection to make the appropriate checks on any supposed instantiation of \( C<T> \) before invoking any method on it. But we expect that this solution would degrade performance, as an extra method invocation would be involved in any method call on an instantiation class. In all likelihood, there will be performance degradation involved with any secure implementation of run-time generic types that does not actually modify the JVM.

**Extending the Existing Java Collections Classes**

One of the most useful applications of generic types for Java would be to extend the Java collections classes with generic versions. But, in most JVMs, the packages containing the collections classes are sealed. This problem can be solved by putting the collections classes in new packages `nextgen.java`... with generic classes that subclass the originals. There are accessibility issues with this solution concerning the final, static, and private members of the original classes, as follows:

1. Final members cannot be overridden/shadowed in the generic classes. Fortunately, there are no final members in these classes after Java 1.2.

2. Static members in these classes will be shared across the various instantiation classes. This issue is best handled by including a facility for shared static members into NextGen. It is possible to include such a facility, but only at the expense of complicating the NextGen semantics.
3. Private members of these classes are, of course, inaccessible in the generic extensions. However, private members are inaccessible to all client classes, so it is not necessary to access them in the generic extensions to ensure backward compatibility with existing clients.

Notice that, unlike GJ, instances of the old collections classes are not instances of the new collections classes in NextGen. Thus, new objects are not interchangeable with their old counterparts in all contexts. To handle this issue, programmers extending legacy code to work with the generic collections classes would have to convert old instances of the collections classes into instances of the new base classes.

Explicit conversion helps keep such hybrid code robust by forcing programmers to document the invariants they believe about the data contained in legacy objects. Moreover, the conversion of old instances of these classes is the unavoidable cost of type soundness. The interoperation with legacy code in NextGen is fully type-checked. In contrast, GJ achieves interoperability by discarding parametric type-checking. Therefore, when interoperating with legacy code, it is not clear what type-checking advantages GJ provides over ordinary Java.
Chapter 6

IMPLEMENTATION

The GJ compiler is organized as a series of visitors that transform a parsed abstract syntax tree into byte code. We have extended this compiler to support NextGen by inserting an extra visitor into the series that detects expressions in the program requiring the use of snippets, adds them to 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 the namespace visible to the NextGen programmer, nor to that used for inner class names, because they include the character sequence $$, disallowed in Java source code and in the mangled names of inner classes. We have also modified the GJ code generator to accept these newly generated names.

The added visitor destructively modifies the syntax tree to add the extra snippet methods. It replaces the expressions including type dependent operations with snippet invocations. Finally, it keeps a counter to ensure that the snippet names it generates are distinct.

Flattening parametric names and snippet generation

The first piece of our implementation of NextGen is a visitor over NextGen ASTs that detects expressions in the source code involving type dependent operations. In order to describe the transformation of such an expression $E$, it is necessary to define the notion of
a distinguished subexpression of $E$. Intuitively, a distinguished subexpression of $E$ is any subexpression whose value must be passed as an argument to the snippet method generated for $E$ to preserve $E$'s behavior:

- The distinguished subexpressions of a new expression are the arguments passed to the constructor.
- The distinguished subexpression of a cast expression is the expression that is being cast.
- The distinguished subexpression of an `instanceof` expression is the left-hand side of the subexpression.

Now, with that definition in hand, we can describe the transformation of each expression $E$ involving applications of type dependent operations over parametric types as follows:

1. Generate a new snippet name, corresponding to the operation performed in $E$, and guaranteed not to conflict with any existing names in the source tree, including snippet names.
2. Generate an abstract snippet with that name and add it to the enclosing class.
3. Replace $E$ with an application of the abstract snippet, passing in as arguments the values of any distinguished subexpressions of $E$. 
4. Recursively visit the distinguished subexpressions of $E$, in case they also involve run-time type operations on parametric types.

Once this transformation is performed, the next step is to generate a template class for every generic class $C<T>$ in the program. This process is described in the next section.

A template class file looks like a conventional class file except for the fact that some of the strings in the constant pool contain embedded references of the form $\{0\}$, $\{1\}$,... to actual type parameters of the class instantiation (which are represented as mangled strings). The class loader replaces these embedded references by the corresponding actual type parameters (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. In order to prevent clashes between the mangled names for instantiation classes and the names of ordinary Java classes, we have relied on the convention that the $\$$ character does not appear in class names in NextGen source code. The implementation of inner classes in Java relies on exactly the same assumption. We also assume that the only source of $\$$ characters in the names of Java class files produced by other Java compilers is the name mangling process used to implement inner classes. Although $\$$ is a legal component of a Java identifier, the Java Language Specification stipulates that it “should be used only in mechanically generated source code or, rarely, to access preexisting names on legacy systems.” All Java compilers (e.g., javac, gic) that comply with the Java Language Specification conform to this convention [5].
Our 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
```

The name mangling performed by the implementation of inner classes substitutes $ for the periods separating inner classes from their enclosing classes, but this usage will never produce a name that includes a sequence of two consecutive dollar signs. Therefore, by including this sequence in our mangled class names, we have ensured that we will never clash with the name space of the inner classes. Furthermore, because all of our character conversions insert exactly three characters into a class name, the character sequence $$x$$ for
any character x, ensures that the occurrence of x was indeed inserted by our name mangling scheme. Therefore, we can rely on the inserted characters x to determine the identity of instantiation classes.

The NextGen Class Loader

When a NextGen class refers to a generic type in a type dependent operation, the corresponding class file uses a mangled name to refer to the generic type. Because we defer the generation of instantiation classes and interfaces until run-time, no actual class file exists for the mangled name corresponding to a generic type. Our custom class loader intercepts requests to load classes (interfaces) with mangled names and uses the corresponding template class (interface) file to generate the requested class (interface). In particular, it replaces each embedded reference tag of the form \( \{ 0 \}, \{ 1 \}, \text{ etc.} \), by the mangled names of the corresponding ground instantiation types in the argument list of the requested instantiation class. Specifically, this replacement is done as follows:

- A constant pool entry of the form \( \{ n \} \), where n is an integer, is replaced by the name of the class bound to parameter n.

- A constant pool entry of the form \( \{ n \} \$ \), where n is an integer, is replaced as follows: If the class bound to parameter n is parametric and not an interface, \( \{ n \} \$ \) is replaced by the name of the parametric interface corresponding to parameter n. (That is, the name of the class prepended with \$). Otherwise it's replaced by the
name of class bound to parameter \( n \). This form of replacement is used for the instantiation interface cast in a cast snippet.

- A constant pool entry of the form \( \{ n \}B \), where \( n \) is an integer, is replaced as follows: If the class bound to parameter \( n \) is parametric, it is replaced with the name of the base class of parameter \( n \). Otherwise, it's replaced with the name of the class bound to parameter \( n \). This form is used for the base class cast in cast snippets.

- A constant pool entry of the form \( \text{prefix}\{n\}\text{contents}\{r\}\text{suffix} \), where \( \text{contents} \) contains one or more substrings of the form \( \{ n \} \) (\( n \) an integer) is replaced as follows: Each token \( \{ n \} \) inside \( \text{contents} \) is replaced with the name of the class bound to parameter \( n \), substituting \( \{D\} \) for occurrences of periods. This substitution fills in type parameters that are passed to other classes. For example, if \( \text{HashMap}\{\text{Key},\text{Value}\} \) were to reference \( \text{Pair}\{\text{Key},\text{Value}\} \) internally there would be a reference in class \( \text{HashMap}'s \) constant pool to \( \text{Pair}\{1\}\{C\{0\}\} \).

After this replacement, the class file denotes a valid Java class. An example of a NextGen program before and after replacement is provided in Appendix A.

**Code Generation**

Ideally, we would like to be able to implement the NextGen compiler as a source-level transformation, with the output fed to the existing GJ compiler. Such an
implementation would allow us to decouple development and maintenance of NextGen from the maintenance of GJ. Unfortunately, implementing NextGen in this way is not possible; the names of instantiated generic types are invalid names in GJ, and are rejected by the code generator. Therefore, it was necessary to modify the code generation process in the GJ compiler slightly to accept these names. These modifications were local and did not affect the overall structure of the process.
Chapter 7

PERFORMANCE

Because no established benchmark suite for generic Java exists, it was necessary to construct our own benchmark suite to measure the performance of NextGen. Existing Java benchmark suites like JavaSpecMark are not appropriate because they do not make any use of the generic facilities that NextGen provides [1]. Additionally, many of the popular benchmark suites for Java do not make heavy use of the object-oriented features of the language, relying instead on computationally intensive procedural code to measure Java performance. Our benchmark suite consists of the following programs:

- **sort**: An implementation of generically typed linked lists, and a quick sort method over them that uses a provided Ordering object.
- **mult**: A visitor over generically types binary trees of integers that multiples the values of the nodes.
- **zeros**: A visitor over binary trees that determines whether there is any child-parent pair, both of which hold the value 0.
- **buff**: An implementation of java.util.Iterator over a BufferedReader. This benchmark constructs a large, buffered, StringReader, and then iterates over the elements.
• **set**: An implementation of multi-sets, and set-theoretic operations on them. This benchmark constructs large multisets and compares them as they are built.

• **bool**: A simplifier of parsed Boolean expressions. This ~500 line program reads in a large number of such expressions from a file, and simplifies each in turn.

The benchmarks were written in GJ. Furthermore, they were written specifically to take advantage of the added type checking provided by GJ.

The source code was then copied and manually modified to produce equivalent Java source code. This modification was minimal, and consisted entirely of the same kinds of modification that are made by the GJ compiler itself (e.g., erasing types to their bounds, inserting casts where necessary, etc). Manual modification was necessary because the transformation performed by the GJ compiler sometimes involves modifications that can be made only at the bytecode level, so corresponding source code cannot always be generated. The modified source code was compiled using the Sun JDK 1.3 compiler on Linux.

Similarly, the source code was copied and compiled with the NextGen compiler. Although we could have modified the source code to take advantage of the added type checking provided by NextGen, we opted to leave it unmodified, in order to ensure as objective a test of NextGen performance as possible.

The results of these benchmarks for Java, GJ, and NextGen under five separate JVMs are illustrated in Figs. 6.1-6.5. These results were obtained by running each benchmark twenty 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 on each JVM, for each language, exhibited significant variance, the results of the first run were uniformly dropped. We attribute this
variance to both the overhead of JVM startup, and initial JIT compilation of the code, neither of which is relevant to our experiment. Once the first run was dropped, the variance of the results for each benchmark was less than 10% of the total running time. The mean running times are presented in table form in Appendix B.

The results for GJ also apply to JSR-14, as the generated class files for these compilers are virtually 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 have any impact on performance.

The most striking feature of these results is that the inclusion of run-time support for generic types does not add significant overhead, 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 results across JVMs, for each language tested.

These results can be explained by considering what costs are incurred by keeping the run-time type information. Once an instantiation of a template class is read into memory, the only added overhead of genericity is the added method call involved in invoking the 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.

When running on modern JVMs, even type dependent operations incur little overhead. Because the evaluation of these operations involve extra snippet calls, performance will be degraded somewhat. However, on modern JVMs, such snippet method calls are dynamically inlined, virtually eliminating the overhead of these operations.

Our benchmark was specifically designed to make heavy use of generic types, and yet, even in this context, generic types added no significant performance overhead.
Therefore, we are confident that performance of the vast majority of Java programs would not be affected at all by the inclusion of support for type dependent operations. On the other hand, the robustness and maintainability of many programs would be greatly enhanced.
Figure 7.1. Performance Results on Sun 1.3 Server (milliseconds)

Figure 7.2. Performance Results for Sun 1.3 Client (milliseconds)
Figure 7.3. Performance Results for IBM 1.3 (milliseconds)

Figure 7.4. Performance Results for Sun 1.4 Server (milliseconds)
Figure 7.5. Performance Results for Sun 1.4 Client (milliseconds)
Chapter 8

FUTURE EXTENSIONS

In addition to complete support for generic classes, there are other language features involving more sophisticated type systems that facilitate good software engineering practices. In this section, we will discuss some of these features, and how they might be implemented on top of the existing NextGen compiler.

Parameter Kinds

In the case of a type dependent operation involving a new expression on a type parameter, it would never make sense to instantiate that parameter with an interface or abstract class. The NextGen type checker could ensure that such instantiations never occur if we extend the language to include prefixed annotations on parameter declarations. These annotations would specify the kind of a type parameter, i.e., class, abstract class, or interface. The new syntax for generic type declarations would be:

```
ClassDec ::= SimpleClassName
           | SimpleClassName<VarDec*>

VarDec ::= AnnotatedVar
          | AnnotatedVar extends ClassOrVarName
```
AnnotatedVar ::= Var
    | Kind Var

Kind ::= class
    | abstract class
    | interface

ClassOrVarName ::= Var
    | ClassName

ClassName ::= SimpleClassName
    | SimpleClassName<ClassOrVarName*>}

By default, a type parameter would be assumed to be of interface kind, unless there is an extends clause (in which case abstract class is assumed). Notice that the type checker, when checking instantiations of parameters, must check not only for new expressions, but also for instantiations of generic classes with parameters of the wrong kind. For example, suppose we have a class \texttt{C<T>}, where \texttt{T} is of class kind, and class \texttt{D<S>}, where \texttt{S} is of interface kind. Then \texttt{C<S>} is an invalid instantiation inside the body of \texttt{D}. In general, parameters of class and abstract class kind may not be instantiated with those of interface kind. But all other cross-kind instantiations are permitted.

Extending the NextGen compiler to support kind annotations is solely a matter of augmenting the parser and type checker. No modifications to the generated class and template files, nor to the augmented class loader, are necessary.
Full Support for Polymorphic Methods

One of the attractive features of the GJ programming language is the ability to abstract individual methods with respect to type variables. The bindings of these variables are inferred automatically at each method invocation site. The existing NextGen compiler inherits this functionality from GJ, but since the GJ compiler relies solely on type erasure, the NextGen compiler does not fully support type dependent operations within polymorphic methods. We are in the process of extending NextGen to support polymorphic methods in their full generality.

Polymorphic methods are more complex to implement than generic classes because polymorphic methods can be overridden. In addition, the type parameter bindings in a polymorphic method invocation come from two different sources: the call site and the receiver type. Integrating these two sources of run-time type information requires a more sophisticated dynamic class loading mechanism than the one currently used in the NextGen compiler.

Final polymorphic methods can be translated to generic inner classes, but this simple approach does not work in the presence of method overriding. This reduction does not yield a particularly efficient implementation either because each polymorphic method call involves allocating an instance of a generic inner class.

Our design relies on using a heterogeneous translation for polymorphic methods within generic classes and passing class objects to transmit the type information from the call site to the receiver object. If the polymorphic method code for the receiver requires snippets that depend on the type parameters from the call site, then the receiver explicitly
calls the custom class loader to load a generic snippet environment class by passing the class objects for all of the type arguments. The loaded environment class is a singleton class defining a snippet environment containing all of the snippets required to implement the type dependent operations in the method that involves call site type parameters.

This design is appealing because it has almost no overhead in the common cases (i) where a polymorphic method requires no snippets and (ii) where a polymorphic method only requires type dependent operations based on the type parameters provided by the receiver. In both cases, the only cost is the bytecode required to push the class objects (which are constants) for the type parameters at the call site; these parameters are ignored by the polymorphic method code in the receiver. The heterogeneous translation of the polymorphic method in the generic receiver class means that the method body can directly include all of the type dependent operations that depend only on the receiver class instantiation.

**Covariant Subtyping of Type Parameters**

A simple but useful extension of NextGen, described by Carwright and Steele, 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 $T$ [3]. Extending the NextGen compiler to support this feature is straightforward. It involves (i) trivially modifying the parser to support syntax for covariant type variable declarations, (ii) extending the type checker to cope with covariant generic types (the typing rules for covariant generic types are more restrictive than they are for non-covariant generic types), and (iii) 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.

As an aside, it should be noted that GJ and NextGen already support covariance in method return types.

**Mixins as Classes with Variable Parent Types**

The NextGen language, and the existing compiler, do not permit the occurrence of a naked type in the extends or implements clauses of a class definition. However, an extension of the language that allowed such class definitions would be very useful, because it would effectively provide linguistic support for mixins. Mixins provide a mechanism for inheriting implementation code from more than one class without the complications of multiple inheritance.

By allowing for the occurrence of a type variable in the extends and implements clauses of a class, NextGen would provide the developer with a way to bind the parent class of an object when it is constructed.

Classes with variable parent type could be supported through the use of a modified class loader that constructs classes with particular instantiated parent types from a template class file for the mixin class, a process strikingly similar to the current mechanism employed by the NextGen class loader to construct instantiations of generic classes. Therefore, extension of the NextGen class loader to support variable parent type is expected to be straightforward. However, the NextGen type system and type checker, would have to be extended to handle mixin types, which is a non trivial endeavor [6].
Chapter 9

CONCLUSION

Adding support for generic types is critical to the evolution of the Java programming language. 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 type dependent operations play in effective software development. We have presented numerous examples of how a lack of support for such operations inhibits the application of many widely practiced design patterns and practices. If type dependent operations are not added to Java, Java programs will be more difficult to develop, maintain, and debug.

As the NextGen compiler design demonstrates, we can achieve essentially the same level of compatibility with existing Java 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 GJ and NextGen compilers. Finally, we have described many ways in which the NextGen compiler could be extended to include additional features, such as parameter kinds, and classes with variable parent type. In many cases, implementation of these additional features would be much more difficult in a language that did not support type dependent operations.

For all of these reasons, leaving support for type dependent operations out of an extension of Java with generic types would be a costly mistake. Hopefully, our proof of concept will be persuasive in determining the future of Java in this regard. Because the
addition of run-time type support is not only feasible, but attainable with little performance overhead, we are optimistic that, eventually, it will be included in the language.
BIBLIOGRAPHY


Appendix A

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.
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));
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();
    }
}
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$$newWrappedIterator$$TS$_list.listIterator();
}

/** Return type was erased. */
public Object[] toArray() {
    return (Object []) _list.toArray(ListWrapper$$newArray$ST$S$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$S$instanceof$ST$T$T$(Object o);
protected abstract WrappedIterator
    ListWrapper$S$newWrappedIterator$T$T$(ListIterator itor);

/**
   * Abstract snippet for new T[]. The two numbers after
   * $T$s 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$S$newArray$T$S$S$I$S$0$(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$;
        }
    }
protected WrappedIterator
    ListWrapper$SnewWrappedIterator$TS{ListIterator itor}
{
    return new WrappedIterator<T>(itor);
}
}

protected Object[] ListWrapper$SnewArray$TS1$S0{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$$castToST(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. TS will be the instantiation
        // interface for T if T is an instantiated parametric
        // class. Otherwise TS will just be T.
        // The two casts are necessary here, each for its own
        // reason:
        // TS: 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;
Appendix B

MEAN RUNNING TIMES FOR JAVA, GJ, AND NEXTGEN

The following results are the mean running times in milliseconds over twenty runs for each of six benchmarks on six separate JVMs. The first run for each benchmark was dropped to eliminate excessive variance caused by the overhead of JVM startup and initial JIT compilation.

<table>
<thead>
<tr>
<th>Sun 1.3 Server</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmarks</td>
<td>Java</td>
<td>GJ</td>
<td>NextGen</td>
</tr>
<tr>
<td>Sort</td>
<td>674.5</td>
<td>814.4</td>
<td>904.5</td>
</tr>
<tr>
<td>Mult</td>
<td>1086.8</td>
<td>1171.4</td>
<td>1172.3</td>
</tr>
<tr>
<td>Zeros</td>
<td>1098.8</td>
<td>1007.7</td>
<td>1093.3</td>
</tr>
<tr>
<td>Buff</td>
<td>726.8</td>
<td>669.3</td>
<td>742.5</td>
</tr>
<tr>
<td>Set</td>
<td>498.9</td>
<td>494.8</td>
<td>506.3</td>
</tr>
<tr>
<td>bool</td>
<td>1582.7</td>
<td>1642.2</td>
<td>1692.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sun 1.3 Client</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmarks</td>
<td>Java</td>
<td>GJ</td>
<td>NextGen</td>
</tr>
<tr>
<td>sort</td>
<td>1002.9</td>
<td>1202.6</td>
<td>1445.4</td>
</tr>
<tr>
<td>mult</td>
<td>1357.9</td>
<td>1611.5</td>
<td>1339.1</td>
</tr>
<tr>
<td>zeros</td>
<td>1194.0</td>
<td>1202.7</td>
<td>1233.9</td>
</tr>
<tr>
<td>buff</td>
<td>553.8</td>
<td>554.3</td>
<td>584.1</td>
</tr>
<tr>
<td>set</td>
<td>692.4</td>
<td>708.4</td>
<td>710.0</td>
</tr>
<tr>
<td>bool</td>
<td>1659.7</td>
<td>1658.2</td>
<td>1663.4</td>
</tr>
</tbody>
</table>
### IBM 1.3

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Java</th>
<th>GJ</th>
<th>NextGen</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>2567.6</td>
<td>3916.3</td>
<td>4423.3</td>
</tr>
<tr>
<td>mult</td>
<td>1049.7</td>
<td>1079.1</td>
<td>1050.5</td>
</tr>
<tr>
<td>zeros</td>
<td>763.9</td>
<td>772.4</td>
<td>997.8</td>
</tr>
<tr>
<td>buff</td>
<td>954.5</td>
<td>932.2</td>
<td>942.8</td>
</tr>
<tr>
<td>set</td>
<td>754.7</td>
<td>725.1</td>
<td>767.6</td>
</tr>
<tr>
<td>bool</td>
<td>827.9</td>
<td>837.2</td>
<td>755.8</td>
</tr>
</tbody>
</table>

### Sun 1.4 Server

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Java</th>
<th>GJ</th>
<th>NextGen</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>860.4</td>
<td>1163.2</td>
<td>1102.3</td>
</tr>
<tr>
<td>mult</td>
<td>2113.1</td>
<td>1936.6</td>
<td>1945.4</td>
</tr>
<tr>
<td>zeros</td>
<td>1170.9</td>
<td>1364.3</td>
<td>1367.4</td>
</tr>
<tr>
<td>buff</td>
<td>668.3</td>
<td>689.4</td>
<td>701.2</td>
</tr>
<tr>
<td>set</td>
<td>744.0</td>
<td>733.3</td>
<td>745.6</td>
</tr>
<tr>
<td>bool</td>
<td>2038.5</td>
<td>1559.4</td>
<td>1537.8</td>
</tr>
</tbody>
</table>

### Sun 1.4 Client

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Java</th>
<th>GJ</th>
<th>NextGen</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>1248.5</td>
<td>1507.9</td>
<td>1859.3</td>
</tr>
<tr>
<td>mult</td>
<td>2133.9</td>
<td>2157.4</td>
<td>2198.9</td>
</tr>
<tr>
<td>zeros</td>
<td>1665.4</td>
<td>1576.8</td>
<td>1576.0</td>
</tr>
<tr>
<td>buff</td>
<td>912.1</td>
<td>938.5</td>
<td>929.7</td>
</tr>
<tr>
<td>set</td>
<td>938.4</td>
<td>928.3</td>
<td>952.7</td>
</tr>
<tr>
<td>bool</td>
<td>1799.4</td>
<td>1777.4</td>
<td>1928.8</td>
</tr>
</tbody>
</table>