EFFICIENT IMPLEMENTATION OF RUN-TIME GENERIC TYPES FOR JAVA

by

Eric E. Allen

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science

Rice University

2001

Approved by ______________________________________

Chairperson of Supervisory Committee

____

____

____

Program Authorized
to Offer Degree ______________________________

Date ______________________________

RICE UNIVERSITY

ABSTRACT

EFFICIENT IMPLEMENTATION OF RUN-TIME
GENERIC TYPES FOR JAVA

Eric E. Allen

Chairperson of the Supervisory Committee: Professor Robert Cartwright
Department of Computer Science

An efficient compiler and run-time system is described for NextGen, a compatible extension of the Java programming language supporting run-time generic types as described by Cartwright and Steele. The NextGen compiler is implemented as an extension to the existing compiler for GJ, a generic extension of Java that does not support run-time generic types. It relies on the homogeneous implementation strategy proposed by Cartwright and Steele with one major exception: To support polymorphic recursion in the definition of generic classes, the compiler generates templates for instantiation classes and relies on a customized class loader to construct instantiations of generic classes on demand. This thesis includes an extensive set of benchmarks, specifically developed to stress the use of generic types. The benchmarks show that the additional code required to support run-time generic types has little or no overhead compared with ordinary Java and GJ.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

The author wishes to thank Professor 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.

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corresponding to a generic class where all parametric types have been erased and all type dependent operations have been replaced by calls on abstract snippet methods.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. The distinguished subexpressions of a type dependent operation **E** are any subexpressions whose values must be passed as arguments 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 class name, in such a way as to prevent name space clashes.

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

Instantiation Class. A synthetically generated class whose instances represent the members of a particular ground instantiation type. Instantiation classes extend both a base class and an instantiation interface.

Instantiation Interface. An interface consisting of no methods that represents a particular ground instantiation type. Instantiation interfaces are used to represent the multiple inheritance necessary to implement NextGen in Java.

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

Isolated 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>`.

Name Mangling. The process by which the NextGen compiler converts the name of an instantiation type into names for the corresponding instantiation class and interface. This process is guaranteed to prevent clashes with existing 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 Class 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 isolated 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 Dependent Operation. An operation that may be performed only in the presence of run-time generic type information. Namely: casts, `instanceof` tests, `new` expressions, and static field accesses.

Chapter 1

INTRODUCTION

One of the most common criticisms of the Java programming language is its lack of support for generic types. 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. Moreover, generic type declarations allow the type checker to analyze these abstractions and perform far more precise static type 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 values 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 JSR14 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" parametric type such as **`new T()`** and **`new T[],`** and static field accesses (since static fields are associated with instantiation types in NextGen). We call such operations *type dependent.*

The Pizza, GJ, and JSR14 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; it has been erased by the compiler. NextGen overcomes the limitations of Pizza, GJ, and JSR14 by introducing a separate Java class for each distinct instantiation of a generic type; all parametric type information is preserved by the compiler and 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 JSR14. For these reasons, we believe that NextGen provides an important step forward in the evolution of the Java programming language.

Chapter 2

DESIGN FUNDAMENTALS

The NextGen source language is a proper extension of the GJ formulation of generic Java. 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:

ClassDec → SimpleClassName | SimpleClassName<VarDec*>
VarDec → Var | Var `extends` **ClassOrVarName**
ClassOrVarName → Var | ClassName
ClassName → SimpleClassName
| SimpleClassName<ClassOrVarName*>

where **Var** denotes the set of valid Java variable names, and **SimpleClassName** denotes the set of simple class names (i.e., without parameterization). **ClassDec** denotes the (possibly parameterized) class name that follows the keyword "class" in a class definition. The productions of **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 **ClassName** can appear.

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

GJ supports parameterized classes and methods through type erasure. For each parametric class **C<T>**, 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. The erasure of any parametric type τ is obtained by replacing each type parameter in τ by its upper bound (typically **Object**). For each program expression with erased type σ appearing in a context with erased type τ that is not a subtype of σ, GJ automatically generates a cast to type τ.

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]. Since the type erasure process will reduce all types to their upper bounds, the signature of any method with an argument of variable type will be erased. If any class were to extend an instantiated class and override such a method, the program would not type check after type erasure because the signature of the overriding method would not match the method signature in the erased base class (in fact, the signature would be covariant in the type of the parameter). For example, consider the following generic class:

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

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

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

Now suppose that a programmer were to subclass an instantiation of **Set<T>**, say, **Set<Integer>**, and override **adjoin:**

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

The type of the parameter to **adjoin** in class **MySet** is correct, but it does not match the erased parameter type in the base class. Thus, the generated class file for class **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 **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, the overloading of methods with arguments of generic type is restricted by the type checker.

Instantiation Classes and Snippet Methods

NextGen enhances the GJ implementation scheme 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 design is the machinery that it uses to map generic Java to conventional Java without adding significant time or space overhead in program execution. Figure 1 shows the hierarchy of Java classes used to implement the generic type **`Vector<T>`** and the instantiations **`Vector<Integer>`** and **`Vector<Double>`**.

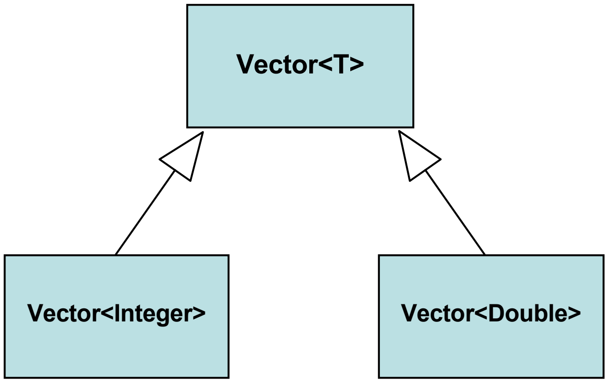


Figure 1. Naïve implementation of generic types over the existing Java class structure.

When one generic class extends another, the simple JVM class hierarchy given in Figure 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 2 shows this illegal hierarchy where the instantiation class **`Vector<`*`E`*`>`** has two superclasses.

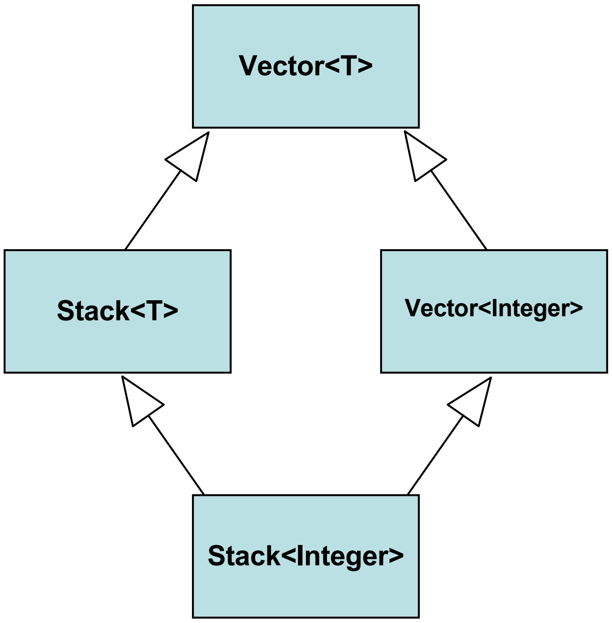


Figure 2. Illegal Class hierarchy in naive JVM Class Representation.

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). Since 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, since these interfaces are empty, their construction involves little code bloat. Figure 3 represents the same type structure as Figure 2 while conforming to the restriction of single class inheritance.

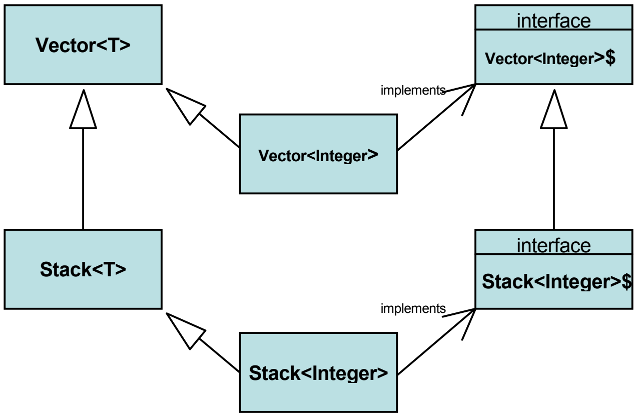


Figure 3. Simple Parametric Type Hierarchy and its JVM Class Representation.

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 super-interfaces 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 an upper 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 in [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. Since all of the types in instantiation classes are of ground instantiation type, 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 possible instantiation classes across all possible program executions was infinite for some programs 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:

```
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 run-time. 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 two of the four type dependent operations, i.e., **new** expressions and 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 body of a snippet method corresponding to these operations would simply perform the operation on its argument with the instantiation class appropriate to the context of the snippet. But this implementation is inadequate because casting would not succeed for all subtypes of that ground instantiation type. For example, consider the class hierarchy for **Vectors** and **Stacks** depicted in Fig. 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, as 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

One complication to this scheme arises in the case where a generic class **C** is declared to extend 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, things get 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′** of **C** to extend both the base class of **C** and an instantiation class **D′** of **D** that binds the type variables of **D** based on the type bindings in **C′.** Such a strategy is illustrated in Figure 4.

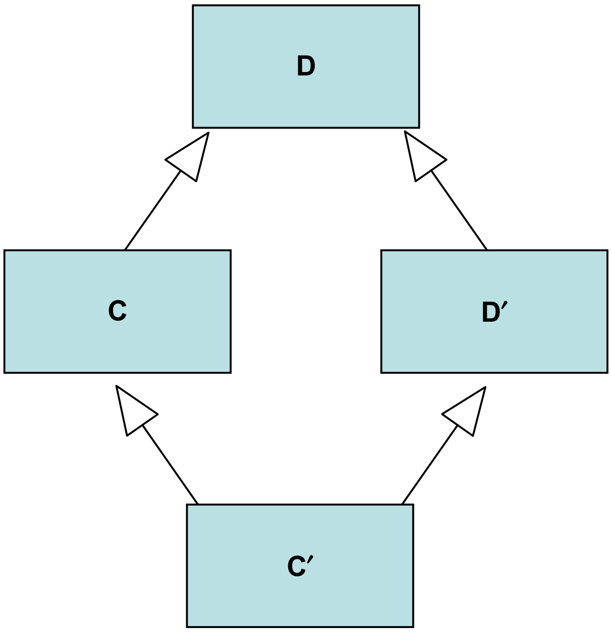


Figure 4. Illegal class hierarchy in naive extension of parametric class D by C, with

Since multiple implementation inheritance is not available in Java, it is necessary to instead 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.`

Extensions of Parametric Interfaces

In the case of parameterized interfaces, extensions are much easier to handle. Since all method bodies of an interface are abstract, they can't generate any snippets. The one complication is with static final fields that are initialized via type-dependent operations.

These fields must be replicated in each instantiation interface, but these instantiation interfaces can't include a type dependent operation. Therefore, such fields must be moved into each instantiation class that extends such an interface. A special snippet **`$getF`** is declared in the template interface file, and it must be defined by any class that extends such an interface. If the extending class is of ground instantiation type, the **`$getF`** method can be defined appropriately when its class file is generated. For other classes, the template class file must include the **`$getF`** snippet, and its body must consist of the original type dependent operation used to initialize the field, transformed according to the type bindings declared in the **`extends`** clause of the class (in a manner identical to that described above to extend parametric classes).

Chapter 3

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 *isolated 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 in [3], is to pass *snippet environments* to the constructors of the instantiation classes of **`C`**. This environment is an object containing the isolated 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.

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 `generic.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 base classes for the new NextGen collections classes. Thus, there can be problems in passing old instances of these classes for processing by methods in the new base classes. 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.

It should be noted that the conversion of old instances of these classes is the unavoidable cost of type soundness. In contrast to GJ, the interoperation with legacy code in NextGen is fully type-checked. Contrariwise, GJ achieves interoperability by discarding parametric type-checking. Therefore, when interoperating with legacy code, it not clear what type-checking advantages GJ provides over ordinary Java.

Chapter 4

IMPLEMENTATION

The GJ compiler is organized as a series of visitors that transform a parsed AST to 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, as they include the 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 AST 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.

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 , it is necessary to define the notion of a *distinguished subexpression* of . Intuitively, a distinguished subexpression of is any subexpression whose value must be passed as an argument to the snippet method generated for to preserve 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 involving run-time type operations over parametric types as follows:

1. Generate a new snippet name, corresponding to the operation performed in , 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 with an application of the abstract snippet, passing in as arguments the values of any distinguished subexpressions of .
4. Recursively visit the distinguished subexpressions of , 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 each class **`C`**, used to generate instantiation classes for **`C`** at runtime. 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 JLS stipulates that it "should be used only in mechanically generated source code or, rarely, to access preexisting names on legacy systems." All Java compilers (javac, gjc, JSR14) that comply with the Java Language Specification [5] conform to this convention.

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 class, 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, since 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. Since 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}", 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

the name of class bound to parameter **`n`**. This form of replacement is used for the instantiation interface cast in cast snippets.

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

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.

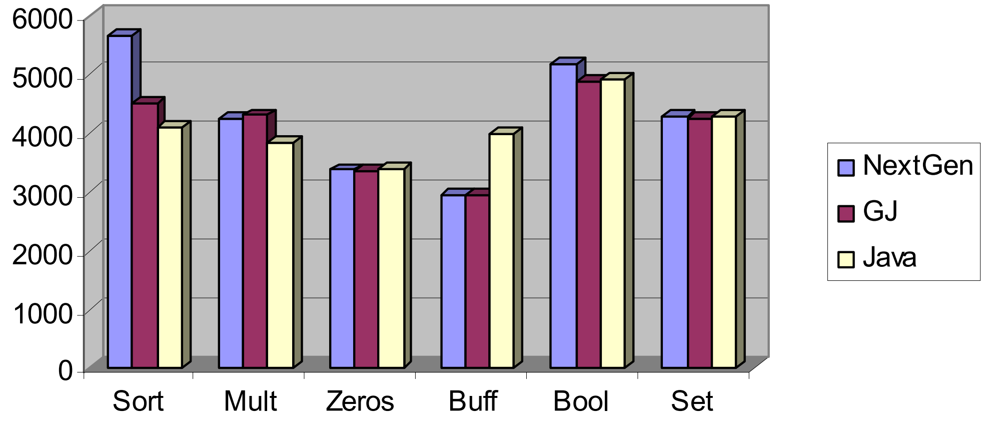
Code Generation

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, since the names of instantiated generic types are invalid names in GJ, and are rejected by the bytecode generator. Therefore, it was necessary to modify the bytecode 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 5

PERFORMANCE

Figure 5: Performance Results for Sun JDK 1.2 on Solaris (in milliseconds)

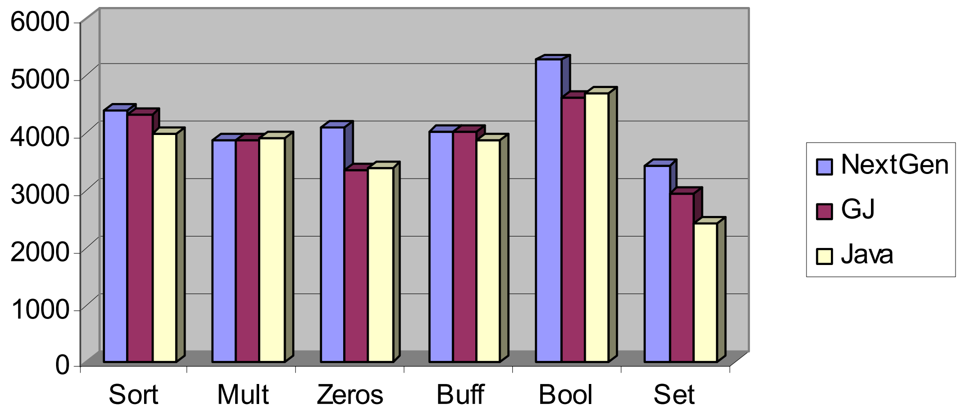


Since no established benchmark suite for generic Java exists, it was necessary to construct our own benchmark suit to measure the performance of NextGen. Existing Java benchmark suites like JavaSpecMark [1] are not appropriate because they do not make any use of the generic facilities that NextGen provides.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

Figure 6: Performance Results for Sun JDK 1.3 on Linux (in milliseconds)

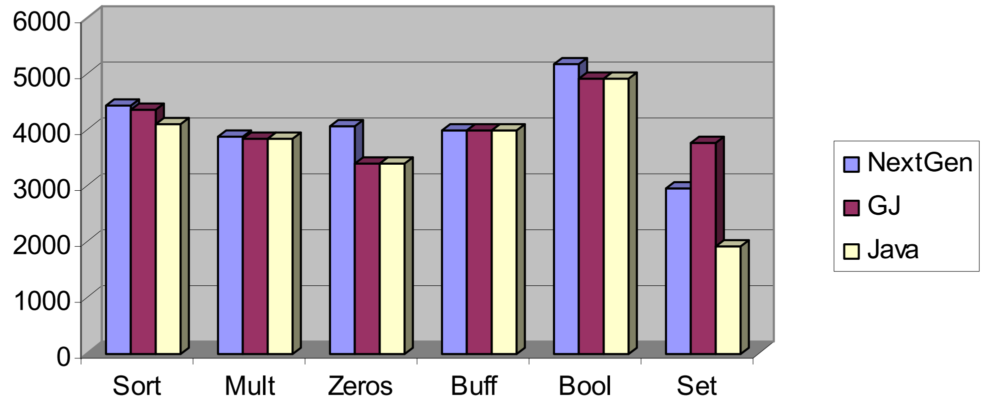


- **Zeros:** A visitor over binary trees that determines whether there is any child-parent pair, both of which hold the value 0.
- **Buff:** An implementation of **`java.util.Iterator`** over a **`BufferedReader`**. This benchmark constructs a large, buffered, **`StringReader`**, and then iterates over the elements.
- **Bool:** A simplifier of parsed Boolean expressions. This benchmark reads in a large number of such expressions from a file, and simplifies each in turn.
- **Set:** An implementation of multi-sets, and set-theoretic operations on them. This benchmark constructs large multisets and compares them as they are built.

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 amounted to many of the same modifications that are made by the GJ compiler itself (e.g., erasing types to their upper bounds, inserting casts where necessary, etc). Manual modification was necessary

Figure 7: Performance Results for IBM JDK 1.3 on Linux (in milliseconds)

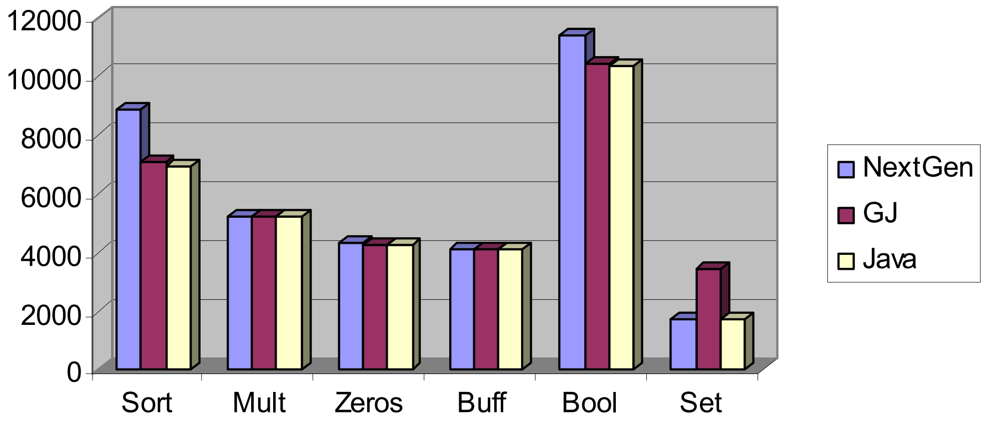


because the transformation performed by the GJ compiler sometimes involves modifications that can be made only at the bytecode level, so corresponding source code cannot always be generated. The modified source code was compiled using the Sun JDK 1.3 compiler.

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 three separate JVMs are illustrated in Figs. 5-8. These results are the mean running times, in milliseconds, of ten runs of each program. 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 (1) JSR-14 inserts an additional entry into the constant pool, and (2) JSR-14 by default makes the class file version 46.0 (the new Java 1.4 version tag). Neither of these differences have any impact on performance.

Figure 8: Performance Results for Sun JDK 1.4 on Linux (in millseconds)

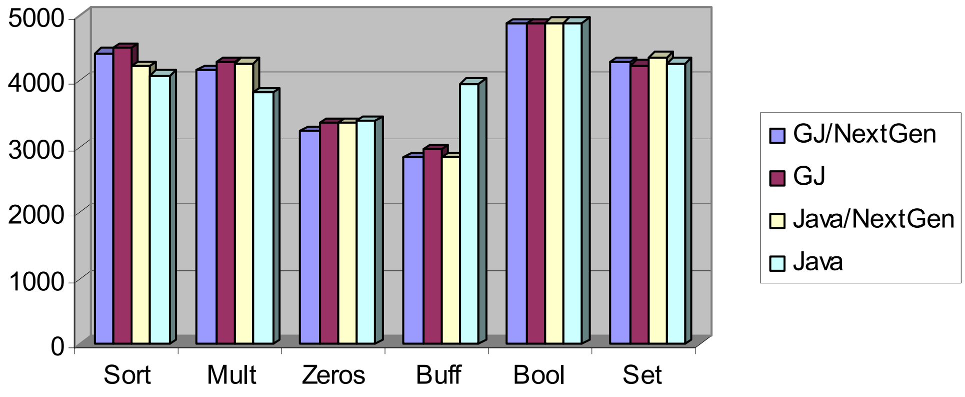


The results for fig. 5 (the Sun 1.2 JVM) were obtained running Solaris on a Sun Sparc 5. The results for figs. 6-8 (the Sun and IBM JVMs for Linux) were obtained running Red Hat Linux 6.0 on a 550 MHz dual-processor Pentium III.

The most striking feature of these results is that the inclusion of run-time support for generic types does not add significant overhead, even in programs that make heavy use of it. In fact, only in the **Sort** benchmark on the Sun JVM 1.2 and the **Set** benchmark on Sun JVM 1.3 does NextGen show any significant overhead at all.

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. Since most of the operations in an ordinary program are not type dependent operations, this small cost is amortized over a large number of instructions.

Figure 9: Performance Comparison of NextGen and Java Class Loaders for JDK 1.2 on Solaris (in milliseconds)

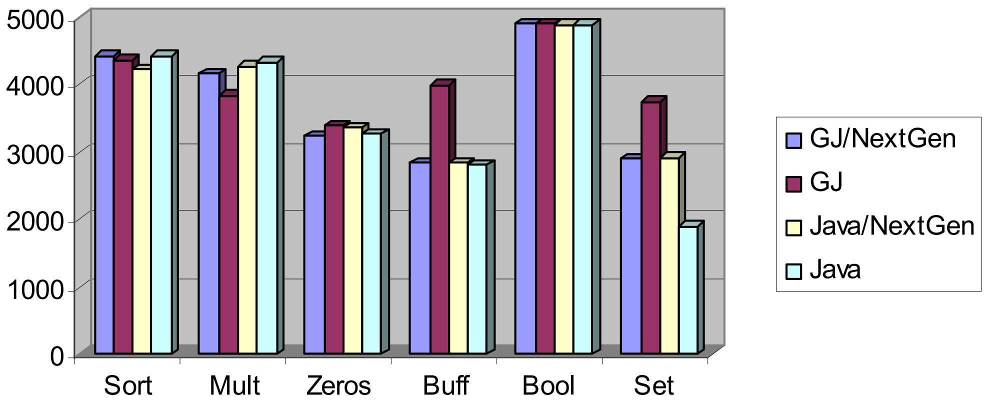


The performance overhead in **Sort** on the IBM JVM is anomalous; on all other JVMs tested, NextGen performs at least as well as GJ on this benchmark. Furthermore, the overhead for GJ on other JVMs is more significant than that for NextGen on the IBM JVM 1.3. This overhead may be attributed to an optimization contained on the IBM JVM that does not apply to the NextGen-generated bytecode. Alternatively, it may be nothing more than noise in the data.

The performance overhead in **Sort** may be attributed to the inclusion of a type dependent operation in a performance critical loop of the program. Since the evaluation of this operation will involve an extra snippet call each time, performance is degraded somewhat. However, in the more recent JVMs, such snippet method calls are dynamically inlined, which explains why the significant overhead seen on 1.2 is significantly reduced. Furthermore, it should be noted that even the performance difference for the **Sort** benchmark on 1.2 is dwarfed by many of the performance differences observed between 1.3 and 1.4 on these benchmarks.

In figs. 9 and 10, hybrid runs of GJ and Java are also presented for two JVMs, in which the code was compiled normally, but run using the NextGen class loader. These

Figure 10: Performance Comparison of NextGen and Java Class Loaders for IBM JDK 1.3 on Linux (in millseconds)



benchmarks isolate the overhead caused by the class loader alone. As the charts indicate, there is no overhead in using our class loader for non-parametric classes.

Chapter 6

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 type-dependent operations involving **`new`** expressions 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 class **`C<T>`**, where **`T`** is of **`class`** kind, and class **`D<S>`**, where **`S`** is of **`interface`** kind. Then **`C<S>`** is an invalid instantiation inside the body of **`D`**. In general, parameters of **`class`** and **`abstract class`** kind may not be instantiated with kind those of kind **`interface`**. 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.

Polymorphic methods without overriding 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 since 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 involve call site type parameters.

This design is appealing because it has almost no overhead in the common cases when (*i*) a polymorphic method requires no snippets and (*ii*) when 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 byte code 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 in [3], 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 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-variant 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 variable 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 7

RELATED WORK

The first generic Java compiler to support type dependent operations was an experimental compiler developed by Agesen, Freund, and Mitchell that 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 [1].

The generic type implementation which 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 mostly homogeneous approach to implementing genericity described in the NextGen design [6]. Since 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. Since 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.

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 completely different approach to implementing genericity from 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. 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.

Chapter 8

CONCLUSION

The existing implementation of the NextGen compiler has been described, and it has been shown how this implementation manages to avoid any serious performance penalties compared to the existing GJ and NextGen compilers. Furthermore, some ways in which this compiler could be extended to include additional features has been described, such as parametric methods with full support for run-time type operations, and classes with variable parent type.

Hopefully, this proof of concept will be persuasive in determining the future of the Java programming language with respect to parametric types. Since the addition of run-time type support is not only feasible, but attainable with little performance overhead, we anticipate its eventual inclusion into the language.

BIBLIOGRAPHY

[1] Agesen, O., Freund, S., Mitchell, J. Adding parameterized types to Java. In ACM Symposium on Object-Oriented Programming, Systems, Languages, and Applications, 1997.

[2] Bracha, G., Odersky, M., Stoutamire, D., Wadler, P. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *OOPSLA Proceedings,* October 1998.

[3] Cartwright., R., Steele, G., Compatible Genericity with Run-time Types for the Java Programming Language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, and Applications,* October 1998.

[4] Flatt, M., Krishnamurthi, S., Felleisen, M. Classes and Mixins. In Proceedings of ACM Symposium on Principles of Programming Languages, pp. 171--184, Jan. 1998.

[5] Gosling, J., Joy, B., Steele, G. The Java Language Specification. Addison-Wesseley. Reading, Mass. 1996.

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

[7] Kennedy, A., Syme, D., Design and implementation of generics for the .NET Common Language Runtime. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI), Snowbird, Utah, USA. June 2001.

[8] Myers, A., Bank, J., Liskov, B. Parameterized Types for Java. ACM POPL '97, Paris, France, January 1997.

[9] Odersky, M., Wadler, P., Pizza into Java: translating theory into practice. In Proc. 25th ACM Symposium on Principles of Programming Languages, 1997, 273-280.

[10] Agesen, Ole and David Detlefs. Mixed-mode Bytecode Execution. Sun Microsystems Technical Report SMLI TR-2000-87, June, 2000.

.

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.

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();
  }
}
```

Program After Compilation

```
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$$instanceOf$T(Object o);
  protected abstract WrappedIterator
    ListWrapper$$newWrappedIterator$T$(ListIterator itor);

  /**
   * Abstract snippet for new T[]. The two numbers after $T$ 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$T$1$0(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$$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$: This cast ensures that o is an instance of T or one of its
    //     subclasses. If T is parametric, we must check this via the
    //     instantiation interface because a subclass of T may not
    //     directly extend T! For example, Stack<String> extends Stack,
    //     not Vector<String>. But Stack<String> does implement
    //     Vector<String>$.
    // 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;
  }
```