Efficient Implementation of Run-time Generic Types for Java

Eric Allen, Robert Cartwright, Brian Stoler

Rice University

6100 Main St.

Houston, TX 77005, USA

{eallen, cork, bstoler}@cs.rice.edu

Abstract We describe an efficient compiler and run-time system for NEXTGEN, a compatible extension of the Java programming language supporting run-time generic types designed by Cartwright and Steele. The resulting system is comparable in performance with both standard Java and the GJ extension of Java, which does not support run-time generic types. Our implementation of NEXTGEN consists of a compiler extending the GJ compiler and a special class loader that constructs type instantiation classes on demand. The compiler relies on the 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 which are constructed on demand by the class loader. We include 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 overhead compared with ordinary Java and GJ.

Keywords: Java, GJ, NextGen, generic types, type dependent operation, JVM extensions, compatibility, specification, design, implementation, class loader, reflection, erasure, performance benchmarks, run-time, object-oriented programming, parametric polymorphism, polymorphic recursion, PolyJ, C#

Introduction

One of the most common criticisms of the Java programming language is the 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 expressed in the language. Moreover, generic type declarations enable the type checker to analyze these abstractions and perform far more precise static type

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35672-3_13](https://doi.org/10.1007/978-0-387-35672-3_13)

J. Gibbons et al. (eds.), *Generic Programming*

© IFIP International Federation for Information Processing 2003

checking than is possible in a simply typed language such as Java [7]. 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 to convert values of the bounding type 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 GJ source language and compiler [3] have shown, it is possible to compile Java with generic types into bytecode for the existing JVM. However, the GJ compiler imposes significant restrictions on the use of generic types because it relies on type erasure to implement genericity. In particular, it forbids all program operations that depend on run-time generic type information. The prohibited operations include:

- parametric casts,
- parametric `instanceof` tests¹,
- parametric catch operations, and
- new operations of “naked” parametric type such as `new T( )` and `new T[ ]`.

We call such operations *type dependent*. In addition, GJ prohibits per-type-instantiation of static fields; static fields in a generic class are shared by all instantiations of the generic class.

The GJ compiler does not support type dependent operations because it relies on type erasure to map generic operations into ordinary Java bytecode. In essence, GJ implements 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. But at the byte code level, the generic structure of the program has been erased.

¹GJ supports parametric casts and `instanceof` tests provided the parametric information in the operation is implied by context. In such cases, the parametric cast or `instanceof` test can be implemented by their type erasures.

Cartwright and Steele's NEXTGEN [4] is a more ambitious extension of Java that is based on essentially the same source language as GJ. NEXTGEN overcomes the limitations of GJ by introducing a separate Java class for each distinct instantiation of a generic type; all generic 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. For these reasons, we believe that NEXTGEN is an appealing prototype for a future extension of the Java language standard.

1. Design Fundamentals

The NEXTGEN formulation of generic types for Java is an implementation of the same source language as GJ, albeit with fewer restrictions on program syntax. In fact, NEXTGEN and GJ were designed in concert with one another [3, 4] so that NEXTGEN would be a graceful extension of GJ. We call this common source language Generic Java. In essence, Generic Java is ordinary Java (JDK 1.3/1.4) generalized to allow class and method definitions to be parameterized by types.

1.1. Generic Classes

In Generic Java, class definitions may be parameterized by type variables and program text may use generic types in place of conventional types. A *generic type* consists of either a type variable or an application of a generic class/interface name to type arguments that may also be generic. Specifically, in a class definition (§8.1 of the JLS [6]), the syntax for the class name appearing in the header of a class definition is generalized from

Identifier

to

Identifier {<TypeParameters>}

where

TypeParameters	↦	TypeParm ₁
		TypeParm ₁ , TypeParameters
TypeParm	↦	TypeVar { TypeBound }
TypeBound	↦	extends ClassType
		implements InterfaceType
TypeVar	↦	Identifier

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

```
class Vector<T>
```

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

$$\text{ReferenceType} \mapsto \text{ClassOrInterfaceType} \mid \text{ArrayType}$$

to

$$\begin{array}{ll} \text{ReferenceType} & \mapsto \text{ClassOrInterfaceType} \\ & \mid \text{ArrayType} \mid \text{TypeVar} \\ \text{TypeVar} & \mapsto \text{Identifier} \\ \text{ClassOrInterfaceType} & \mapsto \text{ClassOrInterface} \{ \langle \text{TypeParameters} \rangle \} \\ \text{ClassOrInterface} & \mapsto \text{Identifier} \\ & \mid \text{ClassOrInterfaceType} . \text{Identifier} \end{array}$$

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

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

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

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

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

where `A` is the element type of the list and `B` is a singleton² ordering class for `A`. Static members of a generic class create “holes” in the scope of all the enclosing type abstractions. A static nested class can be generic but all of the type variables of the class must be explicitly introduced in the definition of the class.

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

²A *singleton* class is a class with only one instance. Classes with no fields can generally be implemented as singletons.

1.2. Polymorphic Methods

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

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

where *Type* can be *void* as well as a conventional type. The scope of the type variables introduced in the type parameter list (*TypeParameters* above) is the header and body of the method. When a polymorphic method is invoked, no type instantiation information is required in most cases. For most polymorphic method applications, Generic Java can infer the values of the type arguments from the types of the argument values in the invocation³. Generic Java also provides a syntax for explicitly binding the type arguments for a polymorphic method invocation, but none of the current compilers (GJ, JSR-14⁴, and NEXTGEN) support this syntax yet.

1.3. The GJ Implementation Scheme

The GJ implementation scheme developed by Odersky and Wadler [3, 10] supports Generic Java 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. Similarly, for each polymorphic method *m*(*T*), GJ generates a single erased method *m*.

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 μ appearing in a context with erased type η that is not a supertype of μ , GJ automatically generates a cast to type η .

1.4. Implications of Type Erasure in GJ

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

³The GJ compiler implements more general inference rules that treat the value *null* as a special case.

⁴Sun Microsystems officially proposed adding generics to the Java language in JSR-14 [12]. Sun bought the rights to the GJ compiler and has released an update of the GJ compiler called the JSR-14 compiler for experimental use by the Java user community.

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

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

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

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

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

which erases to the base class:

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

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

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

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

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

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

1.5. Restrictions on Generic Java Imposed by GJ

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

1.6. The NEXTGEN Implementation Scheme

The NEXTGEN implementation of Generic Java eliminates the restrictions on type dependent operations imposed by GJ. In addition, the implementation architecture of NEXTGEN can support several natural extensions to Generic Java, including per-type static fields⁵ in generic classes and interfaces, co-variant subtyping of generic classes, and mixins. Because these features are not part of the existing Generic Java language, we will elaborate upon them only briefly, in Section 5.

2. NEXTGEN Architecture

NEXTGEN enhances the GJ implementation scheme by making the erased base class *C* abstract and extending *C* by classes representing the various instantiations of the generic class *C*(*T*), *e.g.*, *C*(*Integer*), that occur during the execution of a given program. These subclasses are called instantiation classes. Each instantiation class *C*(*E*) includes forwarding constructors for the constructors of *C* and code for the type dependent

⁵Since a per-type static field is attached to a specific instantiation of a generic class, the definition of such a field does not create a “hole” in the scope of enclosing type abstractions.

Figure 2. Illegal class hierarchy in naive JVM class representation

operations $C\langle E \rangle$. In the base class C , the type dependent operations of $C\langle T \rangle$ are replaced by calls on synthesized abstract methods called snippet methods [4]. These snippet methods are overridden by appropriate type specific code in each instantiation class $C\langle E \rangle$ extending C . The content of these snippet methods in the instantiation classes is discussed later in this section.

2.1. Modeling Generic Types in a Class Hierarchy

The actual implementation of instantiation classes is a bit more complex than the informal description given above. Figure 1 shows the hierarchy of Java classes used to implement the generic type $Vector\langle T \rangle$ and the instantiations $Vector\langle Integer \rangle$ and $Vector\langle String \rangle$.

When one generic class extends another, the simple JVM class hierarchy given in Figure 1 cannot represent the necessary subtyping relationships. For example, consider a generic class $Stack\langle T \rangle$ that extends a generic class $Vector\langle T \rangle$. Any instantiation $Stack\langle E \rangle$ of $Stack\langle T \rangle$ must inherit code from the base class $Stack$ which inherits code from the base class $Vector$. In addition, the type $Stack\langle E \rangle$ must be a subtype of $Vector\langle E \rangle$. Hence, the instantiation class for $Stack\langle E \rangle$ must be a subclass

Figure 3. Simple parametric type hierarchy and its JVM class representation

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.

Fortunately, Cartwright and Steele showed how we can exploit multiple interface inheritance to solve this problem [4]. The Java type corresponding to a class instantiation `C(E)` can be represented by an empty instantiation interface `C(E)$` which is implemented by the class `C(E)`. The `$` at the end of the interface name distinguishes it from the name of corresponding instantiation class and the names of other classes or interfaces (assuming source programs follow the convention of never using `$` in identifiers). Since a Java class can implement an interface (actually an unlimited number of them) as well as extend a class, the multiple inheritance problem disappears. Also, since these interfaces are empty, their construction does not appreciably affect program code size. Figure 3 represents the same type structure as Figure 2 while conforming to the restriction of single class inheritance.

The following rules precisely describe how the NEXTGEN implementation translates generic classes to ordinary Java classes. For each generic class `C(T)`:

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

Generic interfaces are processed similarly, but only steps three and four above are applicable.

The most interesting parts in this process are steps four and five. One might think that the compiler could determine an upper bound U on the set of possible generic types in a program and generate class files for each instantiation in U ⁶. However, early in the process of building a compiler for NEXTGEN, we discovered that the set of all possible generic types across all possible program executions is infinite for some programs. These infinite sets of instantiations are possible because Generic Java permits cycles in the type application graph, *i.e.*, a generic class $C\langle T \rangle$ may refer to non-ground type-applications of itself (or type application chains leading to itself) other than $C\langle T \rangle$. For example, consider the following parametric class:

⁶Cartwright and Steele proposed such a scheme in [4], but it does not handle cycles in the type application graph which is allowed in Generic 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 $C\langle T \rangle$ that reads a sequence of integer values from the console specifying the arguments for calls on the method `nest` for a receiver object of type $C\langle \text{String} \rangle$. Clearly, the set of possible instantiations across all possible input sequences is infinite.

We solved this problem by deferring the instantiation of generic classes until 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. A template class file looks exactly like a class file for a corresponding instantiation class except that the constant pool may contain some references to type variables. The class loader replaces these references (using string substitution) to form instantiation classes. To reduce the overhead of loading instantiation classes on demand, the customized class loader maintains a cache of the template class files that have been read already.

In the case of user-defined generic interfaces, the naive translation shown in Figures 1 and 2 suffices; no supplementary classes or interfaces are required because Java supports multiple interface inheritance.

2.2. Snippet Methods

As mentioned above, expressions involving type dependent operations are replaced with calls to abstract snippet methods, which are overridden in each instantiation class. The snippet methods in each instantiation class $C\langle E \rangle$ must perform the type dependent operations determined by the types E . For new operations and catch operations, the generation of the appropriate type dependent code is straightforward. But a small complication arises in the case of casts and `instanceof` tests on $C\langle E \rangle$. In a naive implementation, the body of a snippet method corresponding to a cast or `instanceof` test of type $C\langle E \rangle$ would simply perform the operation on its argument using the instantiation class for $C\langle E \rangle$. But this implementation fails in some cases because of subtyping: the subclasses of $C\langle E \rangle$ are not necessarily subtypes of the instantiation class $C\langle E \rangle$. (Recall the example depicted in Figure 3.)

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

2.3. Extensions of Generic Classes

If a generic class D extends another generic class C where C is not fully instantiated⁷, a `NEXTGEN` compiler must include concrete snippets for the type dependent operations of C in instantiation classes for D . These added snippets are necessary because the base class for C is the superclass of the base class for D . The requisite snippets are identical to the snippets in the template class for C , specialized with any type bindings established in the definition of D .

2.4. Polymorphic Methods

At first glance, polymorphic methods look easy to implement on top of generic classes: they can be translated to generic inner classes containing a single `execute` method [10]. Each invocation of a polymorphic method can create a generic instance of the associated inner class and invoke the `execute` method on the arguments to the polymorphic method call. Unfortunately, this translation does not work in general because polymorphic methods can be overridden in subclasses but inner classes cannot. In addition, the overhead of creating a new object on every invocation of a polymorphic operation could adversely impact program performance if polymorphic method calls are frequently executed.

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

Our solution to this problem relies on using a heterogeneous translation [10] for polymorphic methods within generic classes. In other words, if the polymorphic method is defined within a generic class, we create a separate copy of the method definition in each instantiation class for the containing generic class. Hence, each receiver class of a polymor-

⁷If C is fully instantiated then D simply extends the instantiation class representing C .

phic method call has a distinct implementation of the method, thereby accommodating method overriding.

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

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

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

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

ing to the particular instantiation at the call site can be passed simply as an extra parameter to the method.

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

2.5. Design Complications

The preceding description of the NEXTGEN architecture neglects three subtle problems that arise in the context of the Java run-time environment: (i) access to private types passed across package boundaries, (ii) access to private generic class constructors from the corresponding instantiation class constructors, (iii) the compatible generalization of the Java collection classes to generic form⁸.

2.5.1 Cross-Package Instantiation. The outline of the NEXTGEN architecture given above does not specify where the instantiation classes of a generic class $C\langle T \rangle$ are placed in the Java name space. Of course, the natural place to put them is in the same package as the base class C , which is what NEXTGEN does.⁹ But this placement raises an interesting problem when a private type is “passed” across a package boundary [10].

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

The simplest solution to the problem of cross-package instantiation is to automatically widen a private class to public visibility if it is passed as a type argument in the instantiation of a generic type in another package. Although this approach raises security concerns, such widening has a precedent in Java. When an inner class refers to the private members of the enclosing class, the Java compiler widens the visibility of these private members by generating getters and setters with package visibility [6]. Although more secure (and expensive) implementations of inner classes are possible, the Java language designers chose to sacrifice some

⁸The GJ and JSR-14 compiler systems include a version of the Java libraries containing generic type signatures for the Java collection classes. The bytecode for the classes is unchanged.

⁹In any other package, private classes from the package containing C are inaccessible to the instantiation classes.

visibility protection for the sake of performance. This loss of visibility security has not been a significant issue in practice because most Java applications are assembled from trusted components.

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

One solution, as laid out in [4], is for the client class accessing an instantiation $C\langle E \rangle$ of a generic class $C\langle T \rangle$ to pass a snippet environment to a synthesized initializer method in the instantiation class. This environment is an object containing all of the snippets in the instantiation $C\langle E \rangle$. The snippet methods defined in $C\langle E \rangle$ simply delegate snippet calls to the snippet environment. But this solution requires an initialization protocol for instantiation classes that is tedious to manage in the context of separate class compilation. Type arguments can be passed from one generic class to another, implying that the composition of a snippet environment for a given generic type instantiation depends on all the classes reachable from the caller in the type application call graph. The protocol described in [4] assumes that the initialization classes are generated statically, which, as we observed earlier, cannot be done in the presence of polymorphic recursion. This protocol can be patched to load snippet environment classes dynamically from template class files. But any changes to a generic class can force the recompilation of all generic classes that can reach the changed class in the type application call graph.

A better solution to the private type visibility problem is for the class loader to construct a separate singleton class for every snippet where the mangled name of the class identifies the specific operation implemented by the snippet. The NEXTGEN compiler already uses a similar name mangling scheme to name snippet methods in instantiation classes, eliminating the possibility of generating multiple snippet methods that implement exactly the same operation. In essence, the class-per-snippet scheme replaces a single snippet environment containing many snippets by many snippet environments each containing a single method. The advantage of this scheme is that name mangling can uniquely specify what operation must be implemented, enabling the class loader to generate the requisite public snippet classes on demand (without reading snippet environment class files) and place them in the same package as the type argument to the snippet. The compiler does not have to keep track of the type application call graph because snippets are dynamically generated as the graph is traversed during program execution. To prevent

unauthorized access to private classes via a mangled snippet class name, the class loader only resolves references to such a class if (i) the accessing class is in within the same package as all of the types embedded in mangled name or (ii) the accessing class is an instantiation class where each “inaccessible” type in the mangled snippet class name appears as a type argument in the instantiation class. In essence, type arguments are “capabilities” passed from one generic class to another.

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

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

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

2.5.3 Extending the Java Collection Classes. One of the most obvious applications of generic types for Java is the definition of generic versions of the Java collection classes. GJ supports such an extension of the Java libraries by simply associating generic signatures with the existing JDK collection classes. To accommodate interoperability with legacy code, GJ allows breaches in the type system of Generic Java. In particular, GJ accepts programs that use erased types in source program text and supports automatic conversion between generic types and their erased counterparts. Using this mechanism, Generic Java programs can interoperate with legacy code that uses erased versions of

generic classes, *e.g.*, the collection classes in the existing JDK 1.3 and 1.4 libraries. But this interoperability is bought at the price of breaking the soundness of polymorphic type-checking¹⁰.

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

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

Because of the distinction between objects of parametric type and objects of raw type, the integration of legacy code and NEXTGEN requires more care than the integration of legacy code and GJ. But the extra care has a major payoff: the soundness of polymorphic type-checking is preserved¹¹.

3. NEXTGEN Implementation

The NEXTGEN compiler is implemented as an extension of the GJ compiler written by Martin Odersky. The GJ compiler is organized as a series of passes that transform a parsed AST to byte code. We have extended this compiler to support NEXTGEN by inserting an extra pass that detects type dependent operations in the code, encapsulates them

¹⁰The type system supported by the GJ compiler includes raw (erased) types. When an object of raw type is used in a context requiring a parametric type, the GJ type-checker flags an unchecked operation error, indicating that the program violates polymorphic type-checking rules. GJ still compiles the program to valid byte code, but the casts inserted by the GJ compiler can fail at run-time.

¹¹To preserve type soundness, raw types must be treated more carefully than they are in GJ. In particular, the raw type C corresponding to a generic type $C\langle T \rangle$ must be interpreted as the existential type $\exists T\ C\langle T \rangle$. Hence, any operation with T as a result type yields type **Object**. Similarly, any method in a generic class with an argument whose type depends on T is illegal.

as snippet methods in the enclosing generic classes, and generates template classes and interfaces for each generic class. The names assigned to these snippets are guaranteed not to clash with identifiers in the source program nor with the synthesized names for inner classes, because they include the character sequence `$$`, which by convention never appears in Java source code or the mangled names of inner classes. We have also modified the GJ code generation pass to accept these newly generated class names even though there is no corresponding class definition.

The added pass destructively modifies the AST for generic classes by adding the requisite abstract snippet methods and replacing dependent operations with snippet invocations. It also generates the template classes that schematically define the instantiation classes corresponding to generic classes. The template classes look like ordinary Java classes except that their constant pools may contain references to type parameters.

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

Both the NEXTGEN compiler and class loader rely on a name-mangling scheme to generate ordinary Java class names for instantiation classes and interfaces.

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

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

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

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

is encoded as:

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

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

3.1. The NEXTGEN Class Loader

When a NEXTGEN class refers to a generic type within a type dependent operation, the corresponding class file refers to a mangled name encoding the generic type. Since we defer the generation of instantiation classes and interfaces until run-time, no actual class file exists for a mangled name encoding a generic type. Our custom class loader intercepts requests to load classes (and interfaces) with mangled names and uses the corresponding template class file to generate the requested class (or interface). A template class file looks exactly like a conventional class file except that the constant pool may contain references to unbound type variables. The references to unbound type variables are written in de Bruijn notation: the strings $\{0\}$, $\{1\}$, $\dots$, refer to the first, second, $\dots$, type variables, respectively. Since the characters $\{$ and $\}$ cannot appear in Java class names, type variable references can be embedded in the middle of mangled class names in the constant pool.

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

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

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

Figure 4. Performance results for Sun 1.3 client (in milliseconds)

3.2. Performance

The NEXTGEN compiler is still under active development. Some aspects of genericity are not yet fully supported, most notably polymorphic methods. In addition, NEXTGEN provides a framework for supporting a richer genericity facility than what is included in Generic Java. Because no established benchmark suite for Generic Java exists, we had to construct our own benchmark suite to measure the performance of NEXTGEN. On existing benchmark suites for ordinary Java like JavaSpecMark [1], the performance of NEXTGEN is identical to that of the GJ and JSR-14 compilers, because they all generate the same class files. Our benchmark suite consists of the following programs, which all involve generic types:

- **Sort:** An implementation of the quicksort algorithm on generically typed linked lists, where quicksort is parameterized by the ordering relation for the sort. This benchmark consists of 769 lines of code in 13 classes. 7 of these classes make heavy use of generics.
- **Mult:** A visitor over generically typed binary trees of integers that multiplies the values of the nodes. This benchmark consists of 428

Figure 5. Performance results for Sun 1.3 server (in milliseconds)

lines of code in 16 classes. 7 of these classes make heavy use of generics.

- **Zeros:** A visitor over generically typed binary trees that determines whether there is any child-parent pair in which both hold the value 0. This benchmark consists of 552 lines of code and 14 classes. 8 of these classes make heavy use of generics.
- **Buff:** An implementation of `java.util.Iterator` over a `BufferedReader`. This benchmark constructs a large, buffered `StringReader`, and then iterates over the elements. This benchmark consists of 305 lines of code in 7 classes. 2 of these classes make heavy use of generics.
- **Bool:** A simplifier of Boolean expressions. This program reads a large number of Boolean expressions from a file, parses them, and simplifies them. The simplification process is organized as a series of passes, each implemented by a generically typed visitor. This benchmark consists of 730 lines of code in 25 classes. 7 of these classes make heavy use of generics.

Figure 6. Performance results for Sun 1.4 client (in milliseconds)

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

The benchmarks were written in Generic Java specifically to take advantage of the added type checking provided by Generic Java. To facilitate comparison with the GJ compiler and the JSR-14 update of the GJ compiler, all of the benchmarks conform to the restrictions imposed by the GJ implementation of Generic Java¹².

The source code for each benchmark was manually translated to equivalent Java source code. Manual modification was necessary because the source transformation performed by the GJ compiler does not necessarily yield valid ordinary Java code. The GJ compiler performs its own

¹²Some type dependent operations in NextGen are not type dependent in GJ, because GJ erases all parametric type information. In particular, all new operations on generic types are type dependent in NextGen but not in GJ.

Figure 7. Performance results for Sun 1.4 server (in milliseconds)

code generation for this reason¹³. Nevertheless, this manual modification consisted merely of inserting casts and bridge methods as necessary; it had no effect on the number of classes or lines of code. The original and converted source code were both compiled using the JSR-14 compiler. The NEXTGEN compiler was applied to exactly the same source code as the JSR-14 compiler.

The results of these benchmarks for Java, GJ, and NEXTGEN under five separate JVMs are illustrated in Figures 5 to 8. These results were obtained by running each benchmark twenty-one times, for each JVM listed, on a 2.0 GHz Pentium 4 with 512 MB RAM running Red Hat Linux 7.2. Because the results of the first run on for each JVM/compiler combination exhibited significant variance, the results of the first run were uniformly dropped. We attribute this variance to the overhead of JVM startup and initial JIT (“just-in-time”) compilation of the code, neither of which is relevant to what our experiment is intended to mea-

¹³The bridge methods generated by GJ may rely on the result type for static overloading resolution; the JVM supports this generalization of Java overloading.

Figure 8. Performance results for IBM 1.3 (in milliseconds)

sure. Once the first run was dropped, the variance in the duration of the individual runs for each benchmark was less than 10%.

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

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

The small overhead in some of the benchmarks can be explained by considering what costs are incurred by keeping the run-time type information. Once an instantiation of a template class is loaded into memory, the only overhead of genericity is the extra method call involved in invoking 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.¹⁴

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

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

4. Future Extensions

4.1. Better support for polymorphic methods

Polymorphic methods that require snippets depending on method type arguments are not yet implemented in NEXTGEN. Passing explicit type arguments to polymorphic methods is not yet supported either. In our experience writing Generic Java, neither of these extensions is likely to be used very frequently but they are occasionally important. We are currently extending NEXTGEN to support **static** polymorphic methods in full generality. We are deferring the implementation of dynamic polymorphic methods in full generality because cooperation from the JIT compiler is the key to obtaining good performance.

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

4.2. Type Parameter Kinds

In new operations on a naked type parameter T , it makes no sense to instantiate T as an interface or abstract class. This constraint should be part of the visible signature of the generic class or polymorphic method binding the type parameter T .

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 kind **interface**, unless there were an **extends** clause (in which case it would be assumed to be of kind **abstract class**). Notice that the type checker, when checking instantiations of type parameters, must not only check that new operations are only applied to types of kind **class**, but also that the types of instantiations of generic classes are of the correct kind. In general, parameters of kind **class** may not be instantiated by types of kind **abstract class** or **interface**. Similarly, parameters of kind **abstract class** cannot be instantiated by types of kind **interface**. 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 files, template files, or to the augmented class loader, are necessary.

4.3. Constructor Declarations for Naked new Operations

NEXTGEN currently restricts naked new operations to 0-arity. There is no reason for this restriction other than Generic Java syntax, which makes no provision for specifying what constructor signatures a generic type argument must support. This omission could easily be remedied by adding an optional suffix to the declaration of type parameters in generic classes (TypeParm in §2.1) of the form

with ConDefList

where

$$\begin{aligned}\text{ConDefList} &\mapsto \text{ConDef} \mid \text{ConDef}, \text{ConDefList} \\ \text{ConDef} &\mapsto \text{TypeVar} (\{ \text{ArgumentList} \})\end{aligned}$$

For example, the class *C* could take a type parameter *T* with a constructor *T*(int *i*) as follows:

```
class C<T with T(int i)> { ... } .
```

4.4. Mixins as Classes with Variable Parent Type

The Generic Java language does not permit the occurrence of a naked type variable in the *extends* or *implements* clauses of a class definition. An extension to the language allowing 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. The functionality for each superclass can be added to the target class by applying a separate mixin. By allowing for the occurrence of a type variable in the *extends* and *implements* clauses of a class, NEXTGEN would provide the programmer with a way to bind the parent class of an object when its generic type is instantiated.

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 template class files for the mixin classes, a process strikingly similar to the existing generic class instantiation mechanism in the NEXTGEN class loader. Consequently, we expect extension of the NEXTGEN class loader to support variable parent type to be straightforward. In contrast, extending the NEXTGEN type system and type checker to handle mixin types is a non-trivial endeavor [7].

4.5. Covariant Subtyping of Type Parameters

A simple but potentially useful extension of NEXTGEN, described in [4], 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 invariant 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 Generic Java (as implemented in GJ and NEXTGEN) already supports covariance in method return types.

5. Related Work

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

The heterogeneous approach to implementing genericity has two disadvantages. First, the heterogeneous expansion of every generic instantiation can produce an exponential blow-up in the size of the executable code and can seriously degrade program performance. Second, the heterogeneous approach does not provide a common superclass (the base class in NEXTGEN) for all instantiations of a particular generic class, preventing the use of “raw” types, which are particularly useful in the context of integrating generic code with legacy non-generic code.

Viroli and Natali have proposed supporting run-time generic type information by embedding the information in an extra field attached to objects of generic type and using reflection to implement type dependent operations [13]. This approach has two obvious disadvantages. First, every object of generic type requires an extra word of memory. For small objects such as list nodes, this space penalty can be significant (25% for an object with two fields assuming a two word header). Second, using reflection to implement type dependent operations is slow. Viroli and Natali argue that the overhead of reflection can largely be eliminated by performing some load-time optimizations to streamline the implementation of type dependent operations. They use synthetic micro-benchmarks to compare the performance of their implementation scheme with the original NEXTGEN implementation scheme described by Cartwright and Steele [4]. We believe their results are misleading because they presume no method inlining for NEXTGEN while presuming it for their implementation¹⁵. Moreover, they model the performance of

¹⁵In NextGen, the inlining of snippets can only occur if the code containing the type dependent operation is specialized to a particular type, but such specialization is part of the standard repertoire of dynamic optimization techniques.

the NEXTGEN implementation scheme described in [4] where a separate instantiation class and interface file must be read from disk for each class instantiation. Their Generic Java implementation is not yet complete, but we are eager to see how well it performs on our benchmark Generic Java programs.

The generic type implementation most closely resembling NEXTGEN is the extension of the .NET common runtime by Kennedy and Syme to support generic types in C# [8]. They follow the same, mostly homogeneous, approach to implementing genericity described in the NEXTGEN design [7]. 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 cycles in the type application graph, 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 is. 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 [9]. 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.

6. Conclusion

Our NEXTGEN implementation demonstrates that it is possible to support run-time generic types on top of the existing Java Virtual Machine while maintaining compatibility with legacy code. Furthermore, our performance testing shows that this support can be provided without significant overhead. We hope this proof-of-concept will eventually lead to the inclusion of run-time generic types in the Java programming language.

References

- [1] O. Agesen, S. Freund, and J. Mitchell. *Adding parameterized types to Java*. OOPSLA 1997.
- [2] E. Allen, R. Cartwright. *The Case for Run-time Types in Generic Java*. Principles and Practice of Programming in Java, June 2002.
- [3] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. *Making the future safe for the past: adding genericity to the Java programming language*. OOPSLA '98, October 1998.
- [4] R. Cartwright, G. Steele. *Compatible genericity with run-time types for the Java programming language*. OOPSLA '98, October 1998.
- [5] M. Flatt, S. Krishnamurthi, M. Felleisen. *A Programmer's Reduction Semantics for Classes and Mixins*. Formal Syntax and Semantics of Java, volume 1523, June 1999.
- [6] J. Gosling, B. Joy, G. Steele. *The Java Language Specification*. Addison-Wesley. Reading, Mass. 1996.
- [7] A. Igarashi, B. Pierce, P. Wadler. *Featherweight Java: A minimal core calculus for Java and GJ*. OOPSLA '99, November 1999.
- [8] A. Kennedy and D. Syme. *Design and implementation of generics for the .NET Common Language Runtime*. PLDI 2001.
- [9] A. Myers, J. Bank, B. Liskov. *Parameterized Types for Java*. ACM POPL 1997.
- [10] M. Odersky and P. Wadler. *Pizza into Java: translating theory into practice*. POPL 1997.
- [11] O. Agesen and D. Detlefs. *Mixed-mode Bytecode Execution*. Sun Microsystems Technical Report SMLI TR-2000-87, June, 2000.
- [12] Sun Microsystems, Inc. *JSR 14: Add Generic Types To The Java Programming Language*. Available at <http://www.jcp.org/jsr/detail/14.jsp>.
- [13] Viroli, M. and A. Natali. *Parametric Polymorphism in Java: an Approach to Translation Based on Reflective Features*. OOPSLA 2000.