Efficient Implementation of Run-time Generic Types for Java

Eric Allen, Robert Cartwright, Brian Stoler
Rice University
6100 Main St.
Houston TX 77005
{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.

Key words: 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#.

1. 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 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 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 [2] 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** tests[1],
- 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.

NextGen is a more ambitious extension of Java, based on the same source language as GJ, developed by Cartwright and Steele [3] that 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 could serve as the basis for an important step forward in the evolution of the Java programming language.

2. DESIGN FUNDAMENTALS

The NextGen formulation of generic 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 [2, 3] 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 1.2/1.3/1.4) generalized to allow class and method definitions to be parameterized by types.

2.1 Generic Classes

In Generic Java, class definitions may be parameterized by type variables and program text may use generic types—consisting of applications of parameterized class names to type arguments—in place of conventional types. Specifically, in a class definition (§8.1 of the JLS [5]), the syntax for the class name appearing in the header is generalized from

Identifier

to

Identifier { **<** *TypeParameters* **>** }
TypeParameters → *TypeParm* | *TypeParm* , *TypeParameters*
TypeParm → *TypeVar* { *TypeBound* }
TypeBound → ***extends*** *ClassType* | ***implements*** *InterfaceType*
TypeVar → *Identifier*

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

class Vector<T> .

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

ReferenceType → *ClassOrInterfaceType* | *ArrayType*

to

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

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

new *TypeVar* **(** { *ArgumentList* } **)** **.**

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

The scope of the type variables introduced 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 singleton[2] ordering class for **A**.

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.

2.2 Polymorphic Methods

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

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

where *Type* can be **void** as well as a conventional type. The scope of the type variables introduced in the type parameter list (*TypeParameters* above) is the header and body of the method. When a polymorphic method is invoked, no type instantiation information is required in most cases. For most polymorphic method applications, Generic Java can infer the values of the type arguments from the types of the argument values in the invocation.[3] Generic Java also provides a syntax for explicitly binding the type arguments for a polymorphic method invocation, but none of the current compilers (GJ, JSR-14[4], and NextGen) support this syntax yet.

2.3 The GJ Implementation Scheme

The GJ implementation scheme developed by Odersky and Wadler [2, 9] 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 τ appearing in a context with erased type σ that is not a supertype of τ, GJ automatically generates a cast to type σ.

2.4 Implications of Type Erasure in GJ

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

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

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

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

Now suppose that 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.

2.5 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.

2.6 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 fields 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.

3. 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 non-private constructors of `C` and code for the type dependent operations `C<E>`. In the base class `C`, the type dependent operations of `C<T>` are replaced by calls on synthesized abstract methods called *snippet methods* [3]. These snippet methods are overridden by appropriate type specific code in each instantiation class `C<E>` extending `C`. The content of these snippet methods in the instantiation classes is discussed later in this section.

3.1 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<T>` and the instantiations `Vector<Integer>` and `Vector<String>`.

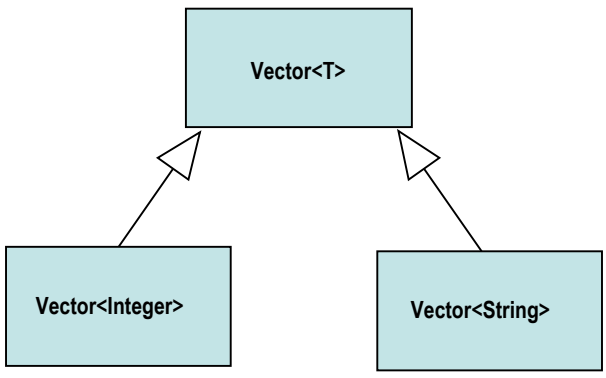


Figure 1. Naïve implementation of generic types over the existing Java class structure.

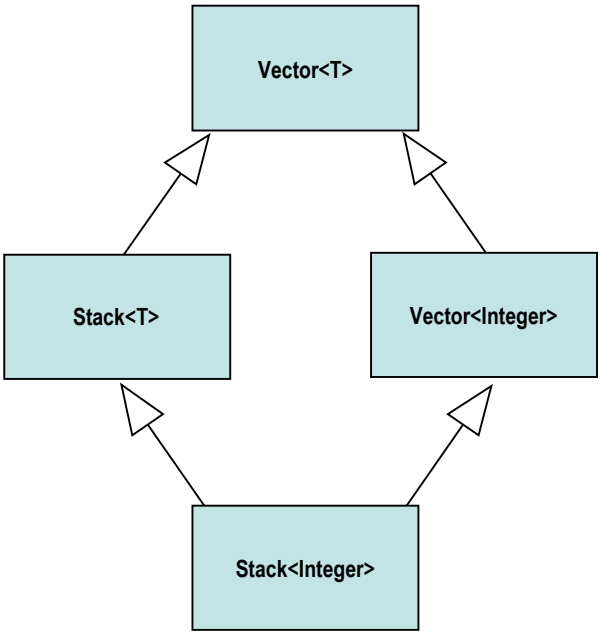


Figure 2. Illegal Class hierarchy in naive JVM Class Representation.

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<T>` that extends a generic class `Vector<T>`. Any instantiation `Stack<E>` of `Stack<T>` must inherit code from the base class `Stack` which inherits code from the base class `Vector`. In addition, the type `Stack<E>` must be a subtype of `Vector<E>`. Hence, the instantiation class for `Stack<E>` must be a subclass of two different superclasses: the base class `Stack` and the instantiation class for `Vector<E>`. This class hierarchy is illegal in Java because Java does not support multiple class inheritance. Figure 2 shows this illegal hierarchy.

Fortunately, Cartwright and Steele showed how we can exploit multiple *interface* inheritance to solve this problem[3]. 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

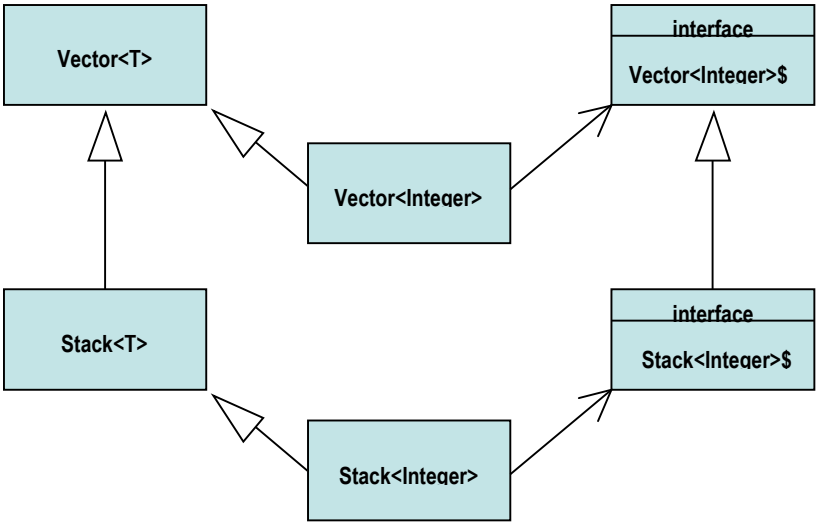


Figure 3. Simple Parametric Type Hierarchy and its JVM Class Representation.

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>`:

- Generate an abstract snippet method in `C<T>` for each application of a type dependent operation.
- Replace each such application with an application of the new snippet method, passing in the appropriate arguments.
- Erase all types in the transformed class `C<T>` to produce the base class `C` for `C<T>`.
- For every instantiation `C<E>` of `C<T>` encountered during program execution, generate an instantiation *interface* for `C<E>` and all superclasses and superinterfaces of `C<E>` in which any of the type parameters of `C<T>` occur.

- For every instantiation `C<E>` of `C<T>` encountered during program execution, generate an instantiation *class* for `C<E>` and all superclasses of `C<E>` in which any of the type parameters of `C<T>` occur.
- Insert the appropriate forwarding constructors and concrete snippet methods into each instantiation class `C<E>`. 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.

Much of the complexity of this process is a result of 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`.[5] However, early in the process of building a compiler for NextGen, we discovered that the set of all possible generic types across all possible program executions is infinite for some programs. These infinite sets of instantiations are possible because Generic Java permits *polymorphic recursion, i.e.*, a generic class `C<T>` may refer to non-ground type-applications of itself (or type application chains leading to itself) other than `C<T>`. 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 the console specifying the arguments for calls on the method nest for a receiver object of type `C<String>`. 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 naïve translation shown in Figures 1 and 2 suffices; no supplementary classes or interfaces are required because Java supports multiple interface inheritance.

3.2 Snippet Methods

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

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

3.3 Extensions of Generic Classes

If a generic class `D` extends another generic class `C` where `C` is not fully instantiated,[6] 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`.

3.4 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 [9]. 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 [9] 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 polymorphic method call has a distinct implementation of the method 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.

Caching can be used in polymorphic method bodies to reduce the overhead of loading snippet environments. A method body can maintain an inline cache consisting of the last type-argument class object and the corresponding snippet environment. If the current type-argument class object matches the cached one, then the cached snippet environment is the current one. In addition, the class loader *must* cache all classes that have been previously loaded (since a Java class can only be loaded once) and *may* cache template class files that have previously been read, eliminating the need to read any template class more than once.

This heterogeneous implementation of polymorphic methods is appealing because it has almost no overhead in the common cases (*i*) where a polymorphic method requires no snippets and (*ii*) where a polymorphic method requires only 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 special class object (a constant) representing the

list of type arguments at the call site; this parameter is ignored by the method code in the receiver. The heterogeneous translation of a polymorphic method in a generic receiver class enables the method body to directly implement (without snippets) all of the type dependent operations that depend only on the receiver class instantiation.

4. DESIGN COMPLICATIONS

The preceding description of the NextGen architecture neglects two subtle problems that arise in the context of the Java run-time environment: (*i*) access to private types passed across package boundaries and (*ii*) the compatible generalization of the Java collection classes to generic form.[7]

4.1 Cross-Package Instantiation

The outline of NextGen architecture given above does not specify where the instantiation classes of a generic class `C<T>` are placed in the Java name space. Of course, the simplest place to put them is in the same package as the base class `C`, which is what NextGen does. But this placement raises an interesting problem when a private type is "passed" across a package boundary [9].

Consider the case where a class `D` in package `Q` uses the instantiation `C<E>` of class `C<T>` in package `P` where `E` is private in `Q`. If the body of class `C<T>` contains type-dependent operations, then the snippet bodies generated for instantiation class `C<E>` will fail because they can not 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 [5]. Although more secure (and expensive) implementations of inner classes are possible, the Java language designers chose to sacrifice some visibility protection for the sake of performance. This loss of visibility security has not been a significant issue in practice because most Java applications are assembled from trusted components. For this reason, we have followed a similar strategy in addressing the visibility issues raised by generic class instantiation: the current NextGen compiler simply widens the visibility of private classes to public when necessary and generates a warning message to the programmer.

Nevertheless, it is possible to implement NextGen without compromising class visibility. One solution, as laid out in [3], is for the client accessing an instantiation `C<E>` of a generic class `C<T>` 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<E>`. The snippet methods defined in `C<E>` simply forward 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 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 [3] assumes that the initialization classes are generated statically, which, as we observed earlier, cannot be done in the presence of polymorphic recursion. Fortunately, this protocol can be patched to load snippet environment classes dynamically from template class files. Buy 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 simple alternative to snippet environments 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 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 these snippet classes, the class loader only resolves references to these classes within generic instantiation classes with type arguments that matches the embedded type names in the snippet class names.

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 class require an extra static method call for each snippet invocation.

4.2 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 interoperation with legacy code, GJ allows breaches in the type system of Generic Java. In particular, GJ accepts programs that use erased types in source program text and supports automatic conversion between generic types and their erased counterparts. Using this mechanism, Generic Java programs can interoperate with legacy code that uses erased versions of generic classes, *e.g.*, the collection classes in the existing JDK 1.3 and 1.4 libraries. But this interoperability is bought at the price of breaking the soundness of polymorphic type-checking.[8]

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 collections 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) collections classes. Given such a library, generic collections 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 collections classes would include methods to perform such conversions.

Because of the distinction between objects of parametric type 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 preserved.[9]

4.3 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 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 identifier 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.

4.4 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}**,…, refer to the first, second, …, 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*$$L*contents*$$R*suffix*** 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 occurrences of "." (period).

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

4.5 Performance

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 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 benchmarks consists of 305 lines of code in 7 classes. 2 of these classes make heavy use of generics.
- **Bool**: A simplifier of Boolean expressions. This program reads a large number of Boolean expressions from a file, parses them, and simplifies them. The simplification process is organized as a series of passes, each implemented by a generically typed visitor. This benchmark consists of 730 lines of code in 25 classes. 7 of these classes make heavy use of generics.
- **Set**: An implementation of generically typed multi-sets, and set-theoretic operations on them. This program constructs large multi-sets and compares them as they are built. This benchmark consists of 316 lines of code in 6 classes. 2 of these classes make heavy use of generics.

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

compiler, all of the benchmarks conform to the restrictions imposed by the GJ implementation of Generic Java.[10]

The source code for each benchmark was manually translated to equivalent Java source code. Manual modification was necessary because the source transformation performed by the GJ compiler does not necessarily yield valid ordinary Java code. The GJ compiler performs its own code generation for this reason.[11] Nevertheless, this manual modification consisted merely of inserting casts and bridge methods as necessary; it had no effect on the number of classes or lines of code. The original and converted source code were both compiled using the JSR-14 compiler. The NextGen compiler was applied to exactly the same source code as the JSR-14 compiler.

The results of these benchmarks for Java, GJ, and NextGen under five separate JVMs are illustrated in Figs. 4-8. 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 measure. Once the first run was dropped, the variance in the duration of the individual runs for each benchmark was less than 10%.

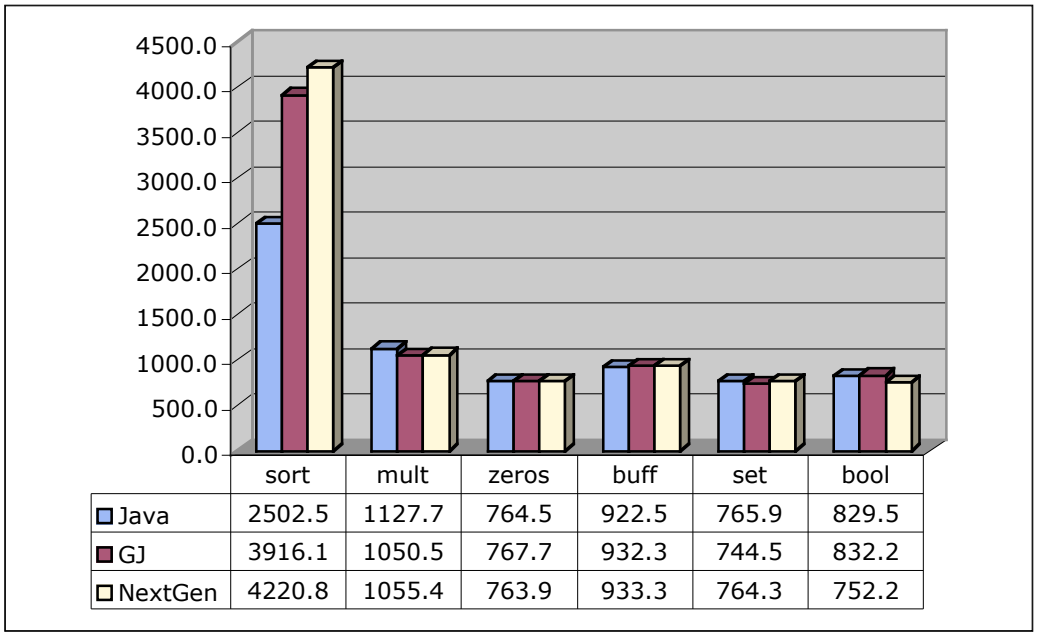
The results for the JSR-14 compiler also apply to the GJ compiler, because the class files generated by these compilers are functionally identical. The only differences are that (*i*) JSR-14 inserts an additional entry into the constant pool, and (*ii*) JSR-14 by default makes the class file version 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 the inclusion of run-time support for generic types does not add significant overhead, even for programs that make heavy use of it. In fact, even the small overhead that NextGen exhibits for some benchmarks is dwarfed by the significant range in performance results across JVMs.

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 added 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.[12]

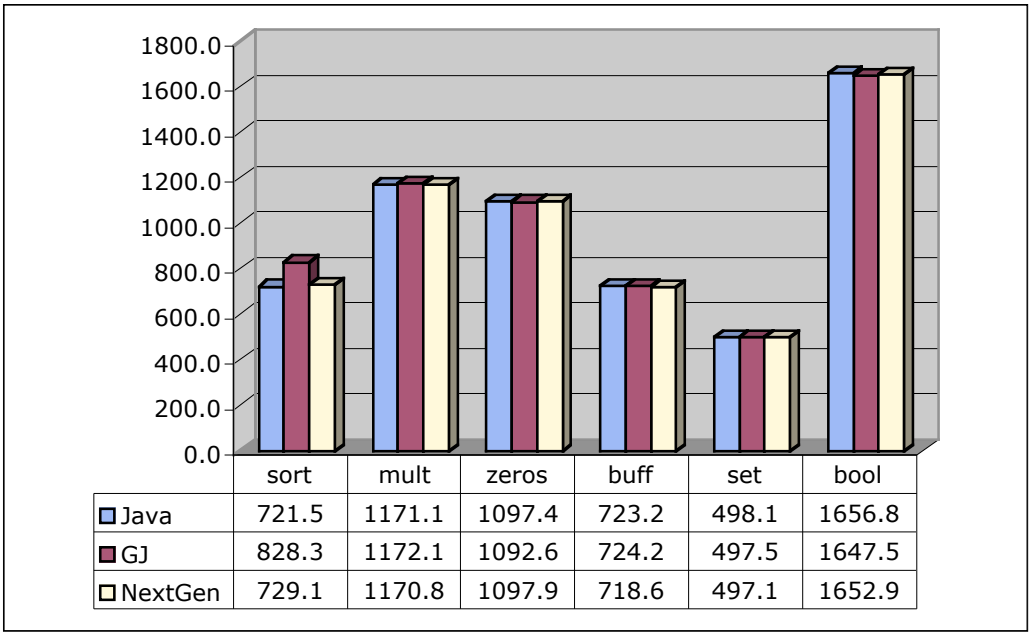
On advanced JVMs that perform dynamic code optimization, even type dependent operations incur little overhead because many of the snippet operations are inlined, eliminating the extra snippet call. If NextGen were adopted as the Java standard, we anticipate that dynamic code optimization would be tuned to eliminate essentially all snippet call overhead.

Our benchmark 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 JVM's for any of the benchmarks suggests that code optimization sensitive to the performance demands of NextGen could completely eliminate the overhead.



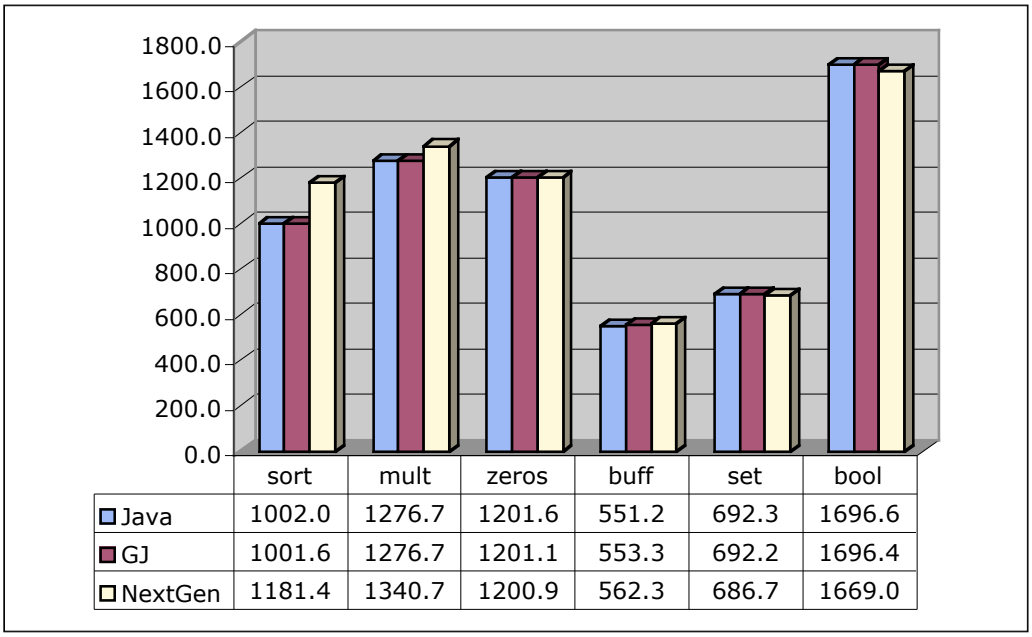
	sort	mult	zeros	buff	set	bool
Java	2502.5	1127.7	764.5	922.5	765.9	829.5
GJ	3916.1	1050.5	767.7	932.3	744.5	832.2
NextGen	4220.8	1055.4	763.9	933.3	764.3	752.2

Figure 4. Performance Results for IBM 1.3 (in milliseconds).



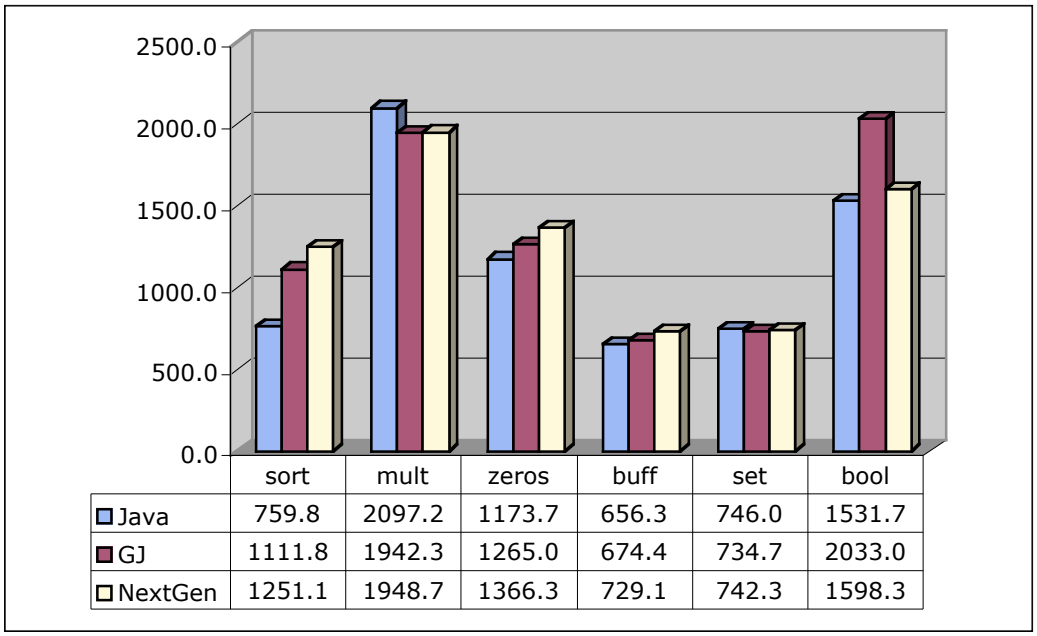
	sort	mult	zeros	buff	set	bool
Java	721.5	1171.1	1097.4	723.2	498.1	1656.8
GJ	828.3	1172.1	1092.6	724.2	497.5	1647.5
NextGen	729.1	1170.8	1097.9	718.6	497.1	1652.9

Figure 5. Performance Results for Sun 1.3 Server (in milliseconds).



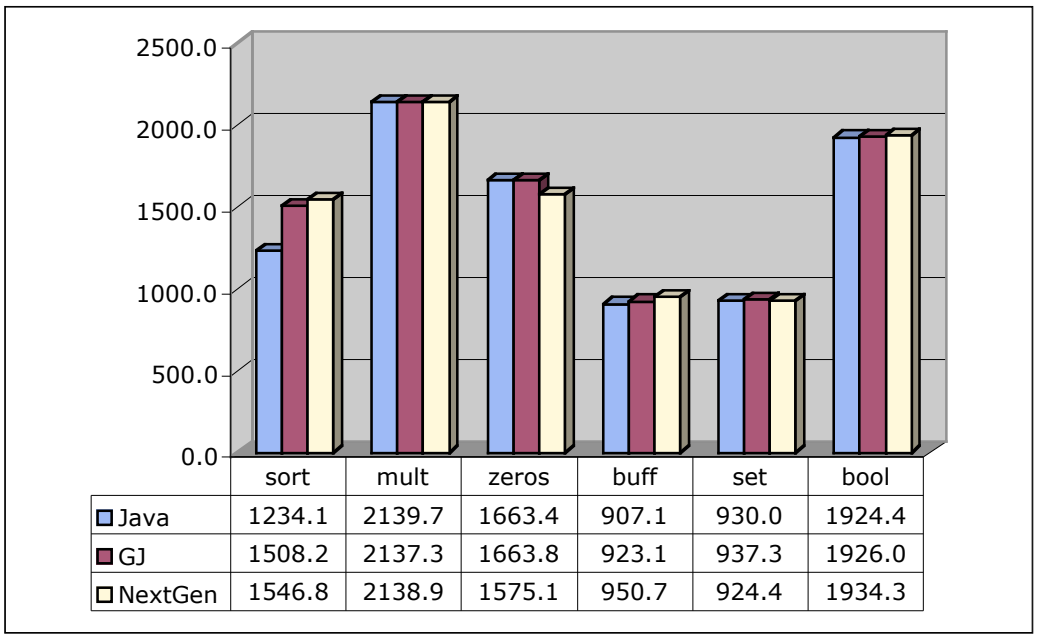
	sort	mult	zeros	buff	set	bool
Java	1002.0	1276.7	1201.6	551.2	692.3	1696.6
GJ	1001.6	1276.7	1201.1	553.3	692.2	1696.4
NextGen	1181.4	1340.7	1200.9	562.3	686.7	1669.0

Figure 6. Performance Results for Sun 1.3 Client (in milliseconds).



	sort	mult	zeros	buff	set	bool
Java	759.8	2097.2	1173.7	656.3	746.0	1531.7
GJ	1111.8	1942.3	1265.0	674.4	734.7	2033.0
NextGen	1251.1	1948.7	1366.3	729.1	742.3	1598.3

Figure 7. Performance Results for Sun 1.4 Server (in milliseconds).



	sort	mult	zeros	buff	set	bool
Java	1234.1	2139.7	1663.4	907.1	930.0	1924.4
GJ	1508.2	2137.3	1663.8	923.1	937.3	1926.0
NextGen	1546.8	2138.9	1575.1	950.7	924.4	1934.3

Figure 8. Performance Results for Sun 1.4 Client (in milliseconds).

5. FUTURE EXTENSIONS

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.

5.1 Full support for polymorphic methods

Polymorphic methods that require snippets 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.

5.2 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 is 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.

5.3 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 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.

5.4 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

ConDefList → *ConDef* | *ConDef* , *ConDefList*
Condef → *TypeVar* ({*ArgumentList* }) .

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)> { ... } .
```

5.5 Mixins as Classes with Variable Parent Types

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. 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 template class files for the mixin classes, 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].

6. 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 heterogeneous approach to implementing genericity has two serious 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 [12]. 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 [3]. We believe their results are misleading because they presume no method inlining for NextGen while presuming it for their implementation.[13] Moreover, they model the performance of the NextGen implementation scheme described in [3] 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 that most closely resembles NextGen is the extension of the .NET common runtime by Kennedy and Syme to support generic types in C# [7]. They follow the same, 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 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 [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.

7. CONCLUSION

Our NextGen implementation demonstrates that run-time generic types can be supported 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.

---

[1] GJ 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.

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

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

[4] Sun Microsystems officially proposed adding generics to the Java language in JSR-14 [11]. 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.

[5] Cartwright and Steele proposed such a scheme in [3], but it does not support polymorphic recursion which is allowed in Generic Java.

[6] If **C** is fully instantiated then **D** simply extends the instantiation class representing **C**.

[7] The 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 in unchanged.

[8] The 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 operations*** 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.

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

[10] Some 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

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

[12] There 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.

[13] In 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.

REFERENCES

[1] O. Agesen, S. Freund, and J. Mitchell. Adding parameterized types to Java. In OOPSLA 1997.

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

[3] R. Cartwright and G Steele. Compatible Genericity with Run-time Types for the Java Programming Language. In OOPSLA 1998.

[4] M. Flatt, S. Krishnamurthi, M. Felleisen. Classes and Mixins. In POPL 1998.

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

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

[7] A. Kennedy and D. Syme., Design and implementation of generics for the .NET Common Language Runtime. In PLDI 2001.

[8] A. Myers, J. Bank, B. Liskov. Pa rameterized Types for Java. ACM POPL 1997.

[9] M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. In POPL 1997.

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

[11] Sun Microsystems. Java Specification Request 14: Adding Generic Types to the Java Programming Language. Available on the internet at URL: http://jcp.org./jsr/detail/14.jsp.

[12] Viroli,, M. and A. Natali. Parametric Polymorphism in Java: an Approach to Translation Based on Reflective Features. In OOPSLA 2000.