Efficient First-Class Generics on Stock Java Virtual Machines*

James Sasitorn
Rice University
6100 South Main St.
Houston TX 77005
camus@cs.rice.edu

Robert Cartwright
Rice University
6100 South Main St.
Houston TX 77005
cork@cs.rice.edu

Abstract

The second-class formulation of generics in Java 5.0 discards generic type information during compilation. As a result, Java 5.0 prohibits run-time type-dependent operations, generates unavoidable unchecked warnings (type errors) during compilation, and causes unexpected behavior during execution. The NextGen Generic Java compiler eliminates these pathologies by retaining parametric type information at runtime and customizing the code in generic classes and polymorphic methods where necessary to support (parametric) type-dependent operations. NextGen is a production compiler for the entire Java 5.0 language; it executes on standard JVMs and is backward compatible with existing libraries and other binaries. Benchmarks show that the first-class implementation of generic types in NextGen has essentially the same performance as Java 5.0 and significanlty outperforms alternative first-class implementation architectures.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes, objects, polymorphism*

General Terms

Design, Language, Measurement, Performance

Keywords

first-class generics, Java implementation, parametric polymorphism, custom class loader, type erasure

1. Introduction

Java has revolutionized mainstream software development by supporting clean object-oriented design, comprehensive static type checking, "safe" program execution, and an unprecedented degree of portability. Despite these significant achievements, the Java language design has some annoying flaws. Prior to Java 5.0, Java did not support generic (parameterized) types, preventing the expression of many statically checkable program invariants within the type system and inhibiting the use of type abstraction (parametric polymorphism). Java programmers typically simulate parametric polymorphism by using the universal type `Object` or a more narrow bounding type in place of a type parameter `T`. Programmers then insert casts to convert an object from the bounding type back its actual type. For example, after extracting an element from a `Vector`, the programmer must cast the element from type `Object` to its actual type. Not only does this idiom clutter source code, but it obscures type abstractions in programs and degrades the precision of static type checking. Moreover, the absence of generic type checking encourages breaches in the use of type abstractions. For this reason, converting legacy code to generic form typically involves far more than adding parameterization to the type declarations.

Generics were added to the Java language by the *Java Specification Request 14: Adding Generics to the Java Programming Language*[9]—building on research by Odersky, Walder, and others on adding genericity to Java. This extension of the Java language is often referred to as Generic Java. Martin Odersky supported this effort by developing a well-engineered compiler for a formulation of Generic Java called GJ that supports genericity using `type erasure`. Sun subsequently adopted this approach to supporting generic types in Java; the Java 5.0 compiler is a direct descendant of Odersky's GJ compiler. GJ and Java 5.0 support a "second-class" formulation of Generic Java designed by Bracha, Odersky, Stoutamire, and Wadler[5]. This approach to genericity erases generic (parametric) type information from programs during compilation relegating generic types to a "second-class" status that prevents them from being used in any context that requires run-time support. The generic types for variables and expressions are analyzed during the type checking phase of compilation. After type checking, the generic types are "erased" to their non-parametric upper bounds. The unsupported operations include parametric casts, parametric `instanceof` operations, and `new` operations on "naked" parametric types, *e.g.*, `new T()`.

First-class approaches to implementing Generic Java have been proposed by Cartwright and Steele [6], Solorzano and Alagić [11], and Viroli [13, 12], but real implementations have been slow to emerge because first-class genericity is

*This research has been partially supported by the National Science Foundation, the Texas Advanced Technology Program, and Sun Microsystems.

SAC'06 April 23-27, 2006, Dijon, France

hard to implement efficiently on a stock Java Virtual Machine and Sun Microsystems has devoted its resources to supporting erasure-based approach.

The NextGen architecture developed by Cartwright and Steele makes extensive use of type erasure wherever it does not interfere with first-class behavior. As a result, Cartwright and his research group have been able to leverage the GJ and Java 5.0 code bases in building a series of progressively more comprehensive implementations of NextGen [4, 2, 1]. This paper reports on the culmination of that process: a production compiler for NextGen that supports the entire Java 5.0 language and interoperates with erasure-based binary code compiled using the conventional Java 5.0 compiler.

The most challenging technical issue involved in implementing NextGen on stock Java Virtual Machines is efficiently supporting polymorphic methods. The original architecture proposed by Cartwright and Steele assumed that all instantiations of generic classes and polymorphic methods could be determined and analyzed at compile time. Unfortunately, this assumption is true only if cyclic type constructions that generate an infinite number of type instantiations are prohibited.[1] GJ and Java 5.0 do not impose such a restriction, so programs with cyclic type constructions can generate an unbounded set of class and method instantiations (infinite in a non-terminating program). The solution to this problem is to generate and class and method instantiations at run-time. The flexible class loading mechanism of the Java Virtual Machine facilitates constructing new class instantiations as a program executes. The NextGen runtime includes a custom class loader that generates generic class instantiations as they are demanded from special template class files. If the Java Virtual Machine supported the dynamic addition of methods to classes, essentially the same technique could be used to efficiently implement polymorphic methods. But such a facility is not available which makes the efficient support of polymorphic methods a challenging software engineering problem.

Static polymorphic methods have a simple, efficient implementation: the type dependent operations (called *snippets*) in the body of the method can be abstracted out using the *template method* design pattern[7] to form a *snippet environment*. At each call site, the bindings of the polymorphic method type variables are known,[2] so the NextGen compiler can construct a singleton class (with no dynamic fields) containing the appropriate implementation for each snippet in the environment and pass it as an auxiliary argument to the polymorphic method.

Dynamic polymorphic methods are much more difficult to support using snippet environments because they can be *overridden*. For a given polymorphic method call site, the specific code body that will be executed depends on the class of the receiver in the call, which can change with each execution of the call. Moreover, if the class containing the dynamic polymorphic method is generic, snippets in the various bodies for the polymorphic method can depend on the bindings of generic type parameters in the *receiver* as well as the polymorphic methods type variables at the call site. Even worse, these generic type parameters can be introduced as *new* generic type parameters in subclasses where the polymorphic method is overridden. In other words, the execution of a dynamic polymorphic method may involve the bindings of type parameters that are not even in scope at the call site.

Besides reporting on the performance of the new production NextGen compiler, the primary technical contribution of this paper is a scheme for implementing dynamic polymorphic methods that performs almost as well in practice on stock Java Virtual Machines as the erasure-based implementation in Java 5.0. The scheme avoids the use of reflection except in a corner case that rarely occurs in practice. In fact, we have not yet seen an actual occurrence of the corner case in any the Generic Java code bases available to us including the Java 5.0 libraries, the Java 5.0 `javac` compiler, and the DrJava programming environment.

2. Type Erasure

In erasure-based implementations of genericity, each parametric class `C<T>` generates a single class file containing the erased base class `C`. Similarly, each parametric method `<T>m` generates a single erased method m. The erasure of a parametric type `T` is obtained by replacing each type instantion `C<E>` by its base (*raw*) type `C` and replacing each "naked" type parameter `T` by its upper bound.

To understand the practical implications of the erasure process, consider the use of the `Cloneable` interface with a generic class. The method `Object clone()` is inherited by all classes from class `Object` but it throws an exception unless the invoking class implements the `Cloneable` interface. Of course, to make any use of the result of clone for a class `C`, the client must cast the result to type `C`. If `C` is generic, then the cast must be to a generic type `C<E>`, which is prohibited in Generic Java (unless the type information `E` can be statically inferred).[3] Each such cast is implemented by the Java 5.0 compiler as a cast to the erasure of the specified generic type, breaking type safety. In NextGen, all generic casts are legal so the `Cloneable` interface naturally applies to generic classes in exactly the same way it does to conventional ones.

A more dramatic problem occurs for operations involving "naked" parametric types such as `new T()` and `new T[]`. These allocation operations are not accepted by the Java 5.0 compiler, which can force programmers to perform some elaborate workarounds. For example, the generic version of the method `Object[] toArray( )` declared in the interface `java.util.Collection<T>` requires an argument of type `T[]` to be passed to the method: `<T> T[] toArray(T[] a)`. Given this extra array argument, the classes implementing `java.util.Collection<T>` use reflection to allocate an array of type `T[]`.[4]:

```
a = (T[])java.lang.reflect.Array.newInstance(
     a.getClass().getComponentType(), size);
```

To cope with the limitations imposed by type erasure, the type-safe subset of Java 5.0 prohibits operations that

[1] For example, consider a generic class `Foo<T>` with a field of type `Foo<List<T>>`. Then a program using type `Foo<Integer>` potentially generates the type instantiations `Foo<Integer>`, `Foo<List<Integer>>`, `Foo<List<List<Integer>>>`, ... But the execution of the program may only load a finite number of these types (classes) because classes are only loaded as they are demanded.

[2] At runtime, all such bindings are *ground*, *i.e.* devoid of type variables.

[3] In practice, the Java 5.0 compiler accepts programs that contain generic casts but it reports a type error (in the form of a "warning message") for each use of a generic cast breaching the type system.

[4] Suprisingly, the code contains a generic cast to `T[]` *that breaches the type system*. The `javac` compiler uses this additional information for typechecking. It is erased to `Object[]` in the bytecode.

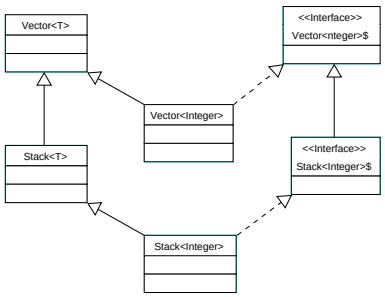

Figure 1: Sample NextGen Type Hierarchy

depend on run-time type information. The prohibited type-dependent operations include:

- parametric `instanceof` tests;
- `catch` operations that take parameterized exception types;
- `new` operations on "naked" parametric types such as `new T()` and `new T[]`
- static operations on generic classes, such as `T.class`

The Java 5.0 compiler issues `unchecked` warnings for the remaining operations that still cannot be statically guaranteed, namely parametric casts.

3. NextGen Architecture

The NextGen compiler eliminates the pathologies associated with type erasure by maintaining parametric type information in generic classes and customizing code involving (parametric) type-dependent operations. NextGen is upward compatible with the Java 5 and runs on existing JVMs.

In NextGen, the relationships between generic classes and their instantiations are encoded in a non-generic class hierarchy. A separate interface, used to preserve subtyping relationships, is generated on demand (via a specialized classloader) for each instantiation of a generic class. An "instantiation class" is also generated to hold the type-dependent code for a particular instantiation. Code common to all instantiations is factored out into a common, `abstract` "base class". This approach essentially decouples the object subtyping from code inheritance. Figure 1 illustrates the encoding of a generic class `Stack`, its parent class `Vector`, and the instantiation `Stack<Integer>`.

Parametric interfaces are implemented in essentially the same way as parametric classes. Since interfaces define method contracts without providing code, the translation of a parametric method generates a base interface and an instantiation interface but no instantiation class.

3.1 Snippet Methods

The type-dependent operations used in a generic class `C<T>` are not erased in `C`, but rather translated into calls on synthetically generated abstract *snippet* methods. The instantiation classes `C<E>` override the abstract *snippet* methods in `C` to provide specialized code encapsulating the type-dependent operations for `C<E>`. Since these lightweight interfaces contain no fields or methods, their use only marginally affects program code size. A more detailed discussion of this issue appears in [6, 4].

```
class List<T> {
   T f; List<T> r;
   List(T x, List<T> y) { f = x; r = y; }

   static <T,U> List<Pair<T,U>> zip (List<T> l, List<U> r) {
      ...
     return new List<Pair<T,U>>( new Pair<T,U>(l.f, r.f),
                       List.<T,U>zip(l.r, r.r));
    }
}
class Pair<X,Y> {
    X x; Y y;
    Pair(X a,Y b) { x=a; y=b; }
}

class Client {
    public static void main(String[] args) {
       List<Integer> i = new List<Integer>(1, ... );
       List<String> s = new List<String>("A", ... );

       List<Pair<Integer, String>> p = List.zip(i, s);
    }
}
```

Figure 2: Static Polymorphic Method Source Code

```
class List<T> {
   T f; List<T> r;
   List(T x, List<T> y) { f=x; r=y; }

   static <T,U> List<Pair<T,U>> zip (List$SE$ $env,
                              List<T> l, List<U> r) {
     ...
     return $env.new$List<Pair<T,U>>(new Pair<T,U>(l.f, r.f),
                       List.<T,U>zip($env, l.r,r.r));
     }

class Pair<X,Y> {
    X x; Y y;
    Pair(X a,Y b) { x=a; y=b; }
}

class Client {
   public static void main(String[] args) {
     List<Integer> i = new List<Integer>(1, ... );
     List<String> s = new List<String>("A", ... );

     List<Pair<Integer, String>> p =
            List.zip(List$SE<Integer,String>.ONLY,i,s);
    }
}

interface List$SE$ {
     List<Pair> new$List<Pair<T,U>> (Pair f, List r);
}

class List$SE<Integer, String> {
   List<Pair> new$List<Pair<T,U>> (Pair f, List r) {
     return new List<Pair<Integer, String>> (f, r);
   }
}
```

Figure 3: Static Polymorphic Method Translation

3.2 Polymorphic Methods

Polymorphic method declarations introduce type parameters whose scope is confined to the body of a method. These type parameters are statically instantiated at each method call site. Conceptually, polymorphic methods can be divided into two categories: static polymorphic methods which are declared as static methods (essentially conventional proce-

```
class List<T> {
    ...
    <U> List<Pair<T,U>> zip (List<U> r) {
          return new List<Pair<T,U>>( new Pair<T,U>(f, r.f),
                        r.<U>zip(r.r));
    }
}
class Client {
   public static void main(String[] args) {
     List<Integer> i = ...;
     List<String> s = ...;
     List<Pair<Integer, String>> p = i.zip(s);
   }
}
```

Figure 4: Dynamic Polymorphic Method Source Code

dures); and dynamic polymorphic methods which involve object-oritented dispatch in the context of a recevier.

The translation of polymorphic methods is very similar to the translation of parametric classes. Type-dependent operations in the body of the method are "snippetized". If the operation depends only on class-level parameterization, the snippet method is stored in the instantiation class; otherwise the snippet is stored in a special class called a snippet environment. The snippet environment is simply a lightweight singleton class containing only snippet methods and a static field bound to the only instance of the class. The name of the snippet environment encodes the instantiated parametric types. The remaining code of the polymorphic method remains intact in the defining class. Also, the formal parameters of the method are prepended with a special interface implemented by all relevant snippet environments.

Each polymorphic call site generates an instantiated snippet environment based on the statically inferred types at that site and passes it as a parameter in the method invocation. If the inferred static types depend on a type variable in an enclosing generic class or polymorphic method, the code that creates the snippet environment must be "snippetized" in the enclosing class or polymorphic method. The following sections describe in detail how snippet environments are generated for static and dynamic polymorphic methods.

3.2.1 Static Polymorphic Methods

Since static polymorphic methods have no dynamic receiver, all of the polymorphic type information for a method call can be inferred statically. As a result, the correct snippet environment can be efficiently inferred and generated.

Figure 3 shows the the NextGen translation of the source code show in Figure 2. The invocation of `List.zip(...)` has the statically inferred type arguments: `T` is instantiated to `Integer` and `U` is instantiated to `String`. Thus, the snippet environment `List$SE<<Integer,String>>.ONLY` is loaded at the call site. The syntax `<<` and `>>` denote the types encoded by the `snippet environment`.

3.2.2 Dynamic Polymorphic Methods

In the case of dynamic polymorphic methods, the snippet environment generated for a call site must carry not only the bindings for the type parameters introduced in the method definition, but also the bindings of type parameters in the generic type of the receiver. Both sets of bindings are determined by static type checking. Figure 4 shows a dynamic version of the `zip` function defined in Figure 2.

At the call site `i.zip(s)`, the object `i` has static type `List<Integer>`. So the call site defines the same type parameter bindings as in the static case discussed earlier: `T` is bound to `Integer` and `U` is bound to `String`. NextGen uses slightly different name mangling schemes for static and dynamic method snippet environments because they are not interchangeable. After name mangling, the call site becomes:

```
List<Pair<Integer, String>> p =
  i.zip(List$$DE<<Integer,String>>.ONLY, s);
```

and the method signature becomes:

```
<U> List<Pair<T,U>> zip (List$DE$ $env, List<U> r)
```

Naming issues aside, the most important aspect of this translation is how snippet environments cope with dynamic dispatch. The class of the receiver can vary during program execution; any class type that is a subtype of the static type of the receiver is possible. Hence, the generated snippet environment must contain the snippets for *all* of the definitions of the method consistent with the static type of the receiver. In each class that belongs to the static type of the receiver, the method may have a distinct definition.

Consider a `List` subclass `RevList` that extends the `List` class in Figure 4, overriding the method `zip` to return a reversed list of zipped pairs:

```
class RevList<T> extends List<T> {
    ...
    <U> List<Pair<T,U>> zip (List<U> r) {  ... }
}
```

At the call site in the `Client` class, the object `i` can be an instance of `List<Integer>` or `RevList<Integer>`. As a result, the snippet environment passed at this site must contain all of the snippets for the definitions of `zip` in both `List` and `RevList`.

Since snippet environment classes only contain code, it is advantageous to combine environments with compatible type bindings. Obviously, distinct calls on the same polymorphic method with the same generic type bindings (for both the method type parameters and the receive type parameters) should use the same snippet environment. In addition, it conserves space and class loading time to use the same snippet environment for all polymorphic method calls with identical type bindings, regardless of the polymorphic method that is being called.

3.2.3 Reflection in Pathological Cases

The snippet environment technique described in the preceding subsection may not work if a subclass introduces a new type parameter `S` that is not bound to a type parameter in the parent. If the statically inferred receiver type at a call site is (a type instantation of) the parent class, then the generated snippet environment will not contain a binding for the new type parameter `S`, and therefore not be able to support type-dependent operations involving `S`. Consider the following:

```
class AssociativeList<S,T> extends List<T> {
    S x; T y; AssociativeList<S,T> r;
    PairList(S a, T b, AssociativeList<S,T> c)
          { x=a; y=b; r=c; }

    <U> List<Pair<T,U>> zip (List<U> r) {  ... }
}
```

Recall the call site generates the snippet enviornment `List$$DE<<Integer,String>>.ONLY`. Since a binding for the type parameter `S` does not appear in the static type of the receiver in the call site in the `Client` class, the snippet environment generated for this site is independent of `S`. In fact, different values of the receiver type may have different bindings for `S` and some (such as instances of the classes `List` and `RevList`) may have no binding for `S` at all. As a result, the code for the body of `zip` in the class `AssociativeList` must use reflection to determine what snippet environment instantiation should be loaded in order to resolve the proper meaning of snippets in this body. Note that the body of `zip` in `AssociativeList` must involve snipppets that depend on the bindings of type variables in the receiver type and the bindings of type variables in the method call type before this reflective mechanism is necessary.

The easiest way to implement this approach is to use "bridge methods". If a method definition `m` in class `C` needs to invoke a snippet requiring reflection, the NextGen compiler generates two methods to implement `m`:

- a method `m` that takes a conventional snippet environment just like versions of `m` in superclasses that do not require reflection and
- an overloaded variant of `m` that takes an augmented snippet environment including the snippets that depend on new class type parameters not present in the context where method `m` is introduced.[5]

The definition for `m` with the conventional snippet signature is bridge code that uses reflection to create the *correct* snippet environment and passes it to the variant version of `m` that takes an augmented snippet environment that includes the special snippets depending on new class type parameters. Note that this implementation only incurs the overhead of reflection when the receiver object requires it. A particular call site for the method `m` only uses reflection when (*i*) the receiver object *o* is in the scope of type parameters that are not in the scope of the static type of the receiver expression and (*ii*) the definition for `m` in class of *o* involves atomic operations that depend both on the bindings of new class type parameters and the bindings of method parameters.[6] Executions at a call site do not use reflection unless the receiver object requires it. In addition, if the static type of the receiver includes bindings for the new type parameters, then the variant of `m` that takes an augmented snippet environment is directly invoked at the call site because the augmented snippet environment can be determined statically by the compiler.

3.2.4 Incidence of Reflection in Practice

Although the body of Java code using generics is still comparatively small, *we have not yet encountered a single case where reflection is required* in any of the generic code bases that we have examined including `javac`, DrJava [3], and the Java libraries that have generified. Introducing new type variables in subclasses is uncommon and overriding a polymorphic method in a way that depends on both these new bindings and method type variable bindings is even more unusual. In practice, we do not believe that the reflection overhead incurred in pathological cases will have any measurable impact on the performance of real programs. The only way we have been able to detect this potential overhead is to write artificial programs specifically designed to exhibit the pathology.

[5]The definition of `m` that does not override a definition of `m` in its superclass. In C#, such a method definition is labeled with the `virtual` modifier while all of the overriding definitions must be labeled with the `override` modifier. Java does not make this syntactic distinction.

[6]Snippet operations that depend only on the bindings of new type parameters can be snippetized like any other type-dependent operations that appear in the class in question. Snippet operations that depend only on the the bindings of polymorphic method parameters (statically determined at the call site) can be included in the conventional snippet environment for `m`.

4. Implementation Details

From the user's point of view using NextGen requires a minimial adjustment from their current Java model: The command `ngc` replaces `javac`, and `nextgen` replaces `java`. The compilation of Generic Java code generates extra class files for the NextGen templates.

Internally, NextGen adds two stages of processing to the normal Java 5.0 compilation process: a "type flattening" stage to encode parametric types in the AST (abstract syntax tree) and a "snippet patching" stage to collect snippetized type-dependent operations in the generated class files. Both stages walk the corresponding code representations and destructively transform segments of code. In conjunction, these two stages snippetize type-dependent operations and generate the template class files used to represent uninstantiated parametric classes and methods environments.

A template class file looks exactly like a conventional class file except that strings in the constant pool may contain embedded references to the type parameters for the class. These references specify an index of the form {0},{1}, *etc*, specifying which type parameter binding should be inserted when the class file is instantiated. The NextGen classloader replaces these references with the corrsesponding type arguments (represented as mangled strings) to generate specific generic class instantiations.

Snippet environments are synthetic generic classes that are directly generated by the NextGen compiler. These synthesized classes could be represented in Generic Java source code, except for the fact that their source representation requires a new language feature not present in Java 5.0, namely per instantiation static fields. Each instantation of a snippet environment is a singeton class with its own static singleton field. In Java 5.0, "per-instantiation" static fields are impossible because Java 5.0 uses a single class to represent all of the "instantiations" of a generic class.

In NextGen, each class instantiation is represented by a distinct class enabling each instantiation to have its own static fields. The NextGen compiler does not currently support this language feature at the source level because there is no syntax in Generic Java for it. If a consensus can be reached on a suitable syntax for "per-instantiation" static fields, then support for the feature could easily be added to the NextGen compiler. The NextGen compiler generates template class files for snippet environments that contain such fields.

4.1 Cross Package Instantiation

The NextGen design presented in Section 3 does not specify where template class files are placed in the Java namespace. The most obvious location is the package containing the corresponding generic class.[7] However, this placement

[7]For a snippet environment template class, the class introducing the

can collide with Java visibility rules if private or package-private types are passed as type parameters [10].

The simplest solution to this instantiation problem is to automatically widen a private or package-private class to public visibility if necessary when it is passed as a type argument. This tradeoff for simplicity at the cost security has a well-known precedent in Java. When an inner class refers to private members of its enclosing class, `javac` automatically widens the visibility of the relevant members by generating getters and setters with package visibility. Although more secure implementations are possible, the Java designers decided to sacrifice some visibility for the sake of performance and simplicity. We have done the same.

4.2 Separate Compilation

By performing "snippet patching" at the bytecode level, NextGen can propagate new snippets to any previously compiled class file by updating the class file. If the class file is in a read-only directory or jar, then NextGen creates an updated snippet environment in the corresponding location (relative to the classpath root for the file) in the build directory.[8]

4.3 Wildcards

In NextGen, a reference to an unbounded wildcard type `C<?>` in executable code is translated to the abstract base class `C` for the type. References to bounded wildcard types are illegal in executable contexts (*e.g.*, casts and `intanceof` operations. Similarly refences to partially instantiated wildcards (*e.g.*, `D<?, String>`) in executable contexts are illegal. NextGen could be extended to support partially instantiated wildcards by adding instantiation interfaces for all possible partial wildcard types to the generic interface hierarchy, but we doubt that the minor gain in functionality would be worth the added complexity.

5. Performance

In analyzing the performance of NextGen, we are primarily concerned with measuring how well NextGen performs on large scale Generic Java programs that are compatible with Java 5.0. There is a growing body of production code written using Java 5.0 generics that we can use for this purpose. We also interested in assessing how well NextGen supports polymorphic methods in comparison with Java 5.0 and alternative first-class implementations. In a prior paper [4], we demonstrated that NextGen was essentially indistinguishable in performance from GJ, a Java 5.0 `javac` prototype, on a variety of benchmark programs of modest size. But these benchmarks made very limited use of polymorphic methods because our prototype NextGen compiler only supported polymorphic methods that did not involve any snippets. We did not provide any experimental evidence that polymorphic method requiring snippets could be efficiently supported using the NextGen architecture.

Unfortunately, there is a paucity of credible alternatives against which to measure the performance of NextGen other than the Java 5.0 `javac` compiler. The most mature and credible competing architecture for supporting first-class generics is the "Load-time Management" (LM) heterogeneous translation developed by M. Viroli and A. Natali. Although no compiler for LM is available, we have been able to hand translate some simple programs from Generic Java to non-generic Java source code using the LM architecture as described in papers by Viroli and Natali [13, 12]. As a result, we were able to directly measure the performance of NextGen and LM on a series of microbenchmark programs that intensively exercise polymorphic methods in full generality.

Since no established benchmark suite for Generic Java currently exists, we have developed our own Generic Java benchmarks to analyze the performance of NextGen [4]. We have augmented this test suite by a few large programs written in Generic Java, most notably the Java 5.0 compiler `javac`.

The NextGen compiler does *not* generate the same code for legal Java 5.0 programs as `javac` because Java 5.0 *erases* parameterized `new` operations (*e.g.*, `new Vector<String>()`) to corresponding unparameterized operations (*e.g.*, `new Vector()` while NextGen does not. Hence, a NextGen implementation can potentially add overhead to the execution of existing Java 5.0 source code. The most important issue that our benchmarking process addresses is how much if any overhead that NextGen adds to the execution of Java 5.0 programs. Since NextGen generates exactly the same code for non-generic programs as `javac`, we have confined our attention to programs that use generics.

All of the following benchmarks were performed on a 2.8 Ghz Pentium 4 with 512 MB of RAM running Debian Linux 3.1. Each benchmark was executed twenty-one times using the Sun 1.5 Java Virtual Machine (JVM) executed in server mode. The corresponding results for client mode are similar but less consistent because JIT optimizations are less aggressive in this mode. We have chosen to focus on server mode performance because we are particularly interested in assessing how well the JIT optimization process eliminates the putative overhead introduced by first-class generics.

We dropped the first iteration of each benchmark from the computed average because it deviated significantly from the remaining 20 iterations. We presume that the source of this deviation is the overhead of JVM startup and JIT compilation of the code, neither or which we were trying to measure. After dropping the first run, the variance among iterations for each benchmark was less than 10%.

Our first set of benchmarks consist of "microbenchmarks" designed to measure the performance of code involving polymporphic methods. The column labels `javac` (the Java 5.0 compiler) and `ngc` (the NextGen compiler) should be self-explanatory. The column labeled `lm` gives the performance of the "Load-time Management" (LM) architecture developed by Viroli and Natali.

The programs `zip1`, `zip2`, `zip3` all perform a standard "zip" operation on two lists (the receiver and an argument) of equal length. This benchmark is revealing because the `zip` method requires a "mixed' snippet that depends on both the type bindings of the receiver (the element type of the receiver) and the polymorphic method (the element type of the method argument). The only difference between `zip1`, `zip2`, `zip3` is the length of lists that are repeatedly zipped. The programs `refl` and `refl2` update lists of pairs or triples where the triple class extends the pair class and introduces a new type variable that is the type of the third element. The polymorphic update method on a pair or triple changes the second element to a new value where the type of the new

the method(s) corresponding to the snippet environment.

[8] If no build directory is specified, the inferred root of the source file hierarchy is used as the build directory.

Program	javac	lm	overhead	ngc	overhead
zip	420.2	574.6	36.8%	386.2	-8.1%
zip2	446.9	601.3	34.6%	457.0	2.3%
zip3	172.5	199.2	15.5%	154.3	-10.6%
refl	428.0	467.3	36.8%	430.7	0.6%
refl2	424.9	470.5	36.8%	554.9	30.6%

Figure 5: Generic Microbenchmarks (ms)

value is a polymorphic method parameter. This is an artificial benchmark specifically designed to assess the overhead created by the use of reflection in implementing polymorphic methods in the pathological case discussed in Section 3.2.3. In `refl`, the list to be updated does not contain any triples so no reflection is required during execution. In `refl2`, 5% of the elements are triples forcing the use of reflection. The `refl2` benchmark shows the use of reflection in NextGen is costly. Fortunately, it is almost never encountered in practice.

With the exception of the `refl2` benchmark, the performance of NextGen is indistinguishable from `javac` and significantly better than LM. As we discussed in Section 3.2.4, we believe that there is growing evidence that the pathological situation captured in the `refl2` benchmark essentially never happens in practice.

The second table of benchmarks, given in Figure 6, shows the performance of Java 5.0 and NextGen for some larger programs including the production program `javac`. In every case, the performance differences (except for `sort` where `ngc` is 26% is faster for unknown reasons) are insignficant, on the order of the variations between runs of the same benchmark. The first six programs were developed by Eric Allen to measure the performance of an early prototype of the NextGen compiler [4]. `sort` performs a linked-list variant of quicksort. `mult` multiplies the values of all of the nodes (both leaves and internal nodes) of a binary tree. `zeros` confirms that a binary tree does not have consecutive zeros on any path from the root to a leaf. `buff` is an implementation of `java.util.iterator` over a `StreamReader`. `set` is an implementation of generically-typed multi-sets. `bool` is a boolean expression simplifier. For a more detailed description fo these six benchmarks, see [4].

The remaining three benchmarks are significantly larger programs. `jam1` and `jam2` are tests of an interpreter for a simple functional programming language which is assigned as a series of programming exercises in the undergraduate course at Rice University on programming languages. It consists of about 2200 lines of Generic Java code. `jam1` executes all of the unit tests used to grade the program. `jam2` focuses on the most computationally demanding unit tests in the test suite including a prime sieve based on lazy streams. This interpreter makes more aggressive use of generics and polymorphic methods than any other program we have seen. The `javac` benchmark is the `javac compiler` applied to the `javac` source code distributed in the J2SE 5.0 JDK source code available under the Sun JRL license. It consists of about 43,500 lines of code.

In every case, the performance of NextGen is essentially indistinguishable from Java 5.0. We think it is noteworthy that we achieved these results without any modifications to the JIT optimization process to address the apparent overhead (such as snippet calls and passing snippet environments) introduced by NextGen generics.

Program	javac	ngc	overhead
sort	553.0	409.7	-25.9%
mult	718.8	702.2	-2.3%
zeros	683.9	688.2	0.6%
buff	591.2	615.9	4.2%
set	806.3	804.6	-0.2%
bool	195.6	194.6	-0.5%
jam1	715.4	716.9	0.2%
jam2	716.5	732.7	2.3%
javac	1248.8	1277.6	2.3%

Figure 6: Production Benchmarks (ms)

6. Related Work

The "Load-time Management" (LM) architecture developed by Viroli and Natali uses the Java Reflection APIs to carry parametric type information[13]. In their implementation, each generic class contains an additional field storing parametric type information called a *type descriptor*. For polymorphic methods, LM passes an extra parameter indexing into a *Virtual Parametric Method Table* (VPMT) associated with each potential dynamic receiver type. VPMTs store the parametric type information available at each call site. Parametric casts and parametric `instanceof` operations require a traversal of the *type descriptor* hierarchy.

While the use of reflection in LM avoids code duplication, it creates a significant performance penalty for two reasons. First, it adds an extra word (the type descriptor) to every generic object. For a collection class like `Pair` with only two fields, adding an extra field incurs a space penalty of 25% assuming a two word header. Second, LM requires the use of reflective code that traverses and compares type descriptors to support parametric casts and `instanceof` checks.

The .NET Framework supports first class generics using a modified Common Language Runtime (CLR), a virtual machine analogous to the Java Virtual Machine [8]. Since the developers of .NET generics were free to modify the CLR, their design is much less constrained by compatibility requirements than NextGen. Nevertheless, their implementation is remarkably similar to NextGen. Like NextGen, they use a base class to share all of the code common to object type instantiations. Like NextGen, they use lightweight template classes (*vtable*s in .NET parlance) to carry the information required for each generic class instantiation. Like NextGen, they pass snippet environments (called *dictionaries*) as extra arguments to static polymorphic methods. The .NET Framework 2.0 Beta also supports dynanmic polymorphic methods, but the implementation techniques involved have not been published in the literature.

7. Conclusion

We have demonstrated that first-class generics can be supported on stock Java Virtual Machines with essentially no additional overhead. A production release of the NextGen compiler is available for download at:

`http://www.cs.rice.edu/~javaplt/nextgen`

8. References

[1] E. Allen, J. Bannet, and R. Cartwright. Mixins in Generic Java are sound. Technical report, Rice University, 2003.

[2] E. Allen and R. Cartwright. The case for run-time types in Generic Java. In *Principles and Practice of Programming in Java*, 2002.

[3] E. Allen, R. Cartwright, and B. Stoler. DrJava: A lightweight pedagogic environment for Java. In *SIGCSE*, 2002.

[4] E. Allen, R. Cartwright, and B. Stoler. Efficient implementation of run-time generic types for Java. In *IFIP WG2.1 Working Conference on Generic Programming*, 2002.

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[6] R. Cartwright and G. L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In C. Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Vancouver, British Columbia*, pages 201–215. ACM, 1998.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] A. Kennedy and D. Syme. Design and implementation of generics for the .net common language runtime. In *PLDI*, 2001.

[9] S. Microsystems. Sun Microsystems, Inc. JSR 14: Add generic types to the Java Programming Language, 2001.

[10] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[11] J. H. Solorzano and S. Alagić. Parametric polymorphism for java: a reflective solution. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 216–225, New York, NY, USA, 1998. ACM Press.

[12] M. Viroli. Parametric polymorphism in java: an efficient implementation for parametric methods. In *Selected Areas in Cryptography*, pages 610–619, 2001.

[13] M. Viroli and A. Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. *ACM SIGPLAN Notices*, 35(10):146–165, 2000.