Compatible Genericity with Run-time Types for the Java$^{\text{TM}}$ Programming Language

Robert Cartwright, Rice University*
Guy L. Steele Jr., Sun Microsystems Laboratories

Abstract

The most serious impediment to writing substantial programs in the Java$^{\text{TM}}$ programming language is the lack of a *genericity* mechanism for abstracting classes and methods with respect to type. During the past two years, several research groups have developed Java extensions that support various forms of genericity, but none has succeeded in accommodating general type parameterization (akin to Java arrays) while retaining compatibility with the existing Java Virtual Machine. In this paper, we explain how to support general type parameterization—including both non-variant and covariant subtyping—on top of the existing Java Virtual Machine at the cost of a larger code footprint and the forwarding of some method calls involving parameterized classes and methods. Our language extension is forward and backward compatible with the Java 1.2 language and run-time environment: programs in the extended language will run on existing Java 1.2 virtual machines (relying only on the unparameterized Java core libraries) and all existing Java 1.2 programs have the same binary representation and semantics (behavior) in the extended language.

1 Introduction

The Java$^{\text{TM}}$ Programming Language (hereafter called simply Java) has achieved widespread commercial acceptance because it supports a truly portable object-oriented programming model with safe program execution, yet it has a familiar syntax similar to that of C++. Nevertheless, Java has some significant limitations from the perspective of software engineering that could be eliminated by judicious language extensions. Ironically, Java's portable programming model makes refining the language more difficult because it imposes severe constraints on the implementation of language extensions.

In our experience, the most serious impediment to writing substantial programs in Java is the lack of a mechanism for abstracting classes and methods with respect to type. Type parameterization can be simulated in Java by using the universal reference type `Object` in place of type parameters and explicitly casting values of type `Object` to their intended types. But this coding idiom is clumsy and error prone because the required casts are tedious to write and largely defeat the error-detection properties of Java's static typing discipline.

During the past two years, several research groups have investigated the problem of extending Java to support the parameterization of classes and methods by type. Odersky and Wadler have designed and implemented an ambitious extension of Java called Pizza [OW97] that supports parameterized class and methods, as well as algebraic data types and first-class closures—while retaining compatibility with the existing Java Virtual Machine. Myers, Bank, and Liskov have designed and implemented an extension of Java called PolyJ [MBL97] supporting parameterized classes that relies on an extension of the Java Virtual Machine. Agesen, Freund, and Mitchell have designed an extension of Java [AFM97] supporting parameterized classes assuming a revision of the class loader and an extension of the class file format to support parameterized class files. Thorup and Torgersen have proposed extending Java to support virtual types [Tho97,Tor98], which accommodates class specialization by narrowing "virtual" type members of classes.

Among these proposals, only those of Odersky-Wadler and Thorup-Torgersen require no change to the existing Java Virtual Machine. Moreover, only the Odersky-Wadler proposal has addressed the issue of forward and backward compatibility with the JVM standard libraries. For this reason, Gilad Bracha and David Stoutamire from JavaSoft are collaborating with Odersky and Wadler on the definition and implementation of a derivative of Pizza called GJ [BOSW98].

In contrast to Pizza, GJ adds only type parameterization to Java. To ensure compatibility with the existing Java language design and software base, GJ is designed to satisfy the following ambitious design objectives:

- a simple, easily-understood semantic definition consistent with the existing language definition;
- complete forward and backward compatibility of Java Virtual Machine code including:
  - efficient implementation of extended language without changing the Java Virtual Machine (JVM);
  - compatibility of new parameterized code with the unparameterized Java 1.2 Core Libraries; and

*This research was primarily conducted while this author on sabbatical at Sun Microsystems Laboratories. Partial support was provided by NSF grants CDA-9414170 and CCR-9633746 and by Texas ATP grant 003604-006.

  - compatibility of unparameterized Java 1.2 code with new parameterized versions of standard Java libraries; and

- avoidance of name mangling.

GJ adroitly achieves these objectives using a technique called *type erasure*, which is explained in detail in Section 2. In essence, type erasure converts all occurrences of type parameters in program text to the universal type Object. In this way all the instantiations of a given parameterized class such as Vector⟨T⟩ are collapsed onto a single class Vector⟨Object⟩ which is identical to an unparameterized Vector class intended to support the generic coding idiom. In this way, GJ transforms Java with type parameterization into ordinary Java. As part of the type erasure transformation, GJ inserts the casts required to recover the erased type information wherever it is needed in the transformed program text.

The primary disadvantage of type erasure is that it cannot support type dependent primitive operations such as

instanceof Vector⟨Integer⟩
new T(...)
new T[...]

where T is a type parameter. Since objects of parametric type (other than built-in arrays) do not carry any type information at run-time, GJ prohibits primitive operations that depend on this information with one notable exception: to accommodate the parametric generalization of existing library classes, GJ provisionally supports the operation **new** T[...] by translating it to **new** Object[...], which is inconsistent with parametric type-checking. GJ generates an explicit "unchecked" warning message whenever it encounters such an operation.[1]

This anomaly in GJ is inescapable in any parameterization scheme that relies on type erasure because Java arrays carry their element type at run-time, enabling Java to support dynamically type-checked array updates, parametric **instanceof** tests and type casts on arrays. In addition, the Java type system forces array assignments to be type-correct, implying that an array value of type Object[ ] cannot be assigned to a variable of type Integer[ ]. As a result, an apparently type-correct GJ program can generate a run-time error. Consider a parametric class C⟨T⟩ with a field T[ ] x; that is initialized by an operation **new** T[...] (generating an "unchecked" warning message). If the variable a has type Integer[ ], and b is an object of type C⟨Integer⟩, then the assignment a = b.x will generate a run-time type error because the erased type of b.x is Object[ ] not Integer[ ].

In this paper, we propose a more general parametric extension of Java than GJ, dubbed NextGen, that is a strict superset of the type-sound portion of GJ with the exception of reflection (which reveals the presence of class instantiations) and the treatment of static fields as discussed below. Like GJ, NextGen is a conservative extension of Java without reflection[2] and ensures full forward and backward compatibility with the JVM. NextGen differs from GJ in four respects:

- Parametric type expressions—including those containing type variables—*can be used anywhere that conventional types are used.* Hence, **new** operations, casting operations, and **instanceof** operations can be parametric. They can even contain type variables.

[1] The use of the operation **new** T[...] in parameterized classes is discouraged in GJ; the workaround of allocating the array at the point of call (where the binding of T is known) is generally preferred.

[2] The meaning of existing Java source programs that do not use the reflection API is unaffected.

- Type variables can be declared as non-variant or covariant.[3] Different type checking rules apply in each case (as discussed below). Both non-variant and covariant type variables can have a declared upper bound. Bounds may be parametric and may include type variables (declared as class parameters). If no bound is declared, the default upper bound type is Object.

- Static members (including static inner classes) behave by default as if they are replicated in every instantiation of a parametric class; this interpretation is dictated by the simple semantic model for NextGen described below. In GJ, static members are shared across all instantiations of a class because GJ implements the various instantiations using a single run-time class. NextGen supports both shared and replicated static variables. We discuss this issue in Section 4.

- NextGen is always type-sound with respect to parametric type declarations. GJ is type-sound except for the problematic operation **new** T[...] (where T is a type parameter) which generates an "unchecked" warning message. Programs that contain this operation can generate run-time type errors even though they are type-correct according conventional parametric type checking rules.

NextGen extends Java to the point where it has essentially the same type parameterization facilities as Eiffel [Mey92] without compromising Java's static typing discipline.

2 Overview of NextGen

The *syntax* of type parameterization in NextGen is a minor extension of that for GJ, which is loosely based on the syntax of templates in C++. A parameterized class (interface) definition has exactly the same form as a conventional class (interface) definition in Java except for the insertion of a list of type variable declarations enclosed in "pointy brackets" ⟨...⟩ and separated by commas. Each type variable declaration may be of the form

Identifier

or

Identifier **extends** *ClassOrInterfaceType*

The latter form specifies an upper bound on the binding of the type variable. The upper bound is any Java type expression, possibly including the type variables introduced in this definition. The scope of a type variable introduced in a class definition is the entire text of that class definition. If the optional bounds clause in a type variable declaration is omitted, the default upper bound is type Object.

In NextGen, a parametric class (or interface) instantiation $C\langle T_0, \ldots\rangle$ may appear anywhere that a simple class (or interface) name C may appear. A type variable may appear anywhere that a simple class (or interface) name may appear *except* as the direct supertype in an **extends** or **implements** clause of a class or interface declaration. (However, a type variable may appear as the bound in an **extends** clause for another type variable.)

[3] The semantic definition of NextGen can easily be extended to include contravariant type variables, but the extension is difficult to support efficiently on top of the existing JVM. See footnote 6.

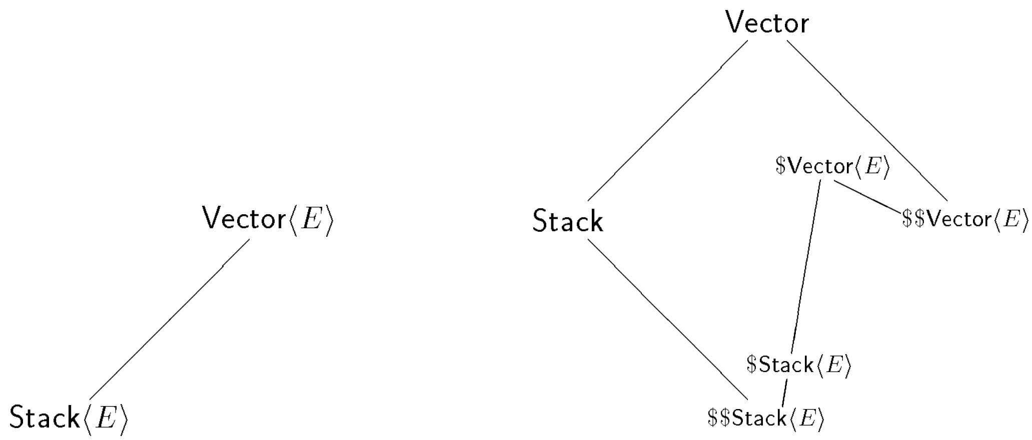

Figure 1: A Simple Parametric Type Hierarchy and its JVM Class Representation

A simple example of a parameterized class definition is the following parametric generalization of the familiar Vector class in the library java.util:

```
public class Vector⟨T⟩{
   private T[ ] elts;
   public Vector(int initCapacity) {
      elts = new T[initCapacity];
   }
   public T elementAt(int index) {
      return elts[index];
   }
   public void setElementAt(T newValue, int index) {
      elts[index] = newValue;
   }
   ...
}
```

Note that the type variable T is used in the type signatures of methods and the allocation of the array representing a vector. Clients of the parameterized Vector class can declare variables and allocate objects of type Vector$\langle E\rangle$ for any reference type E, as well as performing **instanceof** operations and casts for any type expression Vector$\langle E\rangle$.

We will discuss the syntax of parameterized method definitions and some additional annotations available in class definitions later in the paper.

The *semantics* of a parameterized class (interface) definition is completely explained schematically: a parameterized definition abbreviates all possible instantiations of the type variables by fully qualified *ground* (no type variables) types. Even instantiations involving inaccessible "package private" types are included in this expansion because inaccessible types can become indirectly accessible if they are bound to type variables. Of course, all of the ground types appearing in parametric type constructions must satisfy the bounds specified in the definition of the parametric type. In program text, parametric type expressions are composed from bound type variables and accessible parametric and non-parametric type names. The meaning of any such reference is simply the fully qualified ground type obtained by replacing each type variable by its binding expressed as a fully qualified ground type. The meaning of type casts and **instanceof** checks is determined by the type inclusion graph (a directed graph) specified by the subtyping relationships declared in class definitions (including the variance attributes of type parameters).

Like GJ and Pizza, NextGen uses a simple set of type-checking rules implied by F-bounded polymorphism [CCH+]. This static typing discipline dictates some syntactic restrictions on the form of parameterized classes with covariant type variables.[4] These restrictions are discussed in Section 6.

The primary costs of supporting this extension of Java are a modest restriction on the structure of Java programming environments and the code databases that they manage. In addition, NextGen generates some mangled names to represent run-time type information, similar to those used in the implementation of inner classes. JVM implementations are not affected.

3 Basic Implementation Strategy

If we could completely redesign the JVM to support parameterized types, we would extend the run-time representation of classes (the class Class) to support parameterized classes just as existing JVM supports arrays. While we would welcome the opportunity to extend the JVM, we can support a similar implementation of parameterized types *without changing the JVM*. Instead of relying on an enhanced JVM, we can require the compiler for NextGen to generate a lightweight "wrapper" class and interface for each instantiation of parameterized class used in a compilation unit. The wrapper class inherits type-erased methods from an abstract "base" class that is constructed from the parameterized class definition using type erasure. In essence, the wrapper class for each parametric class instantiation carries the type information for the instantiation.

Parameterized interfaces are supported in a similar fashion. For each parameterized interface definition, the NextGen compiler generates a "base" interface using type erasure and a

[4] Similar restrictions are required to support contravariant type variables.

corresponding "wrapper" interface (extending the base interface) for each instantiation appearing in a compilation unit. The wrapper interfaces do not introduce any new abstract methods.

The NextGen compiler must generate both a class and an interface for each parametric class instantiation because every Java class has at most one superclass. When one parameterized class extends another, *e.g.* Stack⟨T⟩ **extends** Vector⟨T⟩, each instantiation Stack$\langle E\rangle$ of the parameterized subclass has two supertypes: its base class Stack, from which it inherits nearly all of its members, and the corresponding instantiation Vector$\langle E\rangle$ of the parameterized superclass. We use the interface corresponding to a wrapper class to represent the type of the wrapper class. As a result, a wrapper class $$Stack$\langle E\rangle$ can extend its base class Stack and implement its own interface which implements the interface for the corresponding "super" wrapper class $$Vector$\langle E\rangle$. In the resulting type hierarchy, the class Stack$\langle E\rangle$ is a subclass of the base class Stack and a subtype of the interfaces for both Stack$\langle E\rangle$ and Vector$\langle E\rangle$. Figure 1 illustrates the JVM class hierarchy produced by NextGen for this example.

Given a parameterized class $C\langle T_0,\ldots\rangle$ in a package P, the NextGen compiler generates a type-erased base class C in P extending (the base class of) the (parameterized) supertype of $C\langle T_0,\ldots\rangle$.[5] All the attributes for each member of $C\langle T_0,\ldots\rangle$ are preserved in the base class. Static members must be treated specially as described in Section 2.2. For each instantiation $C\langle A_0,\ldots\rangle$ of the parametric class $C\langle T_0,\ldots\rangle$, the compiler generates

- an empty *wrapper* interface $\$C\langle A_0,\ldots\rangle$ (with the same visibility attributes as the parametric class $C\langle T_0,\ldots\rangle$) extending the wrapper interfaces of the immediate supertypes of $C\langle T_0,\ldots\rangle$,[6]
- a *wrapper* class $\$\$C\langle A_0,\ldots\rangle$ extending the base class C and implementing the corresponding wrapper interface $\$C\langle A_0,\ldots\rangle$.

If type variables do not occur in method bodies, the wrapper class $\$\$C\langle A_0,\ldots\rangle$ contains only "forwarding" (also called "trampoline") constructors (with erased signatures) that invoke the corresponding constructors from the base class using a superclass constructor call. (Since base class constructors must be accessible in wrapper classes, they must not be **private**.) This translation augments the homogeneous translation of Pizza and GJ by using wrapper classes and interfaces to provide run-time type information. Like Pizza and GJ, NextGen inserts casts where necessary to inform the Java Virtual Machine about the bindings of type variables.

NextGen supports the use of run-time type dependent operations in a method body by replacing each such operation by a call on a generated auxiliary method that can be appropriately overridden in each wrapper class. To construct the base class for a parametric class, the compiler performs type erasure and extracts each atomic operation involving a type variable (a **new** operation, cast, **instanceof** test, or polymorphic method invocation) and encapsulates it in a **protected** method (with a mangled name) called a *snippet*. In the base class, all snippets are **abstract**. Each wrapper class overrides the abstract snippets in the base class with specialized code to perform the appropriate type dependent operation.

NextGen places generated wrapper classes and interfaces in the same package P as the definition of the parametric class $C\langle T_0,\ldots\rangle$. This placement convention prevents the duplication of wrapper classes and interfaces, while giving wrapper classes access to *package-private* types and methods.

The form of wrapper classes is complicated by the fact that a type argument E_i in a parametric instantiation $C\langle E_0,\ldots\rangle$ may be restricted to some other package. In this case, the generated wrapper class W cannot directly perform snippet operations such as **new** $E_i(\ldots)$ because the type E_i is not accessible in P. We overcome this problem by having each *client* class D containing an instantiation $C\langle E_0,\ldots\rangle$ explicitly pass a *snippet environment* object containing the inaccessible snippet operations as part of class initialization for D. The affected snippet methods in the wrapper class simply forward their calls to the corresponding methods in the snippet environment. Note that a wrapper class may instantiate parametric classes using its inaccessible type arguments. In this case, the static initialization block for the wrapper class must pass the appropriate snippet environments to the affected wrapper classes for which it is a client. Each affected wrapper class must include a special method called $init that accepts the passed snippet environments and assigns them to corresponding fields in wrapper objects.

To support the independent compilation of client classes, the NextGen compiler must attach a `Snippet` attribute to the base class file for a parametric class $C\langle T_0,\ldots\rangle$. (The Java class file format allows compilers to embed arbitrary information within class files by attaching "attributes" to the class file.) This `Snippet` attribute tells the NextGen compiler (*i*) how to specialize the snippets in forming wrapper classes, (*ii*) which snippets are inaccessible when type variables are bound to invisible types, and (*iii*) what snippets must be passed to other classes by the static initialization blocks of wrapper classes. Since this initialization process for the wrapper class W is idempotent, it can safely be performed as each client class is loaded.[7]

3.1 Name Mangling

Since names containing pointy brackets are not valid JVM identifiers, NextGen encodes the name $C\langle E_0,\ldots,E_n\rangle$ as the valid JVM identifier $C\$\_e_0\$\ldots\$e_n\_\$$ where $e_0,\ldots,e_n$ are the JVM identifiers encoding the type expressions $E_0,\ldots,E_n$. If an array type $E[\,]$ appears within a type parameter, it is translated to array$$\_e\_$ where e is the JVM identifier encoding the type expression E. Hence Vector⟨Integer[ ]⟩ is encoded as the identifier Vector$_array$$_Integer_$_$.

[5] To support forward and backward compatibility with Java 1.2 libraries, NextGen uses a slightly different process to construct the base classes for parametric generalizations of standard library classes in java. .... This process is described in Section 2.8.

[6] NextGen excludes contravariant parameterization because separate class compilation and dynamic class loading make it impossible to determine all of the supertypes of a contravariant instantiation. At the point where a contravariant wrapper interface is loaded, the collection of supertypes is unknown. With JVM revision, this problem can easily be solved. In particular, the set of supertypes of a contravariant parametric type can be dynamically expanded as they are loaded.

[7] A simple variation on this implementation technique can be used to translate Java inner classes to flattened form *without compromising the security of* **private** *variables from the enclosing class.* In the secure translation, each affected private variable of type T from the enclosing class is allocated on the heap as an array of T of length 1. These arrays are passed as extra arguments to the inner class constructors just like the reference to the enclosing instance. Since program text outside the scope of the enclosing class cannot refer to the affected private variables, the translation must augment the outer class with factory methods corresponding to the inner class constructors. The factory methods supply the extra parameters to the inner class constructors.

NextGen explicitly bans the special character $ from appearing in program identifiers. As a result, this translation ensures that generated mangled names do not collide with program identifiers.[8] In the sequel, we use mangled names in generated program text, but retain the more readable notation using pointy brackets everywhere else.

3.2 A Sample Translation

As a simple example of the translation performed by NextGen, consider the problem of replacing the existing Vector library class by an equivalent class parameterized by element type. For the sake of brevity, the example does not include all of the methods from the actual Vector class.[9] For the moment, we will ignore the translation of the unary constructor for Vector.

The parameterized class Vector⟨T⟩ is translated by the NextGen compiler into a class file for the following *base* class:

```
public abstract class Vector {
   private Object[ ] elts;
   public Vector(int initCapacity) {
      ...  // see discussion below
   }
   public Object elementAt(int index) {
      return elts[index];
   }
   public void setElementAt(Object newValue, int index) {
     elts[index] = newValue;
   }
   ... // and a snippet -- see below
}
```

augmented by a `Signature` attribute describing the parametric signature of the class:

```
class Vector⟨T⟩{
   public Vector(int initCapacity);
   public T elementAt(int index);
   public void setElementAt(T newValue, int index);
}
```

The NextGen compiler uses the `Signature` attribute to generate new wrapper classes and interfaces for instantiations of the parameterized class. For each instantiation Vector⟨A_0⟩ of the parameterized class Vector⟨T⟩, the NextGen Compiler generates class files defining

- an empty *wrapper* interface $Vector⟨$A_0$⟩ extending Object (since Object is the only immediate superclass of $Vector⟨$A_0$⟩ in the parametric type hierarchy)[10] and
- a *wrapper* class $$Vector⟨A_0⟩ that extends Vector and implements $Vector⟨$A_0$⟩.

For example, given the class instantiation Vector⟨Integer⟩, the NextGen compiler generates class files for the following:

```
public interface $Vector$_Integer_$ {}
```

```
public class $$Vector$_Integer_$ extends Vector
      implements $Vector$_Integer_$ {
   public $$Vector$_Integer_$(int initCapacity) {
      super(initCapacity);
   }
   ... // and a snippet -- see below
}
```

With the exception of snippets discussed below, wrapper classes simply inherit *all* the methods of the base class without overriding them and replicate base class constructors by "forwarding" constructor calls to the base class using **super**. For example, the wrapper class $$Vector⟨Integer⟩ inherits the methods elementAt and setElementAt and defines forwarding constructors corresponding to those in the base class Vector. (Like Pizza and GJ, NextGen relies on the compiler and not the byte-code verifier to enforce most parametric type constraints.)

We deferred giving the translation of the constructor in the Vector base class above because the constructor invokes an operation that depends on the type variable T. To accommodate the specialization of this operation in each wrapper class, the translation must introduce an **abstract** snippet $snip$1 for that allocation operation:

```
public abstract class Vector {
   ...
   public Vector(int initCapacity) {
     elts = $snip$1(initCapacity);
   }
   abstract protected final Object[ ]
      $snip$1(int initCapacity);
   ...
}
```

which is subsequently defined in each wrapper class, *e.g.*,

```
public class $$Vector$_Integer_$ extends Vector
      implements $Vector$_Integer_$ {
   ...
   protected Object[ ] $snip$1(int initCapacity) {
      return new Integer[initCapacity];
   }
}
```

3.3 Performance and Optimization

The NextGen implementation architecture should produce good performance across a wide variety of Java Virtual Machines. The type erasure process that NextGen uses to construct base classes generates essentially the same code for parameterized classes as the corresponding unparameterized code written using the generic coding idiom. Pizza has already demonstrated that this form of genericity does not adversely affect Java performance.

NextGen augments the type erasure model with lightweight wrapper interfaces and classes, adding an extra method call in object initialization (invoking the "forwarding" constructor) and increasing the number of class files in a program, slightly enlarging its code footprint. For most programs, these differences should not have much effect on program performance except for a modest increase in load-time latency. For parametric classes containing type dependent operations, NextGen also augments the type erasure model with snippets. If the absence of code optimization, each type dependent operation requires an additional "dynamic" method call to select the appropriate snippet code.

Fortunately, high performance Java Virtual Machines inline most small **final** methods as part of either "just-in-time"

---

[8] We are glossing over the problem of name clashes with mangled names generated by the translation of inner classes. Such conflicts are avoided by more severe name mangling using sequences of $ characters.

[9] This discussion also assumes that package java.util containing `Vector` is an ordinary package under the compiler's full control. In fact, library packages like java.util are *sealed* (immutable) in many JVM's. We will discuss how NextGen handles this complication in Section 4.

[10] Base classes are used in the implementation of NextGen on top of the existing JVM, but they play no role in the semantics of NextGen.

or "dynamic" compilation of Java byte code. Since snippets are small **final** methods, snippet call sites that are repeatedly executed will generally be inlined, eliminating nearly all of the overhead produced by snippets. The same inlining optimization will also eliminate the extra "constructor" calls in object initialization.

4 Parametric Type Translation

NextGen relies on the process of type erasure developed by Odersky and Wadler for Pizza and GJ. GJ is translated to Java 1.2 by type erasure, which replaces *isolated* type variables[11] by their upper bounds (typically Object) and translates all parametric instantiations to the corresponding base classes. In any context, where the generated Java code requires erased type information for type correctness, GJ inserts an appropriate cast to restore the missing information. For example, in GJ, the hypothetic method (in the parameterized Vector class shown above)

```
int add(Vector⟨Integer⟩v) {
   int sum = 0;
   for (int i = 0;i < v.length;i++)
      sum += v[i].intValue();
}
```

would translate to:

```
int add(Vector v) {
   int sum = 0;
   for (int i = 0;i < v.length;i++)
      sum += ((Integer) v[i]).intValue();
}
```

Parametric type translation in NextGen is similar but more complex because NextGen supports parametric types in *any* context; all objects of parametric type carry run-time type information.

The first stage of type translation for NextGen mimics type erasure for GJ. NextGen processes the body of a class by replacing isolated type variables in method signatures and variable declarations by their upper bounds, converting the parametric instantiations appearing in these contexts (method signatures and variable declarations) to the corresponding base classes, and inserting casts where erased type information is required by the enclosing context.[12] But this translation is incomplete because (i) type variables can appear in executable code, (ii) the state corresponding to static variables must be replicated, and (iii) static inner classes have independent extent forcing an independent translation.

For the moment, let us ignore static inner classes and static variables. Let us also assume that parametric types embedded in executable operations *do not contain type variables*, *e.g.*, **new** Stack⟨Number⟩(), but not **new** Stack⟨T⟩() where T is a type variable. Then there are four contexts that we must consider:

1. Each **new** operation involving a parametric type is converted to a **new** operation for the corresponding wrapper class (or array of the designated wrapper *interface*). Hence,

   new Stack⟨Number⟩(25)

   becomes

   new $$Stack$_Number_$(25)

   and

   new Vector⟨Boolean⟩[100]

   becomes

   new $Vector$_Boolean_$[100].

2. Each casting operation to parametric type is converted to a sequence of two casts: a cast to the corresponding wrapper interface (or array of the appropriate wrapper interface), followed by a cast to the matching base class.[13] Hence, the operation

   (Vector⟨Integer⟩) x

   becomes

   (Vector) (($Vector$_Integer_$) x)

   and

   (Stack⟨Number⟩[ ]) x

   becomes

   (Stack[ ]) (($Stack$_Number_$[ ]) x).

3. Each **instanceof** test involving a parametric type is converted to an **instanceof** test on the corresponding wrapper interface (or array of wrapper interface). Hence,

   instanceof Vector⟨Boolean⟩

   becomes

   instanceof $Vector$_Boolean_$.

4. Each parametric class instantiation $C\langle E_0, \ldots\rangle$ that is used to access a static class member is converted to the corresponding wrapper class $$$C$\_$e_0$$\ldots$_$.[14]

For example, NextGen translates the following code fragment

```
Vector⟨Number⟩x = new Vector⟨Integer⟩(100);
for (int i = 0;i < 100;i++)
   x.setElementAt(new Integer(i), i);
if (x instanceof Vector⟨Integer⟩) ...
```

to:

```
Vector x = new $$Vector$_Integer_$(100);
for (int i = 0;i < 100;i++)
   x.setElementAt(new Integer(i), i);
if (x instanceof $Vector$_Integer_$) ...
```

If type variables appear in method bodies, then this simple translation strategy is insufficient, because the generated code depends on the bindings of these variables. Essentially the same problem arises in the translation of static variables with types dependent on type variables. The *snippet* technique used to solve both of these problems is described in the next subsection.

[11] Type variables not embedded within parametric instantiations.

[12] GJ also erases the parametric type information in operations of the form **new** $C\langle E_0, \ldots\rangle$; this does not work for NextGen because every object of parametric type must be recognizable as a member of the parametric type, not just a member of the base class type.

[13] Two casts are necessary for class types because the wrapper interfaces for classes are *empty*.

[14] In the absence of static variables and type variables in executable code, there is no difference between the meaning of static methods in wrapper classes and the base class, but the distinction matters when these restrictions are dropped.

4.1 Type Variables in Method Bodies

Simple type erasure cannot accommodate method code containing type variables because the behavior of such code depends on the bindings of the type variables. To accommodate this variation in behavior, the translation introduces a package-private auxiliary method, called a *snippet*, in the base class for each operation involving type variables. Wrapper classes then override snippets as appropriate.

Simple snippets are insufficient for some parametric instantiations involving types that are not **public**, because the type required in a snippet operation may be inaccessible in the package containing the parametric class definition. For example, assume that the source code library java.util for NextGen includes the parameterized Vector class given above and that NextGen is given the following code to translate:

```
package Client;
import java.util.*;
class Name { ... }  // not a public type
class Main {
   static Vector⟨Name⟩vn = new Vector⟨Name⟩(100);
   ...
}
```

The instantiation Vector ⟨ Name ⟩ in class Main forces the construction of a wrapper class and interface for Vector⟨Client.Name⟩. The constructed wrapper class must include a snippet that allocates an array of type Client.Name but Client.Name is not visible in the library package java.util where the wrapper class is placed. In this example, we could solve the problem by placing the constructed wrapper class in the Client package, but this remedy fails if the wrapper class code refers to a non-**public** class C of package java.util.[15] Hence, we must place the constructed wrapper package in the same package as the corresponding base class and devise a general solution to the "hidden snippet" problem.

To make the **new** Name[...] operation available in the library, the Client package passes a *snippet environment* to the constructed wrapper class that includes the requisite allocation operation. This visibility problem can occur for any "isolated" snippet that appears in a public parameterized class. The only isolated snippets for the type variable T are the operations **new** $T(\ldots)$, $(T)\ \ldots$, and and their analogs for the array types $T[\ ]$, $T[\ ][\ ]$, $\ldots$.[16] All non-isolated snippets are visible in the package containing the parameterized class definition because they involve parameteric types of the form $C\ \langle T, \ldots \rangle$ where C is visible.

The simplest general strategy for solving this visibility problem is to embed all of the isolated snippets for a wrapper class in a *snippet environment* object that is constructed in each client class and passed to the corresponding wrapper class as the *first* step in the static initialization of the client class. Each isolated snippet forwards its calls to the corresponding method in the snippet environment. Each wrapper class W for a parameterized class containing isolated snippets includes the usual forwarding constructors, a private static variable $is of type $IsolatedSnippets to hold the passed snippet environment, a generated method $init($IsolatedSnippets is) that initializes $is, and snippet definitions for both isolated and non-isolated snippets. Non-isolated snippets are defined as before; the appropriate type-specific operation is guaranteed to be in scope. In contrast, each isolated snippet is defined to forwards calls to the corresponding method in the snippet environment $is. The type $IsolatedSnippets is a public abstract inner class in the package containing the parameterized class; the class specifies method signatures for the isolated snippets. The $init method for W initializes $is and invokes the $init methods for the wrapper classes with isolated snippets that W directly calls.[17] Since the static variable $is does not change after it is initialized, the initialization operation for a wrapper class is idempotent and can be performed every time a calling class using W is loaded.

The following code contains the generated base classes for Vector⟨T⟩ and the automatically constructed wrapper class $$Vector⟨Name⟩:

```
package java.util;
abstract public class Vector {
   // snippet environment class
   public abstract static class $IsolatedSnippets {
      Object[ ] $1(int initCapacity);
   }
   private Object[ ] elts;
   public Vector(int initCapacity) {
      elts = $snip$1(initCapacity);
   }
   // snippet
   protected abstract Object[ ] $snip$1(int initCapacity);
   public Object elementAt(int index) {
      return elts[index];
   }
   public void setElementAt(Object newValue, int index) {
      elts[index] = newValue;
   }
   ...
}
public class $$Vector$_Name_$ extends Vector {
   // snippet environment
   static Vector.$IsolatedSnippets $is = null;
   public static void $init($IsolatedSnippets $is) {
      this.$is = $is;
   }
   $$Vector$_Name_$(int initCapacity) {
      super(initCapacity);
   }
   protected Object[ ] $snip$1(int initCapacity) {
      return $is.$1();
   }
   ...
}
package Client;
import java.util.*;
class Name { ... }  // not a public type
class Main {
   static { $$Vector$_Name_$.
      $init(new Vector.$IsolatedSnippets() {
         Object[ ] $1(int initCapacity) {
            return new Name[initCapacity];
         }
      })
   }
   static $$Vector$_Name_$ vn =
      new $$Vector$_Name_$(100);
   ...
}
```

[15] Which happens if the package private class C is parameterized by T and the wrapper class includes a snippet allocating an instance of type $C\langle \$T\$ \rangle$.

[16] Some **new** operations for arrays are implicit in the program syntax.

[17] Note that the calling class will pass snippet environments for all the classes reachable from W.

4.2 Accommodating Static Members

According to NextGen's schematic semantic model for parameterized classes, the static members of a parameterized class must effectively be replicated in every wrapper class. NextGen employs an optimized version of this translation that generally avoids replicating static method code.

Since the static variables of a parameterized class are replicated in each instantiation, all operations involving them must be converted to snippets so that the code for these operations appears in the proper lexical scope. In particular, each access to a static variable x must be expressed in fully qualified form—including the appropriate type parameters. For example, if a static variable x is defined in a parameterized class $C\langle T_0, \ldots\rangle$, any access to x inside of C has fully qualified form $C\langle T_0, \ldots\rangle$.x. Since the meaning of such an access depends on the bindings of the type variables $T_0, \ldots$, NextGen converts these operations to snippets (*getter* methods in this case) which must be appropriately defined in each wrapper class. Note that the types of static variables must be erased in wrapper classes because the snippets appear in the base class signature.

The replication of static method code in each wrapper class can be avoided as follows. If a static method in a parameterized class C does not depend on type variable bindings,[18] then NextGen translates it to its type erasure in the base class. Otherwise, NextGen translates the static method to an explicitly parameterized static method with exactly the same list of type variables as C (including implicit type variables if C is an inner class). The implementation of explicitly parameterized methods is described in Section 6.

Static inner classes are supported within parameterized classes by generalizing the simple syntactic transformation used to support static inner classes in Java 1.1. Each static inner class A embedded in a parameterized class $C\langle T\rangle$ is transformed to a class $C\$A\langle T\rangle$ defined in the same lexical scope as the parent class $C\langle T\rangle$.[19] The transformation also converts each implicit or explicit reference $C\langle T\rangle.A$ anywhere in the program to the corresponding reference $C\$A\langle T\rangle$.

4.3 Wrapper Classes and Interfaces

When a parameterized class C extends another parameterized class (interface) D, the extension establishes a mapping from instantiations of C to instantiations of D, which can be represented as a substitution θ binding the type variables of D to type expressions containing the type variables of C. Each instantiation of a parameterized class is represented by a binding of the class variables to ground types. Given a binding ϕ of the type variables of C to ground types, composing θ with ϕ yields the corresponding binding of the type variables of D to ground types.

If the class (interface) D is a ground instantiation (contains no type variables), then the base class for C extends the corresponding wrapper class (interface). Otherwise, the base class for C simply extends the base class (interface) for D; the substitution θ is left pending until the compiler generates wrapper classes and interfaces for C.

When the parameterized class C is instantiated as a ground type $C\langle E_0, \ldots\rangle$, NextGen generates (i) a wrapper interface $\$C\langle T_0, \ldots\rangle$ extending the wrapper interfaces for the corresponding instantiations of the classes and interfaces that C extends, and (ii) a wrapper class $\$\$C\langle E_0, \ldots\rangle$ extending the base class C and implementing $\$C\langle T_0, \ldots\rangle$. In the wrapper class $\$\$C\langle T_0, \ldots\rangle$, the type dependent operations performed by any snippets inherited from D must reflect the substitution θ. These substitutions can cascade because snippets in class D may have been inherited from a parameterized superclass.

In the common case, the substitution θ is trivial. For example, in the java.util library, the class java.util.Stack extends java.util.Vector. If we parametrically generalize Vector, we should do the same for Stack:

public class Stack⟨A⟩ **extends** Vector⟨A⟩{ ... }

The compiler for NextGen will expand this into the base class definition:

public class Stack **extends** Vector { ... }

and, as needed, wrapper classes and interfaces of the form

```
public interface $Stack$_..._$ extends $Vector$_..._$;
public class $$Stack$_..._$ extends Stack
  implements $Stack$_..._$
{ /* forwarding constructors ... */ }
```

When a parameterized class like Stack⟨T⟩ extends a parameterized class like Vector⟨T⟩, each wrapper class $$Stack⟨E⟩ has two superclasses: the base class Stack and a wrapper class $$Vector⟨E⟩. In the implementation, this relationship is reflected by the fact that the wrapper class $$Stack⟨E⟩ extends the base class Stack and implements the wrapper interface $Stack⟨$E$⟩, which extends the wrapper interface $Vector⟨$E$⟩. Hence, any value of class $$Stack⟨E⟩ inherits methods from the base class Stack, yet belongs to the type $Vector⟨$E$⟩ (because it **implements** $Stack⟨$E$⟩, which is a subtype of $Vector⟨$E$⟩). Any value of class $$Stack⟨E⟩ also belongs to type Vector (because $$Stack⟨E⟩ **extends** Stack which **extends** Vector).

4.4 Parameterized Interfaces

Parameterized interfaces are implemented in almost exactly the same fashion as parameterized classes. The only difference is that the instantiation of a parameterized interface generates a single wrapper interface instead of a wrapper class and interface. The generated wrapper interface extends the base interface and its immediate (possibly parameterized) supertypes. Since base interfaces only contain abstract methods, the wrapper interfaces generated for instantiations of parameterized interfaces include all of the methods of the parameterized interface. As a result, there is no need to cast expressions of wrapper interface type to base interface type before invoking a method in the base interface signature; the wrapper signature and base interface signature are identical.

The only significant complication in implementing parameterized interfaces is the treatment of **static final** fields. If the value of a **static final** field is type dependent, then it must be replicated in each wrapper interface extending the base interface. But wrapper interfaces constructed for the various parametric instantiations cannot include snippets to get the local version of the **static final** field. The only code that can appear in an interface is initialization code for each **static final** field. If a parameterized interface I contains a type dependent **static final** field F, the NextGen compiler adds an abstract snippet **public** Object \$get_F(); to the base interface

[18] As demonstrated above, dependence on static variables implies dependence on type variable bindings.

[19] As we observed in footnote 6, there is a more secure syntactic transformation to support inner classes than the one adopted in Java 1.1. For the sake of uniformity, NextGen follows the Java 1.1 convention.

I and inserts a copy of the field F together with appropriate initialization code in each wrapper interface extending I. The snippet $get_$F$ must be implemented (overridden) in each wrapper *class* that directly implements a wrapper interface extending I. The compiler records the snippet information for an interface in the `Snippet` attribute in its class file.

4.5 Type Erasure Complications

NextGen relies on type erasure to support variables[20] and method parameters of parametric type. Occurrences of parametric types in type casts and **instanceof** checks cannot be translated to the corresponding wrapper class types, because the JVM restricts classes to a single supertype (single inheritance), yet parametric instantiations like Stack⟨Integer⟩ have two supertypes as shown above.[21] Given the class Stack⟨T⟩ extending Vector⟨T⟩ defined above, an instance of the class Stack⟨Integer⟩ can be assigned to a variable of type Vector⟨Integer⟩, yet the wrapper class $$Stack⟨Integer⟩ is not a subclass of the wrapper class $$Vector⟨Integer⟩. NextGen pairs every *wrapper class* $$$C$ with a corresponding *wrapper interface* $$C$ to solve this problem.

Type erasure introduces another complication when a class (interface) C extends a ground instantiation B_0 of a parameterized class (interface). The method signatures in B_0 are erased, because they are inherited from the base class B. Hence, they must be erased in the extending class C. But C may override a method m from B_0 with an erased signature by a method m' with an unerased signature. Worse, the class C may implement another interface I with method signatures matching methods in the *unerased* signature of B_0. Consider the following code fragment in NextGen:

```
public interface CompareTo⟨T⟩{
   public int compareTo(T other);
   // returns neg,0,pos if this ⟨,=,⟩ other
}
public class String implements CompareTo⟨String⟩{
   ...
   public int compareTo(String other) {
      ... other ...
   }
}
```

According to the translation process described above, a NextGen compiler will translate this fragment to a Java code fragment of the following form

```
public interface CompareTo {
   public int compareTo(Object other);
}
public interface $CompareTo$_String_$
      extends CompareTo {}
public class String implements $CompareTo$_String_$ {
   public int compareTo(String other) {
      ... other ...
   }
}
```

This code fragment is ill-formed because String does *not* implement the interface $CompareTo$_String_$; there is no method matching the signature of compareTo in the interface CompareTo.

To solve this problem, NextGen borrows a technique from Pizza. Pizza reconciles this signature conflict by adding an overloaded version of the method compareTo.

[20] Except static variables, which are replicated.

[21] Covariant parameterization introduces additional supertypes. See Section 6.

```
public class String implements CompareTo$_String_$ {
   public int compareTo(String other) {
      ... other ...
   }
   public int compareTo(Object other) {
      ... ((String) other) ...
   }
}
```

The overloaded version of the method casts every reference to other to type String. Static type checking ensures that these casts can never fail.[22]

In general, given a class C that directly or indirectly extends (implements) a ground instantiation B_0 of a parameterized class (interface), NextGen will generate an overloaded method m' (called a *bridge* method) with instantiated signature for each method m with an erased signature in B_0 that is not so overloaded in the superclass of C. If the method m (with instantiated signature) is not defined in class C, the generated method m' simply forwards its calls to the inherited method m with erased signature, casting the output of m' to an instantiated type, if necessary. If C defines m (with instantiated signature), NextGen translates m to a method m' with an instantiated signature (no erasure). In this case (which is illustrated in the example above), the translation of C must include a method implementing or overriding m (with erased signature) that simply calls m', inserting casts where necessary. Note that m and m' always have the same semantics except for run-time type checks dictated by their signatures.[23]

The translation of parametric types to base types conflicts with static overloading because syntactically distinct type signatures may collapse to identical signatures after type translation (erasure). There are two possible solutions to this problem. First, overloading parametric signatures can be restricted to eliminate the conflict. Second, NextGen could mangle the names of operations that would otherwise conflict. When more than one signature matches a method invocation, NextGen could choose the more specific (which might mean "less parameterized") of the two—generalizing an existing rule of Java. We favor the first solution, at least initially, for the sake of simplicity of implementation.

4.6 Parametric Type Spoofing

In an earlier parameterized type proposal for Java, Agesen *et al.* observed that untrusted JVM code can breach the parametric components of program typings in homogeneous implementations[24] such as Pizza and GJ. The implementation that we have described above for NextGen suffers from the same problem because it does not check at run-time that method inputs and outputs obey the specified parametric type constraints. We can modify the implementation to perform the necessary checks, but these checks can add significant overhead. The spoof-resistant translation is complicated by the fact that untrusted code can extend wrapper classes, possibly overriding methods containing parametric checks.

[22] Assuming that all of the executed Java code is produced by the NextGen compiler.

[23] The introduction of bridge methods can introduce an overloading anomaly: the signature of the two versions of a method may differ only in their result type, which is not permitted in Java source code. Fortunately, as Odersky and Wadler have observed [OW97,BOSW97], the JVM supports this form of overloading, so no "workaround" is necessary.

[24] An implementation of type parameterization is homogeneous if it erases parametric type information in object representations.

In the spoof-resistant translation, every base class method with an erased parametric signature is overridden by a "type-secure method" that performs the required checks and invokes the base class (bound to **super** in the implementation). An untrusted extension of a wrapper class can override any non-final method, implying that it can breach parametric type constraints enforced by non-final "type-secure methods". To prevent such overriding, the translation process

- marks each "type-secure method" as **final**,
- forwards calls on type-secure methods (using **super**) to equivalent calls on new **protected** auxiliary methods (with mangled names) that simply invoke their erased analogs in the base class,
- generates the required auxiliary methods, and
- performs exactly the same name mangling on overriding methods in any extension of a class instantiation.

This elaborated translation allows only the "core" of parametric methods (encapsulated in the generated auxiliary methods) to be overridden—not the checks enforcing parametric type constraints. Consider the elaborated translation of Vector⟨Integer⟩ below:

```
public class $$Vector$_Integer_$ extends Vector
    implements $Vector$_Integer_$ {
  public $$Vector$_Integer_$(int initCapacity) {
    super(initCapacity);
  }
  protected Integer $elementAt(int index) {
    return super.elementAt(index);
  }
  public final Object elementAt(int index) {
    return (Integer) $elementAt(index);
  }
  protected void $setElementAt(Integer newValue, int index) {
    super.setElementAt(newValue,index);
  }
  public final void setElementAt(Object newValue, int index) {
    $setElementAt( (Integer) newValue, int index);
  }
}
```

Given a class V that extends Vector⟨Integer⟩, the compiler must mangle the names of any overridden wrapper methods.

The overhead of the additional method calls and large class file footprints in a spoof-resistant implementation would be substantial. This overhead can be reduced by performing optimizations that safely eliminate some of these checks in trusted code. For example, wrapper classes can be augmented with private "shadow" versions of methods without run-time checks on their inputs so that calls within the class can bypass the checks.

In the remainder of this proposal, we will *not* include the elaborations required to prevent parametric type spoofing, because it would complicate the exposition. Moreover, we believe that the best way to prevent parametric type spoofing is to modify the JVM to support parametric types directly.

4.7 Non-parametric Extensions of Parametric Types

When a non-parametric class (interface) C extends or implements a parametric class or interface, the new class (interface) C simply extends the appropriate wrapper class (interface).

For example, consider the following extension of the class Vector⟨Integer⟩:

```
class SummedVectorInteger extends Vector⟨Integer⟩{
  public int sum = 0;
  public SummedVector(int initCapacity) {
    super(initCapacity);
  }
  public void setElementAt(Integer newValue, int index) {
      sum = sum + newValue.intValue() −
            elts[index].intValue();
      ...
  }
}
```

The NextGen compiler must transform this class to the form

```
class SummedVectorInteger extends $$Vector$_Integer_$ {
  public int sum = 0;
  public SummedVector(int initCapacity) {
    super(initCapacity);
  }
  public void setElementAt(Integer newValue, int index) {
      sum = sum + newValue.intValue() −
            elts[index].intValue();
      ...
  }
}
```

5 Compatibility with GJ and Java 1.2

NextGen is a strict extension of type-sound GJ with the exception of the reflection API and the treatment of static class members. The differences at the level of reflection are unavoidable because the reflection API reveals all the unprotected details of the class hierarchy including class fields and their attributes. Even a tiny change to a Java class library (such as dropping a **final** modifier for a public method) is not conservative for programs that use reflection.

In contrast, the differences involving the interpretation of static members are easy to reconcile. NextGen can easily support both shared and replicated static members and interpret the unadorned syntax as the shared version. The implementation of shared members is straightforward: the field or inner class in question is associated with the base class instead of the wrapper class. The biggest challenge in designing this extension is devising an appealing syntax for declaring replicated static members *without introducing a new keyword*. One possible choice[25] is to prefix replicated static members by the augmented keyword **static**⟨ ⟩ instead of **static**.

The primary disadvantage of adding shared static members to NextGen is that it complicates the semantic model. Shared static variables cannot be explained schematically. In addition, static parameterized inner classes and static parameterized methods can be defined in two different ways that are not equivalent.

5.1 Legacy Code and Libraries

When NextGen translates the parametric generalization $C\langle T_0, \ldots\rangle$ of an existing Java 1.2 library class java. ... $.C$, it cannot "place" the generated base class in the package java. ... unless it exactly matches the signature and semantics of the original class. In many JVM's, the standard package library packages java. ... are *sealed* (immutable). Hence, parametric

[25] Suggested by Phil Wadler.

generalization must be implemented using inheritance (leaving the original library package undisturbed), rather than code modification.

To compile parametric generalization $C\langle T_0, \ldots\rangle$ of a class C in a sealed package, NextGen (i) generates the usual base class for $C\langle T_0, \ldots\rangle$, and (ii) extracts the "parametric" operations from this base class (the generated snippets and methods that directly invoke them) and places them in a special abstract base class generic.java. ... .C extending the original library class java. ... .C. The remaining members, which presumably match those in the original library class, are discarded. Obviously, no **final** methods within a class can be parametrically generalized. Similarly, no type dependent method in a parametric generalization can access a **private** variable or invoke a **private** method. Fortunately, all **final** attributes have been dropped from the core library classes in Java 1.2. This translation does not support shared **static** variables in parametrically generalized classes, but such support could easily be added at the cost of adding *shared* static variables to NextGen.

5.2 Interoperability

NextGen differs from GJ in that instances of parametric classes are *not* instances of the base class. In essence, new data is distinct from old data. In general, new data can be used in place of old data, but not vice versa. Hence, programs that intermix old and new data must explicitly convert old data to the new form. The parametric generalization of java.util could include parameterized methods that perform these conversions.[26]

In general, mixed programming involving both old and new data is poor software engineering practice. Old code should be revised to use parametric types. But when source for old code is unavailable, "mixed data" programming may be unavoidable. The ability to mix old and new code also provides a smooth transition path during code conversion. NextGen, following in the footsteps of GJ, supports such mixed programming. NextGen, however, generally requires more "glue code" between new code and old code than GJ does. This is an *unavoidable* cost of *type soundness*. GJ avoids the use of glue code in some contexts by compromising type soundness: an "old" value of unparameterized type List can be assigned to a "new" variable x of parametric type List⟨Integer⟩. If the old List value contains non-Integer elements, then a subsequent operation on x may generate a run-time type error. In contrast, NextGen enforces the type constraints given in program declarations. It will not permit old data of type List to be treated as instances of any parametric type List$\langle T\rangle$. The old data must be converted to new parametric data to be processed by code that operates on parametric data.

6 Parameterized Inner Classes

Static parameterized inner classes can be implemented in essentially the same fashion as conventional (unnested) parameterized classes because static inner classes have the same semantics as conventional classes except for qualified names. Each static parametric inner class $A\langle T\rangle$ embedded in a class C is translated to an unnested class $C\$A\langle T\rangle$ and every implicit or explicit reference $C.A\langle E\rangle$ to an instantiation of the inner class is translated to $C\$A\langle E\rangle$.

The implementation of "dynamic" parameterized inner classes is a more challenging problem because each instance of a dynamic inner class must provide access to the "enclosing" instance. The NextGen compiler translates a parameterized dynamic inner class to an erased base class in exactly the same way that it translates conventional parametric classes to base classes. But the compiler must also extend this inner base class with textually independent wrapper classes. Fortunately, the inner class facility introduced in Java 1.1 supports exactly this form of inner class extension. An accessible dynamic inner class C can be extended by a class C' declared at the top lexical level. Since the containing instances for any instantiation of such a class are not implicit from the text, the constructor for C' must explicitly install the innermost containing instance as its first operation. This instance is typically passed as an extra parameter to the constructor for C'.

Given a parameterized dynamic inner class $C\langle T_0, \ldots\rangle$ embedded in the class D, the NextGen compiler inserts a parameter of type D at the beginning of the parameter list for each constructor in wrapper classes. In the translation of each constructor call for an inner class $C\langle T_0, \ldots\rangle$ NextGen binds the inserted parameter to the enclosing instance of class D. In other words, each operation **new** $C\langle E_0, \ldots\rangle(\ldots)$ in the original source becomes **new** $C\$\_e_0\$\ldots\_\$$(**this**,...) in the translation (where e_0 is the translation of the type expression E_0). Similarly, each operation p.new $C\langle E_0, \ldots\rangle(\ldots)$ in the original source (where p is an an instance of class D) becomes **new** $C\$\_e_0\$\ldots\_\$(p, \ldots)$.

If a (static or dynamic) parameterized inner class C is defined within a parameterized class D, NextGen generates a single base class for the inner class C, but C is indirectly parameterized by the type variables of D. The wrapper classes for C must make this parameterization manifest to accommodate the various snippets (which can depend on indirect as well as direct type variables) and to replicate static variables.

7 Parameterized Methods

NextGen implements parameterized methods using essentially the same techniques that it uses to support parameterized classes. But the details of the implementation are more complex because parameterized methods can be inherited and overridden in subclasses.

Static parameterized methods are easy to implement because no inheritance is possible. Every static parameterized method m in a class C is explicitly invoked using the class name C.[27] If a static parameterized method m does not depend on type variable bindings, then NextGen simply translates m to its type erasure in the class C (or the base class for C if C is parametric). Otherwise, NextGen converts the type dependent operations in body of m to snippets, generates a static abstract inner class A in C (with the same visibility as m) defining the snippet environment signature, translates m to snippeted form in C with an additional argument of type A (the passed snippet environment), and inserts an appropriate anonymous class instance construction as an additional argument in each call on m. The resulting form of the method and method invocations is called *snippet-passing* style. A NextGen compiler can easily optimize construction of anonymous

[26] Since the new methods are type dependent, NextGen will place them in the special package generic.java.util leaving the binary form of java.util unchanged.

[27] Or a variable whose static type specifies the class.

class instances so that no instance is constructed more than once.

The implementation of dynamic parameterized methods is complicated by the fact that such methods can be overridden in subclasses. Since method dispatch is dynamic, a method invocation site x.m<T>(...) can execute different method bodies depending on the receiver x. As a result, dynamic parameterized methods *cannot* be translated to degenerate parameterized inner classes containing a *single* method [OW97]. This translation fails in the presence of method overriding because the inherited parameterized inner class *cannot* be overridden; inner classes are not *virtual*.

The same pathological situation prevents NextGen from translating a parameterized method directly to snippet-passing style. If a parameterized method is overridden, the set of requisite snippets may change. Hence, in the presence of dynamic dispatch, NextGen cannot determine which snippet environment to pass. A given call site may need to pass different snippet environments in different executions.

To cope with method overriding in the context of dynamic dispatch, NextGen augments class files containing overridable, dynamic parameterized methods with *wrapper methods* as demanded by client classes. Since some Java packages are *sealed* (immutable by client code), the parameterized methods in sealed packages must be either `static`, `private`, or both `final` and *original* (not overriding a superclass method) to rule out the possibility of overriding dynamic parameterized methods. In the special case where a dynamic parameterized method is either `private` or both `final` and *original*, NextGen uses the same snippet-passing transformation as it does for static parameterized methods.

For conventional packages that are not sealed, NextGen translates overridable dynamic parameterized methods to snippet-passing style as follows. For each such method m in class C, NextGen first converts m to snippet-passing style and labels it as a `protected` method of C. This converted method, called the *base method* of the parameterized method definition, is never called directly by clients since its signature is not compatible with dynamic dispatch. For each distinct instantiation of a parameterized method in a program, NextGen generates a *wrapper method* in class C (or a wrapper class extending C if C is parametric) with a mangled name and a signature that exactly matches the erased signature of the parameterized method m. The body of the generated wrapper method simply forwards the call to the base method augmented by the appropriate snippet environment as an extra argument. Note that this strategy accommodates method overriding and dynamic dispatch because the wrapper method has the same signature in the overriding and overridden classes, but different code bodies with different snippet environments. To keep track of which wrapper methods have been generated for parameterized methods in the class C, the NextGen compiler attaches a `WrapperMethods` attribute to the class file.

The translation for overridable dynamic parameterized methods described above has one important limitation: it does not support binding the type variables appearing in *isolated* snippets[28] to types that are invisible in the package containing class C. The wrapper method for such a type application cannot define the required isolated snippets because the specified classes are invisible. We refer to such wrapper methods as *escaping* wrapper methods.

The translation of parameterized methods can be extended to accommodate escaping wrapper methods as follows. NextGen can augment the class C by an abstract static inner class $m\$A$ specifying the signature of the isolated snippet environment for m. Then for each escaping wrapper method wm that NextGen adds to class C, it also generates a static variable wm\$IsolatedSnippets of type $m\$A$ and a static initializer method wm\$Init that initializes wm\$IsolatedSnippets to the value specified by its argument. Each client class for the wrapper method wm must invoke the initializer wm\$Init as part of its static initialization. Note that if m is overridden in a subclass C' of C, every client of wm in C is a client of the corresponding escaping wrapper method wm in C'. To update all affected class files incrementally as individual classes are compiled, the NextGen compiler must attach a `Clients` attribute to each class file specifying the client classes for each escaping wrapper method in the class file. In addition, it must attach a `WrapperInits` attribute to each client class file specifying exactly which wrapper-method initializers that it invokes. This wrapper initialization portion of static initialization for each class can be encapsulated in a method \$wrapper\$Init so that it can be incrementally updated without affecting other static initialization code.

The implementation of parameterized methods in NextGen is illustrated by the following examples. Given the following program text defining the parameterized method zip:

```
package P;

public class Vector⟨A⟩{
   private A[ ] elts;
   public Vector(int initCapacity) {
      elts = new A[initCapacity];
   }
   public A elementAt(int index) {
      return elts[index];
   }
   public void setElementAt(A newValue, int index) {
      elts[index] = newValue;
   }
   public⟨B⟩Vector⟨Pair⟨A,B⟩⟩zip(Vector⟨B⟩other) {
      int s = this.size();
      if (s != other.size())
         throw new NoSuchElementException();
      Vector⟨Pair⟨A,B⟩⟩result = new Vector⟨Pair⟨A,B⟩⟩(s);
      for (int i = 0;i < s;i++)
         result.elts[i] = new Pair⟨A,B⟩(this.elts[i], other.elts[i]);
      return result;
   }
}

public class Pair⟨A,B⟩{
   public A fst;
   public B snd;
   public Pair(fst,snd) {
      this.fst = fst;
      this.snd = snd;
   }
}

package Q;
public class Main {
   public static void main(String args[ ]) {
      Vector⟨String⟩names = new Vector⟨String⟩(100);
      Vector⟨Integer⟩numbers = new Vector⟨Integer⟩(100);
      ...
      Vector⟨Pair⟨String,Integer⟩⟩phonebook =
         names⟨Integer⟩zip(numbers);
      ...
   }
}
```

[28] Recall the definition in Section 3.1.

the NextGen compiler

1. translates the class Vector⟨A⟩ to the following base class:

```
package P;
abstract public class Vector {
   // snippet environment
   public static abstract class $IsolatedSnippets {
      Object[ ] $1(int initial capacity);
   }
   private Object[ ] elts;
   public Vector(int initCapacity) {
      elts = $snip$1(initCapacity);
   }
   // snippet
   abstract protected Object[ ] $snip$1(int initCapacity);
   public Object elementAt(int index) {
      return elts[index];
   }
   public void setElementAt(Object newValue, int index) {
      elts[index] = newValue;
   }
   public abstract class $zip$Snippets = {
      Vector $1(int l);
      Pair $2(Object fst, Object snd);
   }
   protected Vector$_Pair$_Object$Object_$_$
      zip($zip$Snippets $is, Vector$_Object_$ other) {
         int s = this.size();
         if (s != other.length())
            throw new NoSuchElementException();
         Vector result = $is.$1(s);
         for (int i = 0;i < s;i++)
            result.elts[i] = $is.$2(this.elts[i], other.elts[i]);
      return result;
   }
}
```

2. attaches `Signature` and `Snippet` attributes to the class Vector specifying the parametric class signature and the class snippets ($snip$1) including those in the snippet environment Vector.$IsolatedSnippets;

3. attaches a `Snippet` attribute to the **protected** base method zip specifying the type dependent operations corresponding to each snippet in the snippet environment zipSnippets;

4. translates the class Main to the form:

```
package Q;

public class Main {
   // inititialize server classes
   static {
      Vector$_String_$.$init(
         new Vector.$IsolatedSnippets() {
            Object[ ] $1(int initCapacity) {
               return new String[initCapacity];
            }
         }
      );
   }
      static {
         Vector$_Integer_$.$init(
            new Vector.$IsolatedSnippets() {
               Object[ ] $1(int initCapacity) {
                  return new Integer[initCapacity];
               }
            }
         );
      }
      static {
         Vector$_Pair$_String$Integer_$_$.$init(
            new Vector.$IsolatedSnippets() {
               Object[ ] $1(int initCapacity) { return
                  new Pair$_String$Integer_$[initCapacity];
               }
            }
         );
      }
   public static void main(String args[ ]) {
      static Vector$_String_$ names =
            Vector$_String_$(100);
      static Vector$_Integer_$ numbers =
            Vector$_Integer_$(100);
      static Vector$_Pair$_String$Integer_$_$ phonebook =
            names.zip$_Integer_$(numbers);
      ...
   }
}
```

and generates the wrapper classes

```
$$Vector$_String_$
$$Vector$_Integer_$
$$Vector$_Pair$_String$Integer_$_$
$$Pair$_String$Integer_$
```

and corresponding interfaces in package P; and

5. augments the class file for $$Vector$_String_$ with a public wrapper method for the method instantiation

```
names.⟨Integer⟩zip(numbers)
```

in class Main:

```
public Vector zip$_Integer_$(Vector other) {
   return zip(new $zip$Snippets() {
      Vector $1(int s) {
         return new $$Pair$_String$Integer_$[s];
      }
      Pair $2(Object fst, Object snd) {
         return new Pair$_String$Integer_$(fst,snd);
      }
   },
   other);
}
```

because the variable names has type $$Vector$_String_$.

Any wrapper class subsequently generated by the compiler that is a subtype of $Vector$_String_$ must include a definition for the wrapper method zip$_Integer_$. This obligation is recorded in a *WrapperMethods* attribute attached to the interface $Vector$_String_$.

Note that if the class Main instantiated zip with a default-access type (invisible outside package Q), the translation would have to invoke the machinery required to support escaping wrapper methods.

7.1 Type Inference

The current GJ design supports type inference for parameterized methods as a convenience to the programmer. Exactly the same facility could be added to NextGen. The presence of covariant subtyping for NextGen parameterized classes does not introduce any new complications because Java array types are already covariant. We are not convinced that the gain in notational convenience is worth the added complexity in the language definition.

8 Classes and Interfaces as Parameters

In NextGen, there are two contexts where a type variable must be bound to a class rather than an interface. The most important is a **new** expression of the form **new** T(...) where T is a type variable. To help programmers recognize which parameterized classes contain **new** operations on isolated type variables, NextGen requires the *binding occurrence* (its declaration in the class header) of such a type variable to be prefixed by the keyword **class**. A **class** type variable must be bound to a *concrete* class; it cannot be an interface or an **abstract** class.

The other context separating classes from interfaces is the instantiation of a type variable bounded by a class. Since an interface cannot extend a class, the binding of such a type variable must be a class. However, in contrast to the context above, the binding may be an **abstract** class. Hence, the binding occurrence of a type variable bound by a class type may include the prefix **abstract class**.

In general, NextGen permits the prefixes **class**, **abstract class**, or **interface** to appear before the binding occurrence (declaration) of a type variable. If none of these prefixes appears, the default is **interface**, unless the binding occurrence contains an **extends** clause, in which case the default is **abstract class**.

For example we may write

```
class Vector⟨abstract class A⟩{ ... }
class Stack⟨class A⟩extends Vector⟨A⟩{ ... }
class List⟨interface A⟩{ ... }
```

When one parameterized class extends another, the prefixes (implicit or explicit) of the new binding classes must be compatible with their use in the extended class (or interface).

9 Covariance and Contravariance

While the distinctions between covariant, contravariant, and non-variant parameterization may appear to be the province of type theoreticians, they *all* arise in practice. In fact, each is invaluable in describing important parameterization patterns that frequently occur in real applications. Since all three forms of type parameterization are important, we would like to support all three in NextGen. Unfortunately, we do not know how to implement contravariant parameterization efficiently on top of the existing JVM. For this reason, contravariant parameterization is not included in the current design. In the following discussion, however, we will assume that contravariant parameterization is permitted, in anticipation of future extensions to the JVM and NextGen.

Covariant and contravariant parameterization are respectively indicated by inserting the prefix + or − in front of the type parameter. The sole effect of these prefixes is to determine the subtyping behavior of parameters, which imposes typing restrictions on program text involving parameterized types. As an example, consider a covariant version of Vector:

```
public class Vector⟨+A⟩{ ... }
```

If the class declaration type-checks, NextGen translates it in almost exactly the same way as a class with non-variant parameterization. The only change is that whenever a wrapper interface is generated, it implements an additional interface corresponding to an immediate covariant supertype:

```
public interface $Vector$_Integer_$
    implements $Vector$_Number_$ {}
```

Since Number is the superclass of Integer, Vector<Number> is an immediate supertype of Vector⟨Integer⟩. This relationship between the interfaces representing parametric types completely captures the covariance property. As another example, consider the declaration

```
public class Pair⟨+A,+B⟩{ ... }
```

For the instantiation Pair⟨String,Integer⟩, NextGen would generate:

```
public interface $Pair$_String$Integer_$ implements
    $Pair$_Object$Integer_$, $Pair$_String$Number_$,
```

In general, a wrapper interface for a covariantly parameterized class or interface implements all the wrapper interfaces for immediate covariant supertypes.

To ensure type safety, NextGen prohibits a *covariant* type from being used as the type of a method parameter and a *contravariant* type from being used as the return type of a method. Note that these restrictions apply to inherited methods as well as manifest ones. In addition, the usage of class instance variables with types involving covariant or contravariant type variables must be restricted because variable access and assignment are implicit methods with parametric signatures. To ensure the type safety of operations on instance variables, we impose two restrictions. First, the class instance variables must be declared **private** or **protected**. Second, these instance variables can only be extracted from the pseudo-variable **this**. These rules ensure that the type constraint associated with an instance variable of parametric type T is exactly membership in T, not membership in some subtype or supertype of T.

The usage of static (class) variables and methods with covariant (or contravariant) parametric types is unrestricted (except by Java's usual type checking rules) because the associated type constraints are manifest at compile time.

10 Future Directions

The most serious limitation of the NextGen genericity facility is the lack of support for parameterization by primitive types. Such "primitive" parameterization is incompatible with the homogeneous implementation techniques (based on type erasure) adopted from Pizza and GJ. NextGen could be extended to support primitive parameterization by incorporating a "brute force" heterogeneous translation for primitive type instantiations. Such a facility may be important in practice despite the potential for explosive increases in program code size. To prevent the growth of program binaries, Agesen et.al. [AFM97] have proposed deferring heterogeneous class instantiation until load time. Unfortunately, this technique requires a revised class loader, breaking backward compatibility with existing Java Virtual Machine.

To support a coherent parameterization facility that uniformly supports both reference and primitive types, we need to explore an extension of Java that incorporates the primitive types in the object type hierarchy. Such an exercise is clearly beyond the scope of this paper.

References

[AFM97] Ole Agesen, Stephen Freund, and John Mitchell. Adding parameterized types to Java. In *ACM Symposium of Object-Oriented Programming: Systems, Languages, and Applications*, October 1997.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java™ Programming Language. To appear in *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, October 1998.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. Functional Programming Languages and Computer Architecture*, September 1989, 273-280.

[Mey92] Bertrand Meyer. Eiffel: The Language. Prentice Hall, 1992.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *ACM Symposium on Principles of Programming Languages*, January 1997, 132–145.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages*, January 1997, 146–159.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming*, LNCS 1241, Springer-Verlag, 1997, 444-471.

[Tor98] Mads Torgersen. Virtual types are statically safe. *Fifth Workshop on Foundations of Object-Oriented Languages*, January 1998.