Making the future safe for the past: Adding Genericity to the Java™ Programming Language

Gilad Bracha, Sun Microsystems, gilad.bracha@sun.com
Martin Odersky, University of South Australia, odersky@cis.unisa.edu.au
David Stoutamire, Sun Microsystems, david.stoutamire@sun.com
Philip Wadler, Bell Labs, Lucent Technologies, wadler@research.bell–labs.com

Abstract

We present GJ, a design that extends the Java programming language with generic types and methods. These are both explained and implemented by translation into the unextended language. The translation closely mimics the way generics are emulated by programmers: it erases all type parameters, maps type variables to their bounds, and inserts casts where needed. Some subtleties of the translation are caused by the handling of overriding.

GJ increases expressiveness and safety: code utilizing generic libraries is no longer buried under a plethora of casts, and the corresponding casts inserted by the translation are guaranteed to not fail.

GJ is designed to be fully backwards compatible with the current Java language, which simplifies the transition from non-generic to generic programming. In particular, one can retrofit existing library classes with generic interfaces without changing their code.

An implementation of GJ has been written in GJ, and is freely available on the web.

1 Introduction

Generic types are so important that even a language that lacks them may be designed to simulate them. Some object-oriented languages are designed to support subtypes directly, and to support generics by the idiom of replacing variable types by the top of the type hierarchy. For instance, a collection with elements of any type is represented by a collection with elements of type Object.

To appear in the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98), Vancouver, BC, Canada, October, 1998.

This approach is exemplified by the Java programming language[GLS96]. Generics are represented by this idiom throughout the standard Java libraries, including vectors, hash tables, and enumerations. As the Java Development Kit (JDK) has evolved, generics have played an increasing role. JDK 1.1 introduced an observer pattern that depends on generics, as do the collection classes introduced in JDK 1.2. Oberon also relies on the generic idiom, and dynamically typed languages such as Smalltalk [GR83] use this idiom implicitly.

Nonetheless, generics may merit direct support. Designing a language with direct support for subtyping and generics is straightforward. Examples include Modula 3, Ada 95, Eiffel, and Sather. Adding generics to an existing language is almost routine. We proposed adding generics to the Java programming language in Pizza [OW97], and we know of four other proposals [AFM97, MBL97, TT98, CS98]. Clemens Szyperski proposed adding generics to Oberon [RS97]. Strongtalk [BG93] layers a type system with generic types on top of Smalltalk.

The generic legacy problem However, few proposals tackle the *generic legacy* problem: when direct support for generics is added to a language that supports them via the generic idiom, what happens to legacy code that exploits this idiom?

Pizza is backward compatible with the Java programming language, in that every legal program of the latter is also legal in the former. However, this compatibility is of little help when it comes to generics. For example, JDK 1.2 contains an extensive library of collection classes based on the generic idiom. It is straightforward to rewrite this library to use generics directly, replacing the legacy type Collection by the parametric type Collection<A>. However, in Pizza these two types are incompatible, so one must rewrite all legacy code, or write adaptor code to convert between legacy and parametric types. Code bloat may result from references to both the legacy and parametric versions of the library. Note the problem is not merely with the size of legacy libraries (which may be small), but with managing the upgrade from the legacy types to parametric types (which can be a major headache if references to legacy types are dispersed over a large body of code). If legacy libraries or code are available only in binary rather than source, then these problems are compounded.

GJ Here we propose GJ, a superset of the Java programming language that provides direct support for generics. GJ compiles into Java virtual machine (JVM) byte codes, and can be executed on any Java compliant browser. In these respects GJ is like Pizza, but GJ differs in that it also tackles the generic legacy problem.

GJ contains a novel language feature, *raw types*, to capture the correspondence between generic and legacy types, and a *retrofitting* mechanism to allow generic types to be imposed on legacy code. A parametric type Collection<A> may be passed wherever the corresponding raw type Collection is expected. The raw type and parametric type have the same representation, so no adaptor code is required. Further, retrofitting allows one to attribute the existing collection class library with parametric types, so one only requires one version of the library; an added plus is that new code will run in any JDK 1.2 compliant browser against the built-in collection class library. Raw types and retrofitting apply even if libraries or code are available only as binary class files, and no source is available. Combined, these techniques greatly ease the task of upgrading from legacy code to generics.

The semantics of GJ is given by a translation back into the Java programming language. The translation erases type parameters, replaces type variables by their bounding type (typically Object), adds casts, and inserts bridge methods so that overriding works properly. The resulting program is pretty much what you would write in the unextended language using the generic idiom. In pathological cases, the translation requires bridge methods that can only be encoded directly in JVM byte codes. Thus GJ extends the expressive power of the Java programming language, while remaining compatible with the JVM.

GJ comes with a cast-iron guarantee: no cast inserted by the compiler will ever fail. (Caveat: this guarantee is void if the compiler generates an 'unchecked' warning, which may occur if legacy and parametric code is mixed without benefit of retrofitting.) Furthermore, since GJ compiles into the JVM, all safety and security properties of the Java platform are preserved. (Reassurance: this second guarantee holds even in the presence of unchecked warnings.)

Security One may contrast two styles of implementing generics, *homogeneous* and *heterogeneous*. The homogeneous style, exemplified by the generic idiom, replaces occurrences of the type parameter by the type Object. The heterogeneous style, exemplified by C++ and Ada, makes one copy of the class for each instantiation of the the type parameter. The GJ and Pizza compilers implement the homogeneous translation, while Agesen, Freund, and Mitchell [AFM97] propose having the class loader implement the heterogeneous translation. Other proposals utilize a mixture of homogeneous and heterogeneous techniques [CS98].

As observed by Agesen, Freund, and Mitchell, the heterogeneous translation provides tighter security guarantees than the homogeneous. For example, under the homogeneous translation a method expecting a collection of secure channels may be passed a collection of any kind of object, perhaps leading to a security breach. To minimize this problem, GJ always inserts bridge methods when subclassing a generic class, so the user may ensure security simply by declaring suitable specialized subclasses.

The homogeneous translation also enjoys some advantages over the heterogeneous. Surprisingly, with the security model of the Java virtual machine, the heterogeneous translation makes it impossible to form some sensible type instantiations. (This problem is entirely obvious, but only in retrospect.) GJ and other languages based on the homogeneous translation do not suffer from this difficulty.

Type inference While type systems for subtyping and for generics are well understood, how to combine the two remains a topic for active research. In particular, it can be difficult to infer instantiations for the type arguments to generic methods.

GJ uses a novel algorithm for this purpose, which combines two desirable (and at first blush contradictory) properties: it is *local*, in that the type of an expression depends only on the types of its subexpressions, and not on the context in which it occurs; and it *works for empty*, in that inference produces best types even for values like the empty list that have many possible types. Further, the inference algorithm *supports subsumption*, in that if an expression has a type, then it may be regarded as having any supertype of that type.

In contrast, the algorithm used in Pizza is non-local and does not support subsumption (although it does work for empty), while the algorithm used in Strongtalk [BG93] does not work for empty (although it is local and supports subsumption), and algorithms for constraint-based type inference [AW93, EST95] are non-local (although they work for empty and support subsumption).

Pizza uses a variant of the Hindley-Milner algorithm [Mil78], which we regard as non-local since the type of a term may depend on its context through unification.

Raw types and retrofitting Raw types serve two purposes in GJ: they support interfacing with legacy code, and they support writing code in those few situations (like the definition of an equality method) where it is necessary to downcast from an unparameterized type (like Object) to a parameterized type (like LinkedList<A>), and one cannot determine the value of the type parameter. The type rules for raw types are carefully crafted so that the compiler can guarantee the absence of type errors in methods like equality. However, when interfacing to legacy code, compile-time checking is not always possible, and in this case, an 'unchecked' warning may be issued. The proliferation of 'unchecked' warnings can be avoided by using retrofitting to add information about type parameters to legacy code.

Related work GJ is based closely on the handling of parametric types in Pizza [OW97]. The Pizza compiler (itself written in Pizza) has been freely available on the web since 1996. GJ differs from Pizza in providing greater support for backward compatibility, notably in allowing new code to work with old libraries. GJ also uses a simpler type system. In Pizza the type of an expression may depend on the type expected by its context, whereas in GJ the type of an expression is determined solely by the type of its constituents.

GJ maintains no run-time information about type parameters. The same design decision is made in Pizza and in a proposal to add parametric types to Oberon [RS97]. There are a number of other proposals for adding parameterized types to the Java programming language, all based on carrying type information at run-time [AFM97, MBL97, CS98]. Run-time types may be less efficient to implement than erasure [OR98], and may be harder to interface with legacy code; on the other hand, it is arguably more expressive to maintain run-time type information, and more consistent with the rest of the design of the Java programming language, which maintains run-time type information about classes and arrays. For this reason, GJ has been designed to be compatible with an extension that maintains type information at run-time.

In particular, Cartwright and Steele have developed the NextGen design in tandem with GJ [CS98]. Just as the Java programming language is a subset of GJ, so GJ is a subset of NextGen. A more detailed comparison with NextGen appears in the conclusion.

Virtual types have been suggested as an alternative to parametric types [Tho97, Tor98]. A comparison of the relative strengths of parametric and virtual types appears elsewhere [BOW98]. It may be possible to merge virtual and parametric types [BOW98, TT98], but it is not clear whether the benefits of the merger outweigh the increase in complexity.

Status An implementation of GJ is freely available on the web [GJ98a]. The GJ compiler is derived from the Pizza compiler and, like it, can also be used as a stand-alone compiler for the Java programming language. The compiler is about 20,000 lines of GJ.

This paper concentrates on the design issues underlying GJ. Companion papers provide a tutorial introduction [GJ98b] and a precise specification [GJ98c].

Outline The remainder of this paper is structured as follows. Section 2 introduces the basic features of GJ, using a running example based on collections and linked lists. Section 3 details the translation from GJ into the Java programming language and JVM byte code. Section 4 explains why an invariant subtyping rule is used for parameterized types. Section 5 describes the type inference algorithm. Section 6 discusses how generics relate to the Java platform's security model. Section 7 details restrictions imposed on the source language by the lack of run-time type information. Section 8 introduces raw types. Section 9 describes retrofitting. Section 10 shows how generics are exploited in the implementation the GJ compiler itself. Section 11 concludes.

2 Generics in GJ

Figure 1 shows a simplified part of the Java collection class library expressed in GJ. There are interfaces for collections and iterators, and a linked list class. The collection interface provides a method to add an element to a collection (*add*), and a method to return an iterator for the collection (*iterator*). In turn, the iterator interface provides a method to determine if the iteration is done (*hasNext*), and (if it is not) a method to return the next element and advance the iterator (*next*). The linked list class implements the collections interface. It contains a nested class for list nodes (*Node*), and an anonymous class for the list iterator.

The interfaces and class take a type parameter A, written in angle brackets, representing the element type. The nested class Node has A as an implicit parameter inherited from the scope, the full name of the class being LinkedList<A>.Node. The scope of a type parameter is the entire class, excluding static members and static initializers. This is required since different instances of a class may have different type parameters, but access the same static members. Parameters are irrelevant when using a class name to access a static member, and must

```
interface Collection<A> {
    public void add (A x);
    public Iterator<A> iterator ();
}

interface Iterator<A> {
    public A next ();
    public boolean hasNext ();
}

class NoSuchElementException extends RuntimeException {}

class LinkedList<A> implements Collection<A> {

    protected class Node {
        A elt;
        Node next = null;
        Node (A elt) { this.elt = elt; }
    }

    protected Node head = null, tail = null;

    public LinkedList () {}

    public void add (A elt) {
        if (head == null) {
            head = new Node(elt); tail = head;
        } else {
            tail.next = new Node(elt); tail = tail.next;
        }
    }

    public Iterator<A> iterator () {
        return new Iterator<A> () {
            protected Node ptr = head;
            public boolean hasNext () { return ptr != null; }
            public A next () {
                if (ptr != null) {
                    A elt = ptr.elt; ptr = ptr.next; return elt;
                } else {
                    throw new NoSuchElementException ();
                }
            }
        };
    }
}

class Test {
    public static void main (String[ ] args) {
        LinkedList<String> ys = new LinkedList<String>();
        ys.add("zero"); ys.add("one");
        String y = ys.iterator().next();
    }
}
```

Figure 1: Collection classes in GJ

```
interface Comparable<A> {
    public int compareTo (A that);
}
class Byte implements Comparable<Byte> {
    private byte value;
    public Byte (byte value) { this.value = value; }
    public byte byteValue () { return value; }
    public int compareTo (Byte that) {
        return this.value – that.value;
    }
}
class Collections {
    public static <A implements Comparable<A>>
        A max (Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0) w = x;
        }
        return w;
    }
}
```

Figure 2: Generic methods and bounds

be omitted. In general, nested classes may have type parameters, and (if not static) also inherit type parameters from any surrounding class.

Angle brackets were chosen for type parameters since they are familiar to C++ users, and each of the other form of brackets may lead to confusion. If round brackets are used, it is difficult to distinguish type and value parameters. If square brackets are used, it is difficult to distinguish type parameters and array dimensions. If curly brackets are used, it is difficult to distinguish type parameters from class bodies.

Phrases like `LinkedList<LinkedList<String>>` pose a problem to the parser, since `>>` is treated as a single lexeme. (Similarly for `>>>`.) In C++, users are required to add extra spaces to avoid this problem. In GJ, the grammar has been modified so that no spaces are required.

The example in Figure 2 shows another part of the collection class library of JDK 1.2 expressed in GJ. There is an interface `Comparable<A>` for objects that can be compared to other objects of type `A`. Class `Byte` implements this interface with itself as the type parameter, hence, bytes can be compared with themselves.

The last class in Figure 2 defines a static method `max` that returns the maximum element of a non-empty collection. This method demonstrates two features: it is a generic method, and also has a bounded type parameter. The method is generic because it applies to a variety of types. To declare a generic method, the quantified type variables are written in angle brackets preceding the method signature and body. The type is automatically instantiated at point of use. For instance, if `ys` has type `Collection<Byte>` we may write

```
Byte x = Collections.max(ys);
```

and the parameter `A` of `max` is inferred to be `Byte`. The type parameter `A` is bounded because it varies not over all types, but only over types that are comparable to themselves. For instance, the parameter may be instantiated to `Byte` because `Byte` implements `Comparable<Byte>`.

Any type parameter (to an interface, class, or generic method) may be bounded. A bound is indicated by following the parameter with the keyword **implements** and an interface or **extends** and a class. The bounding interface or class may itself be parameterized, and may include type variables appearing elsewhere in the parameter section. Recursion or mutual recursion between parameters is allowed — that is, GJ supports F-bounded polymorphism [CCHOM89]. Omitting a bound is equivalent to using the bound `Object`.

3 Translating GJ

To translate from GJ to the Java programming language, one replaces each type by its *erasure*. The erasure of a parametric type is obtained by deleting the parameter (so `LinkedList<A>` erases to `LinkedList`), the erasure of a non-parametric type is the type itself (so `String` erases to `String`) and the erasure of a type parameter is the erasure of its bound (so `A` in `Collections.max` erases to `Comparable`).

Translating the GJ code for collection classes in Figures 1 and 2 yields the code in Figures 3 and 4. The translated code is identical to the original collection class code written using the generic idiom. This property is essential – it means that a GJ program compiled against the parameterized collection library will run on a browser that contains the original collection library.

The translation of a method erases all argument types and the return type, and inserts type casts where required. A cast is inserted in a method call when the result type of the method is a type parameter, or in a field access when the type of the field is a type parameter. For example, compare `Test.main` in Figure 1 with its translation in Figure 3, where a cast is inserted into the call of `next`.

The translation inserts *bridge methods* to ensure overriding works correctly. A bridge is required when-

```
interface Collection {
    public void add (Object x);
    public Iterator iterator ();
}
interface Iterator {
    public Object next ();
    public boolean hasNext ();
}
class NoSuchElementException extends RuntimeException {}
class LinkedList implements Collection {
    protected class Node {
        Object elt;
        Node next = null;
        Node (Object elt) { this.elt = elt; }
    }
    protected Node head = null, tail = null;
    public LinkedList () {}
    public void add (Object elt) {
        if (head == null) {
            head = new Node(elt); tail = head;
        } else {
            tail.next = new Node(elt); tail = tail.next;
        }
    }
    public Iterator iterator () {
        return new Iterator () {
            protected Node ptr = head;
            public boolean hasNext () { return ptr != null; }
            public Object next () {
                if (ptr != null) {
                    Object elt = ptr.elt; ptr = ptr.next; return elt;
                } else {
                    throw new NoSuchElementException ();
                }
            }
        };
    }
}
class Test {
    public static void main (String[ ] args) {
        LinkedList ys = new LinkedList();
        ys.add("zero"); ys.add("one");
        String y = (String)ys.iterator().next();
    }
}
```

Figure 3: Translation of collection classes

```
interface Comparable {
    public int compareTo (Object that);
}
class Byte implements Comparable {
    private byte value;
    public Byte (byte value) { this.value = value; }
    public byte byteValue () { return value; }
    public int compareTo (Byte that) {
        return this.value – that.value;
    }
    public int compareTo (Object that) {
        return this.compareTo((Byte)that);
    }
}
class Collections {
    public static Comparable max (Collection xs) {
        Iterator xi = xs.iterator();
        Comparable w = (Comparable)xi.next();
        while (xi.hasNext()) {
            Comparable x = (Comparable)xi.next();
            if (w.compareTo(x) < 0) w = x;
        }
        return w;
    }
}
```

Figure 4: Translation of generic methods and bounds

ever a subclass (non-trivially) instantiates a type variable in a superclass. For example, erasure of compareTo in Comparable yields a method that takes an Object, while erasure of compareTo in Byte yields a method that takes a Byte. Since overriding occurs only when method signatures match exactly, a bridge method for compareTo is introduced into the translation of Byte that takes an Object and casts it to a Byte. Overloading allows the bridge and the original method to share the same name.

Again, the translation from GJ yields code identical to the original collection class library in JDK 1.2, including the bridge methods.

3.1 A bridge too far

A problematic case of bridging may arise if a type parameter appears in the result but not the arguments of an overridden method.

Here is a class that implements the Iterator interface in Figure 1.

```
class Interval implements Iterator<Integer> {
    private int i, n;
    public Interval (int l, int u) { i = l; n = u; }
    public boolean hasNext () { return (i <= n); }
    public Integer next () { return new Integer(i++); }
}
```

Here the next method of the class returns an Integer, to match the instantiation of the type parameter.

The translation yields the following. As one would expect, a bridge must be added to the Interval class.

```
interface Iterator {
    public boolean hasNext ();
    public Object next ();
}
class Interval implements Iterator {
    private int i, n;
    public Interval (int l, int u) { i = l; n = u; }
    public boolean hasNext () { return (i <= n); }
    public Integer next/*1*/ () {
        return new Integer(i++);
    }
    // bridge
    public Object next/*2*/ () {
        return next/*1*/();
    }
}
```

Unfortunately, this is not legal Java source code, as the two versions of next cannot be distinguished because they have identical arguments. The code above distinguishes our intention by suffixing the declarations and calls with /*1*/ and /*2*/ as appropriate.

Fortunately, the two versions of next *can* be distinguished in the JVM, which identifies methods using a signature that includes the result type. This situation represents the one place where GJ must be defined by translation directly into JVM byte code.

GJ also permits covariant overriding: an overriding method may have a result type that is a subtype of the method it overrides (whereas it must match exactly in the unextended Java programming language). Here is an example.

```
class C implements Cloneable {
    public C copy () { return (C)this.clone(); }
}
class D extends C implements Cloneable {
    public D copy () { return (D)this.clone(); }
}
```

Translation introduces a bridge method into the second class.

```
class D extends C implements Cloneable {
    public D copy/*1*/ () { return (D)this.clone(); }
    // bridge
    public C copy/*2*/ () { return this.copy/*1*/(); }
}
```

This is implemented using the same technique as above.

4 Subtyping

For purposes of type comparison, subtyping is invariant for parameterized types. For instance, even though the class String is a subtype of Object, the parameterized type LinkedList<String> is not a subtype of LinkedList<Object>. In comparison, arrays use covariant subtyping, so the array type String[ ] is a subtype of Object[ ].

Invariant subtyping ensures that the type constraints enforced by GJ are not violated. Consider the following code.

```
class Loophole {
    public static String loophole (Byte y) {
        LinkedList<String> xs =
            new LinkedList<String>();
        LinkedList<Object> ys =
            xs;  // compile–time error
        ys.add(y);
        return xs.iterator().next();
    }
}
```

This code is illegal, because otherwise it would violate the type constraints by returning a byte when a string is expected. Both the method call (which adds a byte, which is itself an object, to a list of objects) and the return (which extracts a string from a list of strings) are unobjectionable, so it must be the assignment (which aliases a list of string to a list of objects) that is at fault.

It is instructive to compare the above to analogous code for arrays.

```
class Loophole {
    public static String loophole (Byte y) {
        String[ ] xs = new String[1];
        Object[ ] ys = xs;
        ys[0] = y;  // run–time error
        return xs[0];
    }
}
```

Now the code is legal, but raises an array store exception. Observe that the type safety of covariant subtyping depends upon the fact that an array carries its type at run-time, making the store check possible. This approach is not viable for parameterized types, since type parameters are not available at run-time.

```
class ListFactory {
    public <A> LinkedList<A> empty () {
        return new LinkedList<A>();
    }
    public <A> LinkedList<A> singleton (A x) {
        LinkedList<A> xs = new LinkedList<A>();
        xs.add(x);
        return xs;
    }
    public <A> LinkedList<A> doublet (A x, A y) {
        LinkedList<A> xs = new LinkedList<A>();
        xs.add(x); xs.add(y);
        return xs;
    }
}
class Test {
    static ListFactory f = new ListFactory();
    public static void main (String[ ] args) {
        LinkedList<Number> zs =
            f.doublet(new Integer(1), new Float(1.0));
        LinkedList<String> ys = f.singleton(null);
        LinkedList<Byte> xs = f.empty();
        LinkedList<Object> err =
            f.doublet("abc", new Integer(1));
                // compile–time error
    }
}
```

Figure 5: Example of inference

It should be noted that explicitly declared subtyping is not a problem. For instance, it is fine to pass a LinkedList<String> when a Collection<String> is expected.

5 Type Parameter Inference

GJ includes a novel type parameter inference algorithm that permits one to elide type parameters to polymorphic method calls. Such type parameters can safely be omitted since they are erased by the translation anyway, and therefore cannot carry any operational meaning.

Type parameters are inferred for a parametric method call by choosing the smallest type parameter that yields a valid call. As an example, consider the the code in Figure 5, which defines factory methods for lists with zero, one, and two elements.

In the example above, the call to doublet with an integer and a float as arguments infers that the type parameter A is Number. If there is no unique smallest type, inference fails, as in the call to doublet with a string and an integer (which have both Comparable and

```
class Cell<A> {
    public A value;
    public Cell (A v) { value = v; }
    public static <A> Cell<A> make (A x) {
        return new Cell(x);
    }
}
class Pair<B,C> {
    public B fst;
    public C snd;
    public Pair (B x, C y) { fst = x; snd = y; }
    public static <D> Pair<D,D> duplicate (D x) {
        return new Pair<D,D>(x,x);
    }
}
class Loophole {
    public static String loophole (Byte y) {
        Pair<Cell<String>,Cell<Byte>> p =
            Pair.duplicate(Cell.make(null));
            // compile–time error
        p.snd.value = y;  return p.fst.value;
    }
    public static String permitted (String x) {
        Pair<Cell<String>,Cell<String>> p =
            Pair.duplicate(Cell.make((String)null));
        p.fst.value = x;  return p.snd.value;
    }
}
```

Figure 6: Illegal situation for inference.

```
    interface I {}
    interface J {}
    interface K extends I {}
    interface L extends I, J {}

class X {
    static <A> A choose(A x, A y) {
        return (x.hash() < y.hash())?x:y;
    }
    static void test (K k, L l) {
        I i = choose(k, l);  // ok
    }
}
```

Figure 7: Near-ambiguous situation for inference

Serializable as common supertypes).

This rule needs to be generalized to the cases where there are no argument types involving an inferred variable or where some argument is null. To support inference in these cases, the type inferencer may bind a type variable to the special 'bottom' type $*$, the type of null. The type $*$ is a subtype of every reference type. This type is used only by the type inference algorithm, and cannot appear in type declarations in GJ programs. Further, any type containing $*$ is regarded as a subtype of any type that results from replacing $*$ with any other reference type. (This is the one exception to the rule of invariant subtyping.) Thus, LinkedList<$*$> is a subtype of LinkedList<String>, and Pair<Byte,$*$> is a subtype of Pair<Byte,Byte>.

For instance, in the second and third calls of method main above, the type parameter is inferred to be $*$. The assignments are valid since LinkedList<$*$> is a subtype of both LinkedList<Byte> and LinkedList<String>.

An additional *linearity* restriction is required: a type parameter cannot be instantiated to $*$ if it appears more than once in the result type. To see why this is necessary, consider the code in Figure 6. The call to duplicate in loophole is illegal, because the smallest choice for D is $*$, but D appears twice in the result type of duplicate. On the other hand, the call to duplicate in permitted is ok, because the cast ensures the smallest choice for D is String. But without the cast, the smallest choice is $*$ and the call would be illegal. Without the restriction, loophole would circumvent the type system, making it possible to treat a string as a byte.

General covariance may lead to an unsound type system, so we have to argue carefully that our type system with its restricted form of covariance remains sound. The argument goes as follows: since one cannot declare variables of type T<...$*$...>, all one can do with a value of that type is assign or pass it once to a variable or parameter of some other type. There are now three possibilities, depending on the variable's type:

- The variable's type is an unparameterized supertype of T. In this case the assignment is clearly sound.

- The variable's type is T<...U...> with some reference type U in the position of $*$. Now, the only value that populates type $*$ is null, which is also a value of every reference type U. Furthermore, any method in type T<...$*$...> with an argument V<...$*$...> that contains the bottom type would have to be parametric in this type, so that it could equally well be applied to V<...U...>. Hence, any value of type T<...$*$...> will also be a value of type T<...U...>, and the assignment is sound.

- The variable's type is a type variable, A. Then code that accesses the variable works for any type A may be instantiated to, so the code itself cannot give rise to type errors. Furthermore, if the variable appears in a method, by the linearity restriction, the method's formal result type will contain at most one occurrence of A, so the actual type of the method application is again of the form T'<...$*$...>.

Our type parameter inference scheme is similar to Pierce and Turner's local type inference [PT98]. Pierce and Turner only consider covariant type constructors, which is a sensible assumption for the predominantly functional languages they are dealing with. For GJ, with its exclusive use of invariant type constructors, our special treatment of $*$ is essential to make type inference work. Experience has so far shown that it works very well indeed. For instance, in the whole GJ compiler (consisting of about 20,000 lines of heavily generic code), there was not a single instance where type inference had to be helped by an explicit parameterization or type cast.

It is instructive to compare GJ's local type inference with the constraint-based inference of the Hindley-Milner system [Mil78] or its extensions to subtyping [AW93, EST95]. In essence, a type T<...$*$...> in our system would correspond to a type T<...A...> in the Hindley-Milner system, where A is a fresh type variable that is used nowhere else. If a type had more than one occurrence of $*$, each occurrence would be replaced by a different type variable. Then our use of subtyping for types containing $*$ corresponds to instantiations of type variables in the Hindley-Milner system. The linearity condition makes sure that $*$ types are not duplicated when types for method calls are inferred, so that each $*$ type can be mapped back to a fresh type variable in the method's result type. Finally, the restriction that $*$ types cannot be declared by the user roughly corresponds to the variable polymorphism restriction for Hindley-Milner [Wri95], which ensures that values containing mutable references cannot be polymorphic.

Note that any inference algorithm is subject to problems with ambiguity. Consider the code in Figure 7. Here the type inferencer can determine that the formal parameter A corresponds to the actual parameter I in the call of choose in the marked line. However, say that the definition of K is later changed, so that K also extends J.

```
interface K extends I, J {}
```

Now the call to choose becomes ambiguous. Thus even though the programmer has taken care to preserve the supertypes and structure of interface K, code using it breaks due to the change. This is an undesirable property from a software engineering perspective. However, the Java programming language already suffers from a similar problem with regard to overloading, so adding type inference does not introduce any new holes. We believe the convenience of inference outweighs this attendant infelicity.

6 Security Implications

Since the homogeneous translation erases type information, it opens a potential security hole at run-time. The hole can be filled, but to do that one needs to be aware of it. Consider the following example, which is due to Agesen, Freund, and Mitchell [AFM97]:

```
class SecureChannel extends Channel {
   public String read ();
}
class C {
   public LinkedList<SecureChannel> cs;
   ...
}
```

Since LinkedList<SecureChannel> gets erased to just LinkedList, it is possible for an attacker to add a non-secure channel to the list, which might be used as a way to leak information from a secure system. If the attacker was itself written in GJ, this would be prevented by the generic type system. But the attacker could be written not in GJ but in the Java programming language or the JVM byte code language, in which case neither the compiler nor the run-time system would detect a type system violation.

To address this problem, a programmer needs to prevent the information about the type parameter from being lost by erasure. If the class in questions does not export any parameterized fields this can be achieved by declaring a specialized type SecureChannelList, which extends type LinkedList<SecureChannel>. The specialization inherits all fields and methods from its supertype, and its constructor simply forwards to the analogous constructor in the supertype:

```
class SecureChannelList
     extends LinkedList<SecureChannel> {
   SecureChannelList () { super(); }
}
class C {
   SecureChannelList cs;
   ...
}
```

Unlike LinkedList<SecureChannel>, SecureChannelList gets translated to itself, so no type information is lost. Furthermore, GJ's translation scheme for bridge methods ensures that argument types are properly checked at run-time. Here is the translation of class SecureChannelList:

```
public static Object[ ] newInstance (Object[ ] a, int n) {
    return (Object[ ])Array.newInstance(a.getClass().getComponentType(), n);
}
```

Figure 8: Creating a new instance of an array

```
class SecureChannelList extends LinkedList {
    SecureChannelList () { super(); }
    public void add (Object x) {
        super.add((SecureChannel)x);
    }
}
```

Note the inserted bridge method for LinkedList.add which checks at run-time that the passed channel is secure. The same scheme cannot be applied to public fields of parameterized types, since access to those fields is not encapsulated by bridge methods.

Type specialization is a general method for maintaining type parameter information which would otherwise be lost by erasure. Since the heterogeneous translation effectively applies type specialization everywhere, it looks like a better fit from a security perspective. This is also argued by Agesen *et al.* [AFM97].

Perhaps surprisingly, the heterogeneous translation nevertheless fits poorly with the security model of the Java virtual machine. The problem, first reported in [OR98], lies in the package based visibility model for types, which can interfere with automatic type specialization.

The JVM security model supports only two kinds of visibility for top-level classes: package-wide and public visibility. It is not possible to refer to a class outside a package unless the class is declared public. The JVM specification [LY96] requires the virtual machine to throw an `IllegalAccessError` if a class refers to any class that is in another package and is not public.

Sometimes these rules make it impossible to find a package where a heterogeneous type instantiation can be placed. Consider an instantiation p.C<q.D> of a parameterized class C defined in package p, applied to a parameter class D defined in a different package q. There are two possibilities: either class D must be public (in which case we can place the instantiation in package p), or else the body of class C must refer only to public classes (in which case we can place the instantiation in package q). If neither of these cases apply (that is, D is private in its package and C refers to private classes in its package), then there is no package in which one can place the instantiation p.C<q.D>, hence the heterogeneous translation must fail. An illegal access error would be raised no matter in which package p.C<q.D> is placed. Since class accesses are checked when identifiers are resolved at run-time, the error would occur irrespective of whether classes are specialized at compile-time or run-time.

This problem makes it difficult to use packages effectively in the presence of the heterogenous translation. Further, even if one could change the JVM security model, it is not clear what change could fix this problem. The problem does not arise for the homogenous translation.

7 Restrictions

GJ's translation by type erasure requires some language restrictions which would not be necessary if a translation maintained types at run-time. The restrictions affect object and array creation, and casts and instance tests.

7.1 Object and array creation

A **new** expression where the type is a type variable is illegal. Thus, **new** A() is illegal, when A is a type variable. Such expressions cannot be executed because type parameters are not available at run-time. This is no great loss, since such generic creation is of limited value. Rather than create an object of variable type, one should pass in an object with a suitable method for creating new objects (commonly called a factory object).

A **new** expression where the type is an array over a type variable generates an unchecked warning. Thus, **new** A[n] is unchecked when A is a type variable. Such expressions cannot be executed with the usual semantics, since type parameters are not available at run-time. Rather than create arrays of variable type, it is recommended that one should use the Vector or ArrayList classes from the collection library, or pass in an array of the same type to be used as a model at run-time (a poor man's factory object).

To facilitate the latter, the following method is provided by the gj.lang.reflect.Array class.

```
public static <A> A[ ] newInstance (A[ ] a, int n)
```

A call returns a new array with the same run-time type as a, with length n and each location initialized to null. This method allows an array to act as a factory for more arrays of the same type. The erasure of the above method can be implemented in terms of existing reflection primitives as shown in Figure 8. But the types in the figure are not parametric, so the typed version is added to the GJ library. It can be implemented using the retrofitting feature discussed in Section 9.

For some purposes, such as defining Vector itself, it is necessary to create new arrays of variable type. This is why such expressions are unchecked rather than illegal. In this case the translation replaces the type variable by its bound, as usual. Thus, **new** A[n] translates to **new** Object[n], when A is a type variable bounded by Object.

Creating a new array of variable type must generate an unchecked warning to indicate that the type soundness constraints normally enforced by GJ may be violated. Consider the following code.

```
class BadArray {
    public static <A> A[ ] singleton (A x) {
        return new A[ ]{ x }; // unchecked warning
    }
    public static void main (String[ ] args) {
        String[ ] a = singleton("zero");
        // run–time exception
    }
}
```

This code passes the compiler, but an unchecked warning is issued for the expression **new** A[ ]{ x }. In this case, the creation expression does indeed violate GJ's type constraints, as when called with A bound to String it creates an array with run-time type Object[ ] rather than String[ ]. Here is the translation of the above code.

```
class BadArray {
    public static Object[ ] singleton (Object x) {
        return new Object[ ]{ x };
    }
    public static void main (String[ ] args) {
        String[ ] a = (String[ ])singleton("zero");
        // run–time exception
    }
}
```

It is important to recognize that the run-time type system of the JVM remains secure, as the last line in the translated code fails at run-time.

It is always safe to create a new array of variable type if one takes care to ensure the array does not escape the scope of the type variable. The method above is unsafe because the new array escapes the scope of the type variable A attached to the singleton method.

As an example of sensible use of arrays, consider the vector class given in Figure 9 (simplified from the collection library).

```
class Vector<A> {
    public final int MIN_CAPACITY = 4;
    protected int n;
    protected A[ ] a;
    public Vector () {
        n = 0;
        a = new A[MIN_CAPACITY];
    }
    public void add (A x) {
        if (n == a.length) {
            A[ ] b = new A[2*n];
            for (int i = 0; i < n; i++) b[i] = a[i];
            a = b;
        }
        a[n++] = x;
    }
    public A get (int i) {
        if (0 <= i && i < n) return a[i];
        else throw new IndexOutOfBoundsException();
    }
    public void set (int i, A x) {
        if (0 <= i && i < n) a[i] = x;
        else throw new IndexOutOfBoundsException();
    }
    public int size () { return n; }
    public A[ ] asArray (A[ ] b) {
        if (b.length < n) b = Array.newInstance(b,n);
        for (int i = 0; i < n; i++) b[i] = a[i];
        for (int i = n; i < b.length; i++) b[i] = null;
        return b;
    }
}
```

Figure 9: Vector class

The array a of type A[ ] always has run-time type Object[ ], but never leaves the scope of the class. The method asArray returns an array that leaves the scope of the class, but this array is either the argument array b (if b is large enough) or is an array with the same run-time type as b (created by newInstance). As usual, the code is translated by replacing A everywhere by Object, including replacing A[ ] by Object[ ].

7.2 Casts and instance tests

Since type parameters are not available at run-time, not all casts and instance tests on parameterized types are permitted. It is legal to include parameters in a cast or instance test if the parameters are determined by a combination of information known at compile-time and determinable at run-time.

```
class Convert {
    public static <A> Collection<A>
        up (LinkedList<A> xs) {
            return (Collection<A>)xs;
    }
    public static <A> LinkedList<A>
        down (Collection<A> xs) {
            if (xs instanceof LinkedList<A>)
                return (LinkedList<A>)xs;
            else
                throw new ConvertException();
    }
}
```

In method up, the cast could be omitted, but is included for clarity. In method down, run-time information can be used to check whether the collection is a linked list; if it is a linked list, then the compile-time constraints ensure that the type parameters match.

Parameterized types cannot be used in casts or instance tests when there is no way to verify the parameter. The following is illegal.

```
class BadConvert {
    public static Object up (LinkedList<String> xs) {
        return (Object)xs;
    }
    public static LinkedList<String> down (Object o) {
        if (o instanceof LinkedList<String>)
            // compile–time error
            return (LinkedList<String>)o;
            // compile–time error
        else throw new ConvertException();
    }
}
```

Here the marked lines indicate compile-time errors. There are two possible workarounds for this problem. One is to use type specialization, as in Section 6, creating a new class that extends LinkedList<String>.

```
class LinkedListString extends LinkedList<String> {...}
```

The other is to create a wrapper class, with a field of type LinkedList<String>.

```
class LinkedListStringWrapper {
    LinkedList<String> contents;
}
```

In either case, the resulting class has no type parameters, and may always be used as the target of a cast.

8 Raw types

It is occasionally necessary to refer to a parameterized type stripped of its parameters, which we call a *raw* type. Raw types maintain consistency with legacy code: for instance, new code may refer to the parameterized type Collection<A> while legacy code will refer to the raw type Collection. Raw types are also useful in cast and instance tests, where there may not be adequate information at run-time to check the full parameterized type.

```
class LinkedList<A> implements Collection<A> {
    ...
    public boolean equals (Object that) {
        if (!that instanceof LinkedList) return false;
        Iterator<A> xi = this.iterator();
        Iterator yi = ((LinkedList)that).iterator();
        while (xi.hasNext() && yi.hasNext()) {
            A x = xi.next();
            Object y = yi.next();
            if (!(x == null ? y == null : x.equals(y)))
                return false;
        }
        return !xi.hasNext() && !yi.hasNext();
    }
}
```

Figure 10: Equality using raw types

Figure 10 defines an extension to the linked list class of Section 2 to define equality. One might expect the object passed to equals to have the type LinkedList<A>, but a cast to that type cannot be checked, since type parameters are not available at run-time. However, it is possible to check a cast to the raw type LinkedList. Roughly speaking, the raw type LinkedList corresponds to the type LinkedList<B> for some indeterminate value of B. In this way, it resembles the existential types used in Pizza. But while Pizza's existential types are useful for writing methods such as equality, they are no help at all for interfacing with legacy code, which raw types do with ease.

In the above, the method call iterator() with receiver **this** of type List<A> returns a value of type Iterator<A>, while the same method with receiver (List)that of raw type List returns a value of raw type Iterator. Similarly, the method call next() with receiver xi of type Iterator<A> returns a value of type A, while the same method with receiver yi of type Iterator returns a value of type Object.

In general, the signature of a member of an object of raw type is the erasure of the signature of the same member for an object of parameterized type. Further, a value of parameterized type is assignable to a variable of the corresponding raw type. A value of raw type may also be assigned to a variable of any corresponding parameterized type, but such an assignment generates an unchecked warning.

Some method calls to objects of raw type must also generate unchecked warnings, to indicate that the type soundness constraints normally enforced by GJ may be violated. Consider the following code.

```
class Loophole {
    public static String loophole (Byte y) {
        LinkedList<String> xs =
            new LinkedList<String>();
        LinkedList ys = xs;
        ys.add(y); // unchecked warning
        return xs.iterator().next();
    }
}
```

This code passes the compiler, but an unchecked warning is issued for the call to the add method. In this case, the call does indeed violate GJ's type constraints, as it adds a byte y to the list of strings xs. Here is the translation of the above code.

```
class Loophole {
    public static String loophole (Byte y) {
        LinkedList xs = new LinkedList();
        LinkedList ys = xs;
        ys.add(y);
        return (String)xs.iterator().next();
            // run–time exception
    }
}
```

The run-time type system of the JVM remains secure, as the last line in the translated code fails at run-time.

The rules for generating unchecked warnings for raw types are:

- A method call to a raw type generates an unchecked warning if the erasure changes the argument types.
- A field assignment to a raw type generates an unchecked warning if erasure changes the field type.

No unchecked warning is required for a method call when only the result type changes, for reading from a field, or for a constructor call on a raw type. For example, in the equality test for linked lists given above, none of the raw method calls is unchecked, since they all have empty argument lists, so erasure leaves the type unchanged. But in the loophole method, the call to add is unchecked, since erasure changes the argument type from A to Object.

The unchecked method calls and field accesses may be needed to interface with legacy code, which is why they are not illegal. For example, one could compile the GJ versions of Collection<A>, Interface<A>, LinkedList<A> and Comparator<A> with the unparameterized version of Collections. The test code will compile, but generate a unchecked warning for the method calls to compare or compareTo, though in this case the calls happen to be sound.

The rule used by GJ to generate unchecked warnings is conservative. In practice, when interfacing legacy code to new GJ code, many calls may be labelled as unchecked that are nevertheless sound. Proliferation of unchecked warnings can be avoided by updating the legacy code, or by using the retrofitting technique discussed in the next section.

9 Retrofitting

To support independent compilation, the GJ compiler must store extra type information at compile-time. Fortunately, the JVM class file format supports adding extra attributes. Information about parameterized types is stored in a 'Signature' attribute, which is read and written by the GJ compiler, but ignored by the JVM at load-time.

GJ is designed so that new code will run with old libraries. For instance, new code may refer to a parameterized linked list type, but run with old code (source or binary) that implements an unparameterized linked list type using the generic idiom.

To make this work smoothly, the GJ compiler has a *retrofitting* mode that can be used to add 'Signature' attributes to existing code. Type information is specified in a source file that contains only type information for fields and methods. For instance, say one has a class file for the unparameterized version of LinkedList, but one wishes to use it as if it has parameterized types. This can be done using the following retrofitting file.

```
class LinkedList<A> implements Collection<A> {
    public LinkedList ();
    public void add (A elt);
    public Iterator<A> iterator ();
}
```

The GJ compiler takes the above file as source, and looks up the unparameterized class file along a specified classpath. It then outputs the new class file, including an appropriate 'Signature' attribute, in a directory specified by the user. (In the current GJ compiler, these are specified using the flags –retrofit *path* and –d *directory*). At compile-time, the classpath must specify the retrofitted class file. At run-time, the classpath may specify either the retrofitted or the legacy class file. In particular, new code can compile against the retrofitted linked list class file, then run in a browser containing the legacy linked list library.

The entire collection class library available in JDK 1.2 has been retrofitted in this way. All of the public methods in the JDK 1.2 collection classes — without a single exception — can be given sensible parameterized type signatures in GJ. Only the type signatures were rewritten, the legacy code did not even need to be recompiled. Since signatures are more than an order of magnitude more compact than code, this saves considerable effort.

In most cases, one would anticipate eventually rewriting the source library with parameterized types. The advantage of the compatibility offered by GJ is that one may schedule this rewriting at a convenient time — it is not necessary to rewrite all legacy code before new code can exploit parametric types.

We anticipate that most rewriting of code will be straightforward, consisting of adding type parameters and replacing some occurrences of Object by suitable type variables. However, not all code may be so easy to upgrade.

For instance, in the collection class library the implementation of finite maps includes code that may return either the key or value of a map entry. This is well-typed using the generic idiom with class Map, because both the key and value have type Object. But it is not well-typed using parameterized types with the class Map<K,V>, where the key has type K and the value has type V. So this portion of the code must be restructured to update the source to GJ, providing separate code to process keys and values. This need to restructure a (usually small) portion of the code shows why the flexibility of interfacing with legacy code offered by GJ is so helpful.

10 Implementation

GJ has been implemented and is publicly available from a number of web sites [GJ98a]. The GJ compiler is originally derived from the Pizza compiler, but has been substantially redesigned. It is itself written in GJ. Generic types and methods were essential in its implementation. For instance, the compiler makes heavy use of generic container types, such as linked lists, dictionaries, and iterators.

Besides these uses, the compiler also relies on generic methods for its central tree traversal routines, which are implemented using the visitor pattern [GHJV94]. The Pizza and GJ compilers are both structured as a series of passes over an abstract syntax tree. The Pizza compiler made extensive use of algebraic data types and pattern matching, which are supported in Pizza but not in the Java programming language. The syntax tree in the Pizza compiler is represented as an algebraic data type with a case for each of Pizza's syntactic constructs. Each pass consists of a recursive method with a case statement that pattern matches against all relevant cases in the tree type. It is thus possible to decouple the traversal algorithms from the tree definition itself. This makes sense since we would expect the language processors (implemented by traversal passes) to change more frequently than the language they process (represented by the tree itself).

In GJ, algebraic types and pattern matching are not available. Instead, the visitor pattern is applied to achieve an analogous program decomposition. Figure 11 gives an overview. There is an abstract class Tree with subclasses for each of GJ's syntactic constructs. In total, there are 38 such subclasses, although only one is shown. The base class and each subclass define a method visit, which takes a visitor object and applies a method in the visitor which corresponds to the subclass being defined. All such visitor methods use the overloaded name **_case**; they are distinguished by the subclass of Tree which they take as first argument. The abstract visitor class contains a **_case** method for each of the tree subclasses. Concrete subclasses override those **_case** methods that can possibly be encountered during traversal.

To make this standard idiom widely applicable, the visitor class is generic, with two type parameters. The R parameter stands for the result type of the **_case** methods in a concrete visitor. The A parameter stands for the type of an additional argument which those methods take. For instance, Figure 11 shows a fragment of the tree attribution visitor. Each **_case** method in that visitor takes an environment (of type Env<AttrContext>) as additional parameter and each method returns a type (of type Type). Other visitor passes in the compiler would use different argument and result types. Missing result types or argument types get instantiated to class Void. Multiple results or arguments are expressed using tuple types such as Pair.

Since visit in class Tree needs to be able to apply different parameterized instantiations of the visitor class, it needs to be polymorphic itself. Consequently, its type in Tree is:

<R,A> R *visit* (Visitor<R,A> v, A arg)

With this technique, the application of the visitor pattern in the compiler is quite natural. If one tried instead to apply the pattern in this form in a language without generics using the generic idiom, the abundance of required type casts would make the concept considerably harder to use. It is also worth noting that the use of polymorphic methods was essential to achieve a generic typing of visitors; parameterized types alone are not enough.

11 Conclusions

We have presented GJ, an extension of the Java programming language with generic types and methods.

```
abstract class Tree {
   public <R,A> R visit (Visitor<R,A> v, A arg) {
      return v._case(this, arg);
   }
   static class Return extends Tree {
      public Tree expr;
      public Return(Tree expr) { this.expr = expr; }
      public <R,A> R visit (Visitor<R,A> v, A arg) {
         return v._case(this, arg);
      }
   }
   static abstract class Visitor<R,A> {
      public R _case(Tree that, A arg) {
         throw new InternalError(
            "unexpected: " + that);
      }
      public R _case(Return that, A arg) {
         return _case((Tree)that, arg);
      }
      public R _case(Throw that, A arg) {
         return _case((Tree)that, arg);
      }
      // other cases ...
   }
}
public class Attr
   extends Tree.Visitor<Type,Env<AttrContext>> {
   ...
   public Type _case( Return tree,
                      Env<AttrContext> env) {
      Type owntype;
      // code for attribution of return statements ...
      return owntype;
   }
   // other attribution cases ...
}
```

Figure 11: Visitors in the GJ compiler

GJ is implemented by translating back to the unextended language, repeating the idiom used by programmers to simulate generics. For this reason, it is easy to interface GJ with legacy code, and it is straightforward to use reflection on GJ programs.

The design of Pizza is strongly constrained by the criterion of backward compatibility with the Java programming language. The design of GJ is further constrained by the criterion of smooth interfacing with legacy code, and of forward compatibility with a language design (such as NextGen [CS98]) that maintains information about type parameters at run-time. (Indeed, one referee characterized this paper as 'polymorphism with one hand tied behind your back'.) Remarkably, even though the constraints on GJ are tighter than those on Pizza, it's design is arguably simpler. GJ's inference algorithm is simpler than Pizza's, and GJ's use of raw types is simpler and more powerful than Pizza's use of existential types.

There are two main alternatives to the design pursued in GJ.

The first is to use the heterogenous translation. As we saw in Section 6, this alternative either makes it difficult to use packages effectively, or requires change to the security model of the JVM.

The second is to pass type information at run-time, as explored in the NextGen design of Cartwright and Steele [CS98]. GJ's forward compatibility makes it possible to arrange for NextGen to be a superset of GJ: every legal GJ program is also a legal NextGen program with an identical meaning. (The one exception is that NextGen, unlike GJ, changes some properties of a program under reflection.)

Both GJ and NextGen have advantages. NextGen is more expressive than GJ, in that none of the restrictions discussed in Section 7 need be imposed on NextGen. In particular, NextGen can implement **new** A[n] by allocating a new array with the correct run-time type information, avoiding the severe restrictions placed on this construct in GJ. And NextGen can implement an instance test or cast to a parameterized type such as LinkedList<String> without the workarounds required by GJ. Arguably, the use of run-time types in NextGen is a better fit with the Java programming language, which maintains run-time type information about the class of an object and the type of elements in an array.

On the other hand, GJ has a considerably simpler design that NextGen. And since GJ maintains no type information at run-time, it may be more efficient than NextGen, although measurement is required to determine if this difference is significant.

More importantly, GJ achieves greater compatibility than NextGen with legacy code. Not only is GJ backward compatible with the Java programming language and forward compatible with NextGen, but GJ also has backward and forward compatibility with legacy code. It has backward compatibility, in that legacy code using the generic idiom may call new parameterized libraries, and in that newly created objects of parameterized type may be passed to legacy code that uses the generic idiom. And it has forward compatibility, in that new parameterized code may call legacy libraries that use the generic idiom, and in that objects created by legacy code using the generic idiom may be passed to new code that expects objects of parameterized type. Roughly speaking, GJ achieves backward compatibility through raw types, and forward compatibility through retrofitting.

In contrast, NextGen has only backward compatibility. New code cannot use legacy libraries, and objects created by legacy code can be passed to new code only via adaptor methods that convert legacy objects (with no run-time type information) into NextGen objects (with run-time type information specified for each type parameter). The combination of forward and backward compatibility in GJ makes it considerably easier to manage the process of upgrading from legacy to parameterized code, and we believe that this is the chief advantage of GJ over NextGen.

Acknowledgements

Thanks to Enno Runne and Matthias Zenger, for their input on implementation and security aspects, and to Joshua Bloch, Corky Cartwright, and Guy Steele, for their support and many productive discussions. Thanks also to the members of the Java-genericity and Pizza-users mailing lists, for valuable criticism and continued feedback. Finally, thanks to the anonymous referees for their cogent comments.

References

[AFM97] Ole Agesen, Stephen Freund, and John C. Mitchell. Adding parameterized types to Java. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 215-230, 1997.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. *Functional Programming Languages and Computer Architecture*, pages 31–41, ACM, 1993.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 215-230, 1993.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. *European Conference on Object-Oriented Programming*, July 1998. (An earlier version was presented at *5th Workshop on Foundations of Object-Oriented Languages*, January 1998.)

[CCHOM89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. *Functional Programming Languages and Computer Architecture*, pages 273–280, ACM, 1989.

[CS98] Corky Cartwright and Guy Steele. Compatible genericity with run-time types for the Java programming language. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1998.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 169–184, 1995.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[GJ98a] Martin Odersky. The GJ compiler. Available from
www.cis.unisa.edu.au/~pizza/gj
wwwipd.ira.uka.de/~pizza/gj
www.math.luc.edu/pizza/gj
www.cs.bell–labs.com/~wadler/pizza/gj

[GJ98b] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ: the Java programming language with type parameters. Manuscript, 1998. Available at the GJ web site.

[GJ98c] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ Specification. Manuscript, 1998. Available at the GJ web site.

[GLS96] James Gosling, Bill Joy, and Guy Steele. *The Java language specification*. Java Series, Sun Microsystems, ISBN 0-201-63451-1, 1996.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine specification*. Java Series, Sun Microsystems, ISBN 0-201-63452-X, 1996.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

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

[OR98] Martin Odersky and Enno Runne. Measuring the cost of parameterized types in Java. Research Report CIS-98-004, Advanced Computing Research Centre, University of South Australia, January 1998.

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

[PT98] Benjamin C. Pierce and David N. Turner. Local Type Inference. *Symposium on Principles of Programming Languages*, pages 252–265, ACM, 1998.

[RS97] Paul Roe and Clemens Szyperski. Lightweight Parametric Polymorphism for Oberon. *Proceedings Joint Modular Languages Conference*, Johannes Kepler University Linz Schloß Hagenberg Austria, March, 1997 http://www.fit.qut.edu.au/~szypersk/Gardens/

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

[Tor98] Mads Togersen. Virtual types are statically safe. *5th Workshop on Foundations of Object-Oriented Languages*, January 1998.

[TT98] Kresten Krab Thorup and Mads Togersen. Structural virtual types. Informal session on types for Java, *5th Workshop on Foundations of Object-Oriented Languages*, January 1998.

[Wri95] A. Wright, Simple imperative polymorphism, *Lisp and Symbolic Computation*, 8:343–355, 1995.