GJ: Extending the Java™ programming language with type parameters

Gilad Bracha, Sun Microsystems
Martin Odersky, University of South Australia
David Stoutamire, Sun Microsystems
Philip Wadler, Bell Labs, Lucent Technologies

March 1998; revised August 1998

Say you wish to process collections. Some may be collections of bytes, others collections of strings, and yet others collections of collections of strings. The Java programming language supports such variation by allowing you to form a collection of `Object`, so the elements may have any reference type. In order to keep the language simple, you are forced to do some of the work yourself: you must keep track of the fact that you have a collection of bytes, and when you extract an element from the collection you must cast it to class `Byte` before further processing.

This situation is becoming more common as the Java platform evolves, notably with the addition of collection classes to JDK 1.2. Other languages provide additional support for this situation: in C++, it is supported with templates; in Ada, it is supported with generics; and in ML and Haskell, it is supported with parametric polymorphism.

This note proposes GJ, an extension to the Java programming language that supports types with parameters. GJ programs look much like the equivalent Java programs, except they have more type information and fewer casts. The semantics of GJ is given by a translation 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 if generics weren't available. The translation is designed so that new GJ code will work with existing Java libraries, even when the libraries are available only in binary class file form.

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 be necessary in certain cases for purposes of compatibility.) Furthermore, since GJ translates into Java virtual machine (JVM) byte codes, all safety and security properties of the Java platform are preserved. (Reassurance: this second guarantee holds even in the presence of unchecked warnings.)

In pathological cases, the translation requires bridge methods that must be encoded directly as JVM byte codes. Thus GJ extends the expressive power of the Java programming language, while remaining compatible with the JVM.

GJ is backward and forward compatible with the Java programming language and the JVM.

- *Compatible with the Java programming language.* Every Java program is still legal and retains the same meaning in the extension to GJ. This is true for both Java source code and JVM class files.
- *Compatible with JVM.* GJ compiles into JVM code. No change to the JVM is required. The code is verifiable and can be executed on any JDK compliant browser.
- *Compatible with existing libraries.* One can compile a program that uses a parameterized type against existing class file libraries for an unparameterized version of that type. For instance, one can use parameterized collections with the existing collections library.
- *Transparent translation.* There is a straightforward translation from GJ into the Java programming language. The translation leaves field and method names unchanged, and bridge methods are easy to predict. It is therefore easy to use reflection with GJ. The translation introduces minimal overhead, so program performance remains easy to predict.
- *Efficient.* GJ is translated by erasure: no information about type parameters is maintained at run-time. This means GJ code is pretty much identical to Java code for the same purpose, and equally efficient.
- *Futureproof.* Greater flexibility may be achieved by carrying type parameters at run-time, and this may be possible in future implementations. GJ is designed so it smoothly extends to include run-time type parameters.

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, like Pizza, implements parametric types by erasure. A similar idea has been proposed and implemented for 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, CS97, MBL97]. Run-time types may be less efficient to implement than erasure, and may be harder to interface with legacy code. As noted above, GJ is designed to extend smoothly if it is later judged practicable to use run-time types.

Virtual types have been suggested as an alternative to parametric types for increasing the expressiveness of types [Tho97, Tor98]. A comparison of the relative strengths of parameteric 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.

This paper is intended as a readable introduction, motivating the features of GJ. A companion paper provides a more precise specification [BOSW98].

This paper is structured as follows. Sections 1–3 introduce the basic features of GJ, using a running example based on collections and linked lists. Section 4 describes how bridge methods must be expressed by direct translation into the JVM. Section 5 explains why an invariant subtyping rule is used for parameterized types. Section 6 explains type inference for parametric method calls. Section 7 discusses object and array creation. Sections 8–10 discuss casts and techniques that allow GJ to fit with legacy code.

1 Generic classes

The first example demonstrates the fundamentals of generic classes. For this example and the following ones, we first consider Java code for a task, then see how it is rewritten in GJ.

Here are interfaces for collections and iterators, and a linked list class (all much simplified from the Java collections library).

```
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 ();
        }
      };
    }
  }
```

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, and contains a nested class for list nodes and an anonymous class for the list iterator. Each element has type `Object`, so one may form linked lists with elements of any reference type, including `Byte`, `String`, or `LinkedList` itself.

Here is code utilizing linked lists.

```
  class Test {
    public static void main (String[] args) {

      // byte list
      LinkedList xs = new LinkedList();
      xs.add(new Byte(0)); xs.add(new Byte(1));
      Byte x = (Byte)xs.iterator().next();

      // string list
      LinkedList ys = new LinkedList();
      ys.add("zero"); ys.add("one");
      String y = (String)ys.iterator().next();

      // string list list
      LinkedList zss = new LinkedList();
      zss.add(ys);
```

```
    String z = (String)((LinkedList)zss.iterator().next()).iterator().next();

    // string list treated as byte list
    Byte w = (Byte)ys.iterator().next();  // run-time exception
  }
}
```

The user must recall what type of element is stored in a list, and cast to the appropriate type when extracting an element from a list. Extracting an element from a list of lists requires two casts. If the user accidentally attempts to extract a byte from a list of strings, this raises an exception at run-time.

Now, here are collections, iterators, and linked lists rewritten in GJ.

```
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 ();
      }
    };
  }
```

```
}
```

The interfaces and class take a type parameter `A`, written in angle brackets, representing the element type. Each place where `Object` appeared in the previous code, it is now replaced by `A`. Each place where `Collection`, `Iterator`, or `LinkedList` appeared in the previous code, it is now replaced by `Collection<A>`, `Iterator<A>`, or `LinkedList<A>` (the one exception being the declaration of the class constructor). The nested class `Node` has `A` as an implicit parameter inherited from the scope, the full name of the class being `LinkedList<A>.Node`.

Here is the test code rewritten in GJ.

```
class Test {
  public static void main (String[] args) {

    // byte list
    LinkedList<Byte> xs = new LinkedList<Byte>();
    xs.add(new Byte(0)); xs.add(new Byte(1));
    Byte x = xs.iterator().next();

    // string list
    LinkedList<String> ys = new LinkedList<String>();
    ys.add("zero"); ys.add("one");
    String y = ys.iterator().next();

    // string list list
    LinkedList<LinkedList<String>> zss
      = new LinkedList<LinkedList<String>>();
    zss.add(ys);
    String z = zss.iterator().next().iterator().next();

    // string list treated as byte list
    Byte w = ys.iterator().next();  // compile-time error
  }
}
```

Instead of relying on the user's memory, parameters document the type of each list's elements, and no casts are required. The code to extract an element from a list of lists is more perspicuous. Now an attempt to extract a byte from a list of strings indicates an error at compile-time.

To translate from GJ into 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 `Byte` erases to `Byte`) and the erasure of a type parameter is `Object` (so `A` erases to `Object`). A suitable cast is inserted around each method call where the return type is a type parameter. Translating the GJ code for lists yields the Java code we started with (except for the line that was in error). Thus, a new program compiled against the GJ code could be used with an old library compiled against the Java code.

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 be omitted.

Angle brackets were chosen for type parameters since they are familiar to C++ users, and since they avoid confusion that may otherwise arise. 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, there is no worry for the user, the problem is instead solved by a slight complication to the grammar.

2 Parametric methods and bridges

The second example demonstrates a parametric method and illustrates the need for bridges.

Here is an interface for comparators, a comparator for bytes, and a class with a static method to find the maximum of a collection (all based on the Java collections library).

```
interface Comparator {
  public int compare (Object x, Object y);
}
class ByteComparator implements Comparator {
  public int compare (Object x, Object y) {
    return ((Byte)x).byteValue() - ((Byte)y).byteValue();
  }
}
class Collections {
  public static Object max (Collection xs, Comparator c) {
    Iterator xi = xs.iterator();
    Object w = xi.next();
    while (xi.hasNext()) {
      Object x = xi.next();
      if (c.compare(w,x) < 0) w = x;
    }
    return w;
```

```
    }
  }
```

The comparator interface specifies a method that takes a pair of objects, and returns an integer that is negative, zero, or positive if the first object is less than, equal to, or greater than the second. In the byte comparator class, the two objects are cast to bytes, and then subtracted to generate a comparison value. The last class defines a static method that accepts a collection and a comparator and returns the maximum element in the collection (if it is non-empty).

Here is code utilising comparators.

```
class Test {
  public static void main (String[] args) {

    // byte list with byte comparator
    LinkedList xs = new LinkedList();
    xs.add(new Byte(0)); xs.add(new Byte(1));
    Byte x = (Byte)Collections.max(xs, new ByteComparator());

    // string list with byte comparator
    LinkedList ys = new LinkedList();
    ys.add("zero"); ys.add("one");
    String y = (String)Collections.max(ys, new ByteComparator());
                                          // run-time exception
  }
}
```

As before, the user must keep track of the result type and cast the results as appropriate. Using a string collection with a byte comparator raises an exception at run-time.

Now, here are comparators and maximum rewritten in GJ.

```
interface Comparator<A> {
  public int compare (A x, A y);
}
class ByteComparator implements Comparator<Byte> {
  public int compare (Byte x, Byte y) {
    return x.byteValue() - y.byteValue();
  }
}
class Collections {
  public static <A> A max (Collection<A> xs, Comparator<A> c) {
    Iterator<A> xi = xs.iterator();
    A w = xi.next();
    while (xi.hasNext()) {
      A x = xi.next();
```

```
      if (c.compare(w,x) < 0) w = x;
    }
    return w;
  }
}
```

The interface now has a type parameter `A`, and the compare method takes two arguments of type `A`. The byte comparator class implements the comparator interface with the type parameter instantiated to `Byte`, and so the compare method takes two arguments of type `Byte`, and no casts are required. In the last class, the static method `max` takes a type parameter, indicated by writing `<A>` before the method signature; the type parameter `A` appears in the result type, argument types, and method body.

Here is the test code rewritten in GJ.

```
class Test {
  public static void main (String[] args) {

    // byte list with byte comparator
    LinkedList<Byte> xs = new LinkedList<Byte>();
    xs.add(new Byte(0)); xs.add(new Byte(1));
    Byte x = Collections.max(xs, new ByteComparator());

    // string list with byte comparator
    LinkedList<String> ys = new LinkedList<String>();
    ys.add("zero"); ys.add("one");
    String y = Collections.max(ys, new ByteComparator());
                                      // compile-time error
  }
}
```

As before, no casts are required. Now using a string collection with a byte comparator indicates an error at compile-time.

Type parameters are inferred for a parametric method call as follows: the smallest type parameter that yields a valid call is chosen. In the example above, the call to `max` with a byte collection and a byte comparator infers that the type parameter `A` is `Byte`. If there is no unique smallest type, inference fails, as in the call to `max` with a string collection and a byte comparator. An extra rule, described in Section 6, supports inference when there are no argument types involving an inferred variable, or when the argument is `null`.

To translate from GJ one erases types and adds casts as before. The `Comparator` interface and `Collections` class in GJ translate back to the Java source code given above. The `ByteComparator` requires a little more cleverness.

```
class ByteComparator implements Comparator {
```

```
    public int compare (Byte x, Byte y) {
      return x.byteValue() - y.byteValue();
    }
    // bridge
    public int compare (Object x, Object y) {
      return this.compare((Byte)x, (Byte)y);
    }
  }
```

A *bridge* method must be introduced, since overriding occurs only when types match exactly. Overloading allows the bridge and the original method to share the same name. In general, a bridge is introduced whenever a class implements a parameterized interface or extends a parameterized class at an instantiated type. A problematic case arises if the overridden method has a parameterized type but no arguments, as discussed in Section 4.

A class can implement an interface in only one way. Thus, a class cannot implement the parameterized interfaces `I<S>` and `I<T>` unless `S` is identical to `T`. Similarly for the superinterfaces of an interface.

3 Bounds

The third example demonstrates that type parameters may be bounded.

As an alternative to comparators, one may specify that the object itself contains a suitable comparison method. Here is an interface for comparable objects, the standard byte class that implements this interface, and a variant method to find the maximum of a collection (all based on the Java collections library).

```
  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;
  }
}
```

Whereas a comparator defines a method `compare` that takes two objects, a comparable class defines a method `compareTo` that takes one object, which is compared with the method's receiver. The standard `Byte` class implements the `Comparable` interface (as of JDK 1.2). It defines two version of `compareTo`, one which expects a `Byte`, and one which expects an `Object`; the second of these overrides the method in the interface. The last class defines a static method that returns the maximum element in the collection (if it is non-empty and all the elements in it are comparable.

Here is code utilising comparable objects.

```
class Test {
  public static void main (String[] args) {

    // byte collection
    LinkedList xs = new LinkedList();
    xs.add(new Byte(0)); xs.add(new Byte(1));
    Byte x = (Byte)Collections.max(xs);

    // boolean collection
    LinkedList ys = new LinkedList();
    ys.add(new Boolean(false)); ys.add(new Boolean(true));
    Boolean y = (Boolean)Collections.max(ys);  // run-time exception
  }
}
```

As before, the user must keep track of the result type and cast the results as appropriate. Booleans do not implement the comparable interface, so an attempt to find the maximum of a collection of booleans raises an exception at run-time.

Now, here are comparable objects rewritten 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;
    }
  }
```

In `Byte`, the `compareTo` method now only needs to be defined at type `Byte`, the definition for `Object` being automatically derived by the insertion of a bridge method. In `max` the type parameter is constrained by a bound, `A implements Comparable<A>`, indicating that the type variable should only be instantiated by a class that is comparable with itself. It is legal to apply `max` at type `Byte` because `Byte implements Comparable<Byte>`. In the body of `max` it is permitted to call the `compareTo` method since the receiver and argument are both of type `A`, and `A` implements `Comparable<A>`.

Here is the test code rewritten in GJ.

```
  class Test {
    public static void main (String[] args) {

      // byte collection
      LinkedList<Byte> xs = new LinkedList<Byte>();
      xs.add(new Byte(0)); xs.add(new Byte(1));
      Byte x = Collections.max(xs);

      // boolean collection
      LinkedList<Boolean> ys = new LinkedList<Boolean>();
      ys.add(new Boolean(false)); ys.add(new Boolean(true));
      Boolean y = Collections.max(ys);  // compile-time error
    }
  }
```

Now an attempt to find the maximum of a collection of booleans indicates an error at compile-time.

In general, a bound is introduced by following the parameter with `extends` and a class, or `implements` and an interface. The bounding class or interface may itself be parameterized, and may include type variables appearing elsewhere in the parameter section; recursion or mutual recursion between parameters is allowed. Omitting a bound is equivalent to using the bound `Object`.

The definition of erasure is extended so that the erasure of a type variable is the erasure of its bound (so `A` erases to `Comparable` in `max`). The GJ versions of `Comparable`, `Byte`, and `Collections` translate back to the Java source code given above.

4 A bridge too far

A problematic case of bridging may arise if the overridden method has a identical arguments but different return types.

Here is the iterator interface in GJ (adopted from the collections library), and a class that implements that interface.

```
inteface Iterator<A> {
  public boolean hasNext ();
  public A next ();
}
class Interval implements Iterator<Integer> {
  private int i;
  private int 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.

Translating from GJ 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;
  private int 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.

This loophole points out an underspecification of the Java programming language and the JVM. Since legal JVM code may in fact contain more than one implementation of a method with the same name and the same argument types, it needs to be specified which of these is returned by `Class.getMethod`. (Tests show the Sun and Microsoft implementations differ.)

GJ also permits covariant overriding: an overriding method may have a result type that is a subtype of the method it overrides. This is lllegal in the Java programming language, which requires that the result type be identical in the overriding method. Note that both languages require that the argument types be identical. Here is an example that is legalin GJ.

```
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.

5 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 object) and the return (which extracts a string from a list of string) are unobjectionable, so it must be the assignment (which aliases a list of string to a list of object) 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] = x;  // 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.

Observe that explicitly declared subtyping is not a problem. For instance, it is fine to pass a `LinkedList<String>` when a `Collection<String>` is expected.

6 Inference

Recall that type parameters are inferred for a parametric method call as follows: the smallest type parameter that yields a valid call is chosen. This rule does not apply when there is no unique smallest type to choose, as may happen if there are no argument types involving an inferred variable or if some argument is `null`.

To support inference in these cases, the type inferencer may bind a type variable to the special type `*`. This type is used only by the type inference algorithm, and cannot appear in type declarations in GJ programs. The type `*` is a subtype of every other type. Further, any type containing `*` is regarded as a subtype of any type that results from replacing * with any other 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>. The type of null is taken to be *.

The inference rule then applies as before, with the smallest type parameter that yields a valid call being chosen. Consider the following method declarations and calls.

```
class Test {
  public static <A> LinkedList<A> empty () {
    return new LinkedList<A>();
  }
  public static <A> LinkedList<A> singleton (A x) {
    LinkedList<A> xs = new LinkedList<A>();
    xs.add(x);
    return xs;
  }
  public static void main (String[] args) {
    LinkedList<Byte> xs = empty();
    LinkedList<String> ys = singleton(null);
  }
}
```

In both calls in main, the type parameter is inferred to be *. The assignments are valid side LinkedList<*> is a subtype of both LinkedList<Byte> and LinkedList<String>.

One additional restriction is required: a type parameter cannot be instantiated to * if it appears more than once in the result type. Consider the following code.

```
class Cell<A> {
  public A value;
  public Cell (A v) { value = v; }
  public static <A> Cell<A> allocate (A x) {
    return new Cell(x);
  }
}
class Pair<A,B> {
  public A fst;
  public B snd;
  public Pair (A x, B y) { fst = x; snd = y; }
  public static <A> Pair<A,A> duplicate (A x) {
    return new Pair<A>(x,x);
  }
}
class Loophole {
  public static String loophole (Byte y) {
    Pair<Cell<String>,Cell<Byte>> p =
      Pair.duplicate(Cell.allocate(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.allocate((String)null));
    p.fst.value = x;  return p.snd.value;
  }
}
```

The call to `duplicate` in `loophole` is illegal, because the smallest choice for `A` is `*`, but `A` appears twice in the result type of `loophole`. On the other hand, the call to `duplicate` in `permitted` is ok, because the cast ensures the smallest choice for `A` is `String`. But without the cast, the smallest choice is `*` and the call would be illegal.

7 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, since 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` class 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 in the `gj.lang.reflect.Arrays` 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.

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

But the types here are not parametric, so the typed version is added to the GJ library. It can be implemented using the retrofitting feature discussed in Section 10.

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 recognise that the run-time type system of JVM remains secure, as the marked line fails at run-time.

However, it is 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, here is a vector class (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 A 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 = Arrays.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;
    }
  }
```

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[]`.

8 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 `up` the cast could be omitted, but is included for clarity. In `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.

A workaround is to create a wrapper class.

```
class Convert {
  private class Wrapper {
    private LinkedList<String> value;
    public Wrapper (LinkedList<String> value) { this.value = value; }
    public LinkedList<String> value () { return value; }
  }
  public static Object up (LinkedList<String> xs) {
    return new Wrapper(xs);
  }
  public static LinkedList<String> down (Object o) {
    if (o instanceof Wrapper) return ((Wrapper)o).value();
    else throw new ConvertException();
  }
}
```

While one cannot test the parameter type at run-time, one can test whether an object belongs to class `Wrapper`, which contains a field of parameterized type.

9 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.

Here is an extension to the linked list class of Section 1 to define equality.

```
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();
  }
}
```

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 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 remains secure, as the marked line fails at run-time.

The rules for generating unchecked warnings for raw types are as follows:

- 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 Java 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.

10 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 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.

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.

References

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

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ Specification. Draft document, 1998.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. *5th Workshop on Foundations of Object-Oriented Languages*, January 1998.

[CS97] Corky Cartwright and Guy Steele. Yet another parametric types proposal. Message to Java genericity mailing list, August, 1997.

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

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

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

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In *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.