Parametric Polymorphism for Java: Is There Any Hope in Sight?

Brian Cabana, Suad Alagić, and Jeff Faulkner
Department of Computer Science
University of Southern Maine
Portland, ME 04104-9300
bcabana@maine.rr.com, alagić@cs.usm.maine.edu, faulkner@cs.usm.maine.edu

ABSTRACT

In spite of years of research toward a solution for the problem of extending Java with parametric polymorphism (genericity) the officially accepted solution already in its beta release allows violation of the Java type system and turns a type safe language into an unsafe one. The run-time type information in this release is incorrect which leads to major problems for the programmers relying on the Java reflective capabilities. We show that there are two basic reasons for these problems. The first one is that the idiom underlying this solution is provably incorrect. The second one is that the problem of extending Java with parametric polymorphism does not have a correct solution unless the Java Virtual Machine is extended to handle it properly. This paper elaborates the subtleties required by a correct implementation technique that includes representation of parametric classes in the standard Java class file format, representation of the instantiated parametric class objects, extensions of the Java Core Reflection to report type information about (instantiated) parametric classes, and the loading techniques required by this solution for extending Java with generics. Previous solutions for this problem are analyzed as well.

Keywords

Parametric polymorphism, generics, Java Core Reflection, Java Virtual Machine, class files, class objects, loading.

1. INTRODUCTION

The problem of extending Java with parametric polymorphism (generic classes and interfaces) is well-known. It has attracted a fair amount of research and implementation efforts, and a number of solutions have been proposed and/or implemented. Some of the significant implementations include PolyJ [19], Pizza [20], GJ [11], NextGen [12], and LM Translator [24]. This problem also carries clear pragmatic and industrial implications, and was the impetus behind JSR 14 [9], [10], a Java Specification Request for the addition of generics to the Java programming language. And finally, an official solution has emerged already in the beta release of JDK 1.5.

In this paper we show that this solution has unforgivable pitfalls. We explain what the problems are and where they are coming from. We analyze the limitations of the idiom underlying this implementation. We also demonstrate that this problem does not have a correct solution unless the Java Virtual Machine is extended appropriately. Other existing solutions of this problem are also analyzed in this paper to show why they are in fact unacceptable. Details of a correct implementation technique are developed. Nontrivial subtleties of the underlying Java technology that make this problem so difficult are analyzed.

The working model for Java programmers for writing generic classes in the absence of parametric polymorphism plays an important role in most implementation techniques for extending Java with generics. However, relying heavily on this generic Java idiom is in fact one of the reasons why the problem of extending Java with parametric polymorphism still lacks a correct solution.

The generic Java idiom is illustrated below along with its alternative, using the Java type system extended with bounded parametric polymorphism, specified as comments.

```
// class SortedList<T implements Comparable>
class SortedList{

     // private T [] elements;
     private Comparable [] elements;

     // public SortedList<T>()
     public SortedList() {
          //elements = new T[size];
          elements = new Comparable[size];
     }

     // public int add(T t)
     public int add(Comparable t);

     // public T remove(int index)
     public Comparable remove(int i);

     public int size();
 }
```

By subtype polymorphism a static check will ensure that a sorted list contains elements whose type is a subtype of the type `Comparable`. The implication is that processing the objects of a specific subtype of `Comparable` requires type casts, as illustrated by the code below. This code also demonstrates the difference between the generic idiom and parametric polymorphism.

While the use of type casts may seem merely an inconvenience and results in unattractive code, there are potentially serious problems. Since any object whose type is a subtype of `Comparable` can be placed in the collection, the runtime type check may fail and result in an exception. Runtime type checks also increase the overhead of using collections.

Use of parametric classes solves this problem. The parametric `SortedList` class is instantiated in the code below with a

specific type parameter, `Integer`. A system that supports parametric polymorphism would interpret that only `Integer` objects are allowed in the collection. The code can be statically type checked, and there would be no need for runtime type checks and casts, a major advantage from the viewpoint of efficiency and reliability.

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

     // SortedList<Integer> list =
     //  new SortedList<Integer>();
     SortedList list = new SortedList();
     List.add(new Integer(1));
     //continue adding to list
     // Integer i = list.remove(0);
     Integer i = (Integer)list.remove(0);
     //do stuff with i
}
```

2. GJ LIMITATIONS

In Java, classes are compiled into class files of a specific structure. These class files can be loaded over the web and the source files may not be available. In a technique called homogeneous translation, a generic class file, and even the class object may be shared by all instantiated classes. This generic class file is constructed according to the generic Java idiom explained above in section 1. Pizza [20] and GJ [11] use this technique for implementing parametric classes in Java.

In GJ, a parametric type is erased to a non-parametric raw type by substituting each occurrence of a type parameter by its bound type. This way the type parameters are erased altogether. When compiling a client class, references to the parametric class are replaced by references to the class obtained by type erasure and the required type casts are inserted automatically. Note that this method does not reduce the problem of using type casts; it just removes the necessity of the application programmer manually inserting them. In addition, if both the type parameter and the bound type appear in the signatures of fields and methods of a parametric class, type erasure will identify the two, which creates additional semantic problems [3], [21].

This technique has a rather obvious problem. In the Java type system, the type `SortedList<Integer>` which is obtained by instantiation of `SortedList<T implements Comparable>` could never be derived from the class `SortedList`, the raw type which is all that exists after type erasure. Consider the instance variable `elements` which is declared to be of type `T[]` in the definition of SortedList<`T implements Comparable`>. As a result of type erasure, the `elements` instance variable would have a type `Comparable[]`, whereas the type of `elements` in `SortedList<Integer>` should be `Integer[]`. However, in Java, no subclass of `SortedList` could ever override `elements` to have the type `Integer[]`. The same can be said of method signatures.

The strength of GJ and one reason it has been accepted as the implementation of parametric polymorphism by Sun [10] is its compatibility with legacy code through its use of raw types. Another advantage is that its homogenous translation results in no increase in disk space or memory requirements. Since the homogeneous translation occurs at compilation, runtime performance is not affected.

However, there are additional limitations of GJ that are caused by its use of type erasure idiom, some of which are described in [4], [5], [6] and elaborated in other papers by a co-author of this work [2], [3], [21]. For instance, since the actual type parameters are not available at runtime, it is not possible to perform casts or type checks on type parameters. In general, the type parameters cannot be treated as first class types, one of the requirements of JSR 14.

One particularly unfortunate problem of the pure homogeneous translation is that the run-time type information as reported by Java Core Reflection is incorrect. For the instantiated class `SortedList<Integer>` the Java Core Reflection will report that its type is `SortedList<Comparable>`. This creates nontrivial practical problems as many packages rely on the correctly reported types when introspected by Java Core Reflection. For all these reasons, GJ fails to meet the goals of JSR14.

3. JDK 1.5 PITFALLS

The implications of the above limitations of the GJ approach to extending Java with parametric polymorphism appear in JDK 1.5 which is based on this solution. In this section we present the most important pitfalls in the beta release of JDK 1.5

3.1 Violations of the Java Type System Possible Because of Type Erasure

Type erasure eliminates the actual type parameters to produce a raw type. Because of this instances of the actual type parameters appear as instances of type `Object` or some other bound type. By subtype polymorphism objects of any type (or a subtype of the bound type) may now be assigned to instances of the actual type parameters. The implication is that objects of completely unrelated types may be assigned to each other.

In the example below an instantiated parametric type is `Vector<Integer>` and so only `Integer` values should be inserted into an object of `Vector<Integer>` type. In the invocation `myMethod(v)` the actual parameter is of type `Vector<Integer>`. The `instanceof` test in `myMethod` asserts that the actual parameter is of type `Vector` so the most specific type cast possible is `(Vector)collection` rather than `(Vector<Integer>)collection`. This makes it possible to add instances of type `String` to the vector v which is of type `Vector<Integer>`. This is a complete violation of the type system and yet type checking succeeds. This defeats completely the purpose of the type system. Confusion of the runtime type system is further illustrated by an attempt to retrieve this element from the vector. This internally leads to an attempt to type cast an object of type `String` to type `Integer`. This attempt of course fails with a `ClassCastException` error.

```
public class MyClass {

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

```
      Vector<Integer> v = new Vector<Integer>();
      v.add(new Integer(9));
      myMethod(v);
      Integer i = v.lastElement();
   }
   public static void myMethod(Object collection)
   {
      if (collection instanceof Vector) {
         ((Vector)collection).add("trouble"));
      }
   }
}
```

3.2 Subtyping Rules are Violated

Type erasure makes it possible to violate the rule that an instance of a subtype may be substituted where an instance of a supertype is expected and not the other way around. The code given below (like in [13]) assigns an object of type `Vector` to an object of type `Vector<Integer>`. `Vector` is the Java standard utility class and it is by no means a subtype of the type `Vector<Integer>` in the Java type system, even when it is extended with parametric polymorphism in any reasonable way.

```
Vector<Integer> a = new Vector<Integer>();
Vector b = new Vector();
b.add(new Object());
a = b;
Integer i = a.remove(0);
```

Notice that a perfectly valid attempt that satisfies the static typing requirements to remove an Integer object from a vector of type `Vector<Integer>` fails and produces a `ClassCastException`.

3.3 Overloading does not Work

Type erasure leads to all instantiated parametric types with completely different actual type parameters to be identified as the same raw type. This makes different method signatures that satisfy Java overloading rules identical and hence rejected by the compiler. This is illustrated by the example below.

```
public class MyClass {
     public void myMethod(Vector<Integer> list) {}
     public void myMethod(Vector<String> list) {}
}
```

The type erasure produces two identical method signatures for `myMethod` which violates the Java overloading rules contrary to the above perfectly valid user class which thus can not be compiled. How would one explain this to a Java programmer? Since the type erasure identifies the bound type and the type parameter the following perfectly valid example of a parametric class with a bounded type constraint will not compile either.

```
public class LinkedList <G extends Comparable> {
     public void myMethod(G g) {}
     public void myMethod(Comparable c) {}
}
```

Note that in all instantiations of the `LinkedList` class in which the type parameter `G` is not replaced by the bound type `Comparable` the rules of overloading will in fact be satisfied. In a typical instantiation the actual type parameter will not be `Comparable` but its subtype. It is also obvious to any programmer that the type parameter `G` should not be replaced by `Comparable` because the overloading rules would be violated.

3.4 Java Core Reflection does not Work

Type erasure leads to incorrect runtime type information. Because of this, using Java Core Reflection leads to very unfortunate errors illustrated in the example below. Because of type erasure the signature `void myMethod(Vector<String> value)` is recorded in the class objects as `void myMethod(Vector value)`. Invocation of Java Core Reflection method `getMethod` with the wrong signature produces a positive answer. `getMethod` takes the name of the method ("myMethod") and the list of class objects of the parameter types. But in `c.getClass().getMethod("myMethod",v.getClass())` `v.getClass()` is of type `Vector<Integer>`. A user misled by this incorrect runtime type information would now invoke the method with a `Vector<Integer>` parameter. This is of course a complete violation of the typing rules because `Vector<Integer>` is not a subtype of `Vector<String>` by any means. Furthermore you can invoke this method in accordance to its wrongly reported parameter type. So the user of Java Core Reflection would naturally expect this invocation to succeed. In spite of the fact that Java Core Reflection would allow the invocation of this method, the invocation would fail when attempting to get a `String` object out of the vector that now contains an `Integer` object.

```
public class MyClass {
   public void myMethod(Vector<String> value) {
      String s = value.lastElement();
   }
   public static void main(String[] args) {
       MyClass c = new MyClass();
       Vector<Integer> v = new
                           Vector<Integer>();
       Method method = c.getClass().getMethod(
                    "myMethod", v.getClass());
       v.add(new Integer(123));
       method.invoke(c, v);
   }
}
```

4. TOWARD A CORRECT SOLUTION

In this section we explain the subtleties in the Java Virtual Machine that must be addressed in order to produce a correct solution for parametric polymorphism in Java.

4.1 The Java Class File

A fundamental component of the JVM is the class file structure. The whole Java platform depends critically on this structure defined in the JVM Specification [18]. Portability,

interoperability across the net, compatibility with the legacy systems, dynamic loading and generally correct functioning of the JVM depend upon maintaining this predefined class file structure. This is why the fundamental component of our technique is a representation of a parametric class or interface in the standard class file format. Preserving the Java class file format is one of the most important considerations in maintaining compatibility with existing JVMs.

Previous implementations of parametric polymorphism make varying degrees of modification to the class file structure. PolyJ [19] and load time instantiation [1] both modify the class file to such an extent that it would no longer be recognized by an unmodified JVM's class loader. At the other end of the spectrum, GJ modifies the class file by adding a signature attribute that provides information used in retrofitting classes but does not provide runtime type information.

The Java class file contains all the information necessary for the JVM to load and create objects of a particular class. This information includes the declared fields and methods of the class, the code of the declared methods, the constant pool (a symbol table for the class), and the names of the super class and interfaces of the class. A class file is loaded by the class loading system, its format is verified to be correct, and it is converted to the runtime representation of the class. The format is specified in the *Java Virtual Machine Specification* [18].

The problem is that the Java class file structure was never designed with parametric classes in mind. This is why a suitable representation of a parametric class in a standard Java class file is a nontrivial issue. The first problem is to find a way of representing type parameters and their bounds in the Java class file structure. The second problem is instantiating parametric classes, i.e. replacement of the formal type parameters with the actual ones.

The type signatures of fields and methods are encoded in the Java class file as type descriptors according to the rules specified by a context free grammar in the *Java Virtual Machine Specification*. The type descriptors in the class file for SortedList obtained by type erasure for the `elements` field, the `add` method and the `remove` method are given below:

```
[Ljava/lang/Comparable;
(Ljava/lang/Comparable;)I
(I)Ljava/lang/Comparable;
```

In the above notation [denotes the array, L<classname>; indicates an object type and I stands for the integer type. In the instantiated class object of `SortedList<Integer>` the above signatures would have the following form.

```
[Ljava/lang/Integer;
(Ljava/lang/Integer;)I
(I)Ljava/lang/Integer;
```

Additionally, the type parameters or names of other parametric types can be found in the bodies of methods. These are later translated to different types of entries in the constant pool, to which the byte code of methods will refer. Therefore, the instantiation of a parametric class must ensure that the type descriptors are properly encoded and that the code correctly references the appropriate constant pool entries. This is a major implementation issue in this technique.

In order to represent the information about type parameters and their bound types in the Java class file structure, the flexibility offered by optional attributes is exploited in this paper. Attributes in the Java class file structure are attached to classes, methods, and fields. In addition to the standard, predefined attributes, the Java Virtual Machine Specification allows for optional attributes and requires that these optional attributes be silently ignored by the JVM [18]. Our previous work [2] describes the use of optional attributes to encode the type descriptors in the parametric class. A single parametric class attribute contains the parametric type information for the class and its fields and methods, as well as the information to encode the transformations needed to instantiate the class. This information is included in such a way that a legacy JVM would have no problem recognizing or loading the class file.

4.2 Class Loading

Unlike the traditional compile, link and run model, our solution is targeted to the Java dynamic model which includes dynamic loading and compilation. Instantiation of a parametric class is thus performed "just-in-time" which means at load time, just like in the solution for extending C# with parametric polymorphism [16]. An instantiation of a generic class is performed by an extended class loader that transforms the information in the generic class and loads its instantiated class object.

Each class has a reference to the class loader that loaded it and that loader is referred to as the defining class loader. The defining class loader is used to load any classes referenced by that class, following the parent delegation model. Parent delegation simply means that the parent class loader is first given the opportunity to load the class if it can, before the defining loader tries to load it.

For example, given an instance Lm of a user defined class loader MyLoader (extends ClassLoader) used to load a class C, Lm is the defining class loader for class C. This means a reference to a class D from class C is loaded by Lm. Unfortunately, this does not guarantee that Lm will actually load the class. A call to load a class D by Lm results in Lm calling `loadClass` on its parent class loader Lp, which then continues to delegate until the System Class Loader attempts to load class D. If the attempt fails, then Lm can attempt to load the class by calling `findClass("D")`. If the parent class loader is successful in loading class D, then Lp becomes the defining loader for class D [17]. This means that class D is in the namespace of the parent class loader as depicted in Figure 1.

The parent-delegation model has serious implications when attempting to use an extended class loader to load parametric classes. First of all, the loading of an instantiated parametric class by the system class loader must be prevented so that the extended class loader can load and instantiate the class. Not only do parametric classes need to be loaded by the extended class loader, any classes that reference a parametric class also need to be defined using the extended class loader. A better but also more demanding solution is to extend the native class loader with the ability to load and instantiate parametric classes, as described in the implementation section.

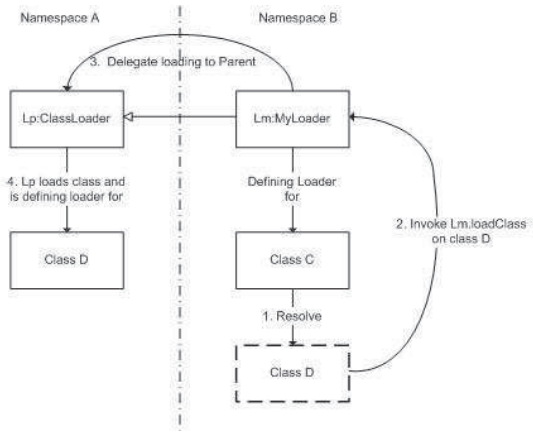

Figure 1: Parent Delegation Example

4.3 Java Core Reflection

Java Core Reflection is a feature of the Java language that allows introspection into the structure of a class. Many existing APIs, such as Java RMI, Enterprise Java Beans, XML parsers and persistence mechanisms to name a few, make use of reflection to inspect classes and invoke methods, making JCR an important and integral feature of the Java environment. But we showed that in GJ and JDK 1.5 run time type information in the class objects of instantiated parametric classes is incorrect as the bound types appear where the actual type parameters should. This should be compared to the technique for extending C# with parametric polymorphism [16] which provides exact run-time type information. In general, Java Core Reflection should be preferably extended by additional methods that extract the type information pertinent to parametric classes.

Java Core Reflection consists of a collection of final classes, `Class`, `Method`, `Field`, etc. Since these classes are final classes, their extensions are possible only by recompilation of the whole platform which does not present any serious difficulties. These extended methods operate on the internal Class object structure which must be also extended accordingly. Finally the loading system must be extended so that it will load the relevant information from the class files into the extended class objects. We established that it is possible to make these changes in such a way that there is no impact on the correct functioning of the JVM.

5. IMPLEMENTATION

In this section we elaborate full details required by a correct implementation technique. The components of this implementation technique are representation of parametric classes in class files, extensions of the loading system and class objects, and extensions of Java Core Reflection.

5.1 Class File Representation

A type parameter is represented in a class file as a triple (name index, bound index, actual index). Name index is an index to a constant pool entry which contains the name of the formal type parameter. The bound index determines a constant pool entry which identifies the bound class. Likewise, actual index determines the actual type. Figure 2 shows part of a class file for `SortedList<T implements Comparable>` that serves to illustrate the changes needed to the *Constant Class* entries and their corresponding name indices. There is a single parameter which is represented by the `parameters` field of the parametric class attribute. The `this_class` field points to a *Constant Class* entry which has a name index pointing to a *Constant UTF8* entry with the name `SortedList`. The super class is Object.

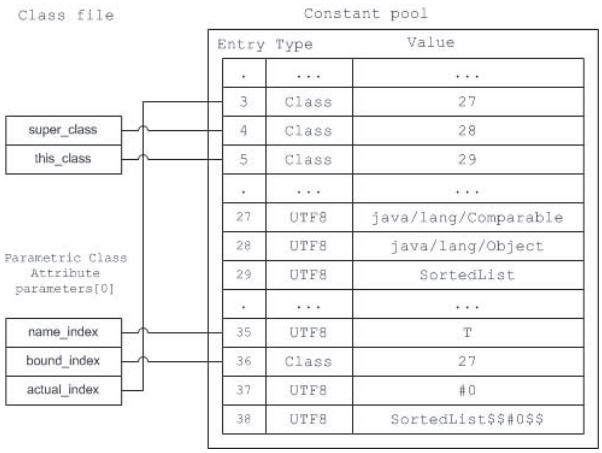

Figure 2: Class SortedList<T implements Comparable>

The bound type of `T` is *Constant Class* entry 36, which has a name index of 27, `java/lang/Comparable`. The actual type is also set to `java/lang/Comparable` but under a different *Constant Class* entry (3). Although at this point, the bound and actual types are the same, these two *Constant Class* entries must be kept distinct so that when instantiated, the actual type will in general be different from the bound type. Class file attributes that need to have a reference to the bound type even after instantiation still need to refer to that *Constant Class* entry (36). The class file features that reference the formal parameter need to reference the *Constant Class* entry (3) representing the actual type. The constant pool entries at 37 and 38 are the special *Constant UTF8* entries that encode the type parameter name and instantiated class name. The "#0" is a placeholder that will contain the class name of the actual type parameter at instantiation. The "0" refers to the index in the parameters array of the parametric attribute which will be discussed below. These entries have no references to them prior to instantiation so a standard class loader will load this class with no complaints.

Figure 3 shows this same segment of the class file after instantiation. The constant pool entries 37 and 38 reflect an instantiation of the parametric class `SortedList<Integer>`. The name index of the *Constant Class* entry (3) is changed from 27 to 37, the *Constant UTF8* entry that contains the name of the actual type. *Constant Class* entry (5) is changed from 29 to 38 to reflect the instantiated class name.

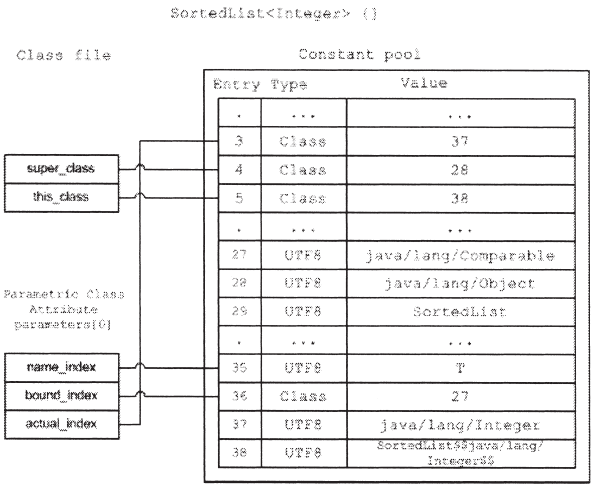

Figure 3: Class SortedList<Integer>

Additionally, field and method descriptors also refer to *Constant UTF8* entries, and if these descriptors refer to type parameters they are modified to contain the actual type descriptors during instantiation. For instance, the `elements` field of the `SortedList` class is a parametric field. The generic class file contains the descriptor "[Ljava/lang/Comparable;" to represent the bound type of the field. The constant pool will also have the entry "[L#0;". Again, upon instantiation, the #0 is replaced by the actual type and the descriptor field of the `Field_info` entry of the class's field array will be redirected to reference the new descriptor. Method descriptors in the class are also represented in this fashion. The motivation behind this arrangement was to allow a generic class to be loaded by a standard class loader; these additional entries would never be referenced. However, if the parametric class is instantiated with the actual type parameters by an extended class loader, these entries would be used by the class to represent the actual types. Even with an extended class loader, unless the JVM is modified as described later in the paper, these parametric types would be considered just as any other type.

5.2 Class Object Representation

Representation of parametric classes in Java class files is based on the work reported in [2]. An optional attribute is introduced for parametric class files to encode information about type parameters and the transformation necessary to instantiate the class. Since attributes not recognized by the JVM are ignored, adding a parametric attribute has no effect on a legacy JVM. The representation of the class object in the JVM also requires an extension to contain the information on the type parameters at runtime.

The access flags field of the class is used to indicate that a class is parametric. The access flags field of the class contains bits which indicate whether a class is public, private, final, abstract or an interface. An unused bit was found and established as the `ACC_GENERIC` modifier which indicates a parametric class when set. The JVM was also modified to recognize the presence of the flag.

The definition of the JVM class object is contained in the `Classjava_lang_Class struct`, hereafter referred to as the `Class struct`. A pointer to a `parametricattribute struct` must be added to the end of the `Class struct`. This allows access to the parametric information contained in the `parametricattribute struct` by the Java Core Reflection classes. Here is where this approach differs from many other approaches that utilize other structures to access type information. In this implementation, the type information is contained with all other class information.

In the Java class file, attributes are attached to classes, methods, and fields. The information about type parameters as specified by the parametric attribute must be associated with all of them. With type information now represented in the class file and in the JVM Class structure, there remains the problem of loading the information about type parameters from the class file into the extended class object as represented in the `Class struct`.

The internal Class structure is created from the class file's byte representation by the JVM function `createInternalClass`. The function reads the byte array representation of the class and constructs the various parts of the `Class struct`. As it reads the bytes that represent the class, it reads the name and length of each attribute. If the attribute has a name that the JVM is supposed to act upon, it passes the control off to a function that constructs the appropriate attribute from the bytes. If not, it skips the number of bytes defined by the length field of the attribute and continues loading the class. The JVM was modified to recognize the name of the parametric attribute and transfer control to a new function that creates the `parametricattribute` structure from the byte stream.

While this method involves extending the structure of the JVM Class objects and recompiling the JVM, classes compiled using the previous JVM versions will still be correctly represented by the new virtual machine. The parametric type information is also contained more naturally within the Class information that is internal to the JVM and may be accessed through JCR. This has significant advantages in making the Class objects persistent as reported in [2].

5.3 Extending Java Core Reflection

The relevant information for a programmer using parametric classes would obviously include whether the class is parametric, what the names and the bound types of the type parameters are, as well as what the actual type parameters are for instantiated parametric classes. This explains what extensions of Java Core Reflection are required for parametric classes. Therefore *`java.lang.Class`* is extended by the following methods.

```
public native boolean isParametric()
public String[] getTypeParameters()
public Class[] getBoundTypes()
public Class[] getActualTypes()
```

The above extensions are actually comparable to what is available in JDK 1.5 with a very important difference; there is no way to get the actual type parameters by Java Core Reflection in JDK 1.5.

The `isParametric` method is a native call that simply tests for the `ACC_GENERIC` flag in the class's access flags. The

`getTypeParameters` method first tests that the class is parametric. If it is parametric, a native method call gets a `String` array for the type parameter identifiers. The method `getBoundTypes` returns an array of `Class` objects of the bound type of each parameter. Finally, the method `getActualTypes` would return the array of `Class` objects of the actual types of the instantiation.

The class `java.lang.reflect.Modifier` contains methods to decode access flags and method and field modifiers. An additional method is required to test if a modifier contains the `ACC_GENERIC` flag.

The resulting structure of the modified Java Core Reflection extends that of the legacy JVM, and the performance of this part of the extended JVM would be comparable with no effect on the correctness of legacy code.

5.4 Class Loading

There are two approaches taken in this project to instantiate parametric classes from their generic class file representation described above. The result of instantiation is a loaded class object representing the instantiated class.

The first approach is to develop an extended class loader that would load the instantiated class object with the actual parameters specified as part of the class name. An extended class loader was developed and tested successfully. The chief advantage of the extended class loader is that no further modifications to the JVM are necessary for the scheme to work. In this case, the JVM would treat the loaded parametric classes as any other loaded class. The drawback is that due to the parent-delegation model of Java class loading described previously, the developer of software that would take advantage of parametric classes would have to ensure that all classes that reference parametric classes would be loaded by the extended class loader. This is so the classes would appear in the same namespace as the parametric classes they reference. It would also mean that system classes, such as *Collection* classes, could not be made parametric.

The solution to this problem is to modify the native class loader. By modifying the native class loader, parametric classes could be located and loaded without relying on an extended class loader. The changes to the native loader require it to do the following.

1. Recognize a parametric class from its mangled class name as found in the constant pool of the referencing class.
2. Parse the mangled name into the generic class name and the actual parameters.
3. Find the generic class and load it.
4. Modify the class as defined by the parametric attribute.

The class loader system is convoluted with a number of classes and native methods participating in the class loading process. The *java.net.URLClassLoader* has the main responsibility of loading classes found in the class path. Since this class is written in Java, the modifications to this class are similar to the code found in our initial extended class loader. Another major participant in the class loading process is the system class loader, a set of JVM methods, many of which are native, that load Java core classes.
In this project the *URLClassLoader* has been extended in order to be able to load instantiated class objects from class files. The main modification, the addition of a `findParametricClass` method, enables it to find the class given the mangled class name and then delegate the construction of the class to a set of classes designed to modify it using the parametric attribute. A package *classfilestruct* developed specifically for this purpose does the work of translating the bytes of the generic class to the bytes of an instantiated class. The class loading process continues as normal from there. The sequence is shown in Figure 4 and is described below.

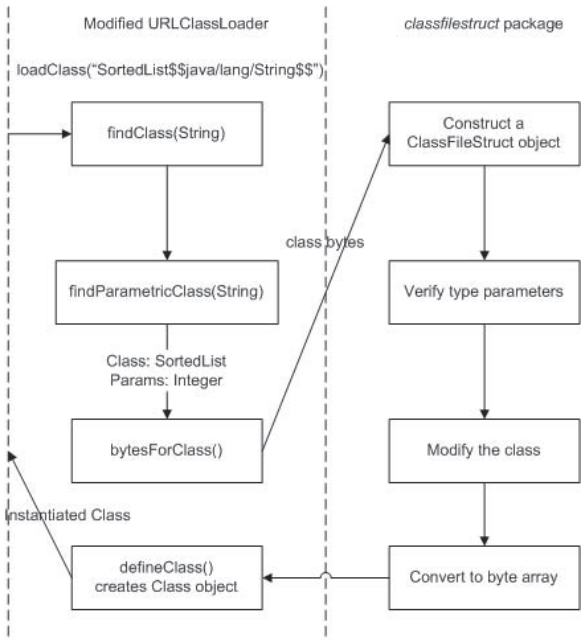

Figure 4: Parametric Class Loading Sequence

The basic operation of the modified class loader is as follows:

1. The `loadClass` method is called on the modified URLClassLoader. Since the `loadClass` method is not overridden, the default `loadClass` of the `ClassLoader` class is invoked.
2. After failing to find the parametric class, due to its mangled name consisting of the generic name and the type parameters, the `findClass` method is called on the modified URLClassLoader.
3. `findClass` first attempts to load a parametric class by calling `findParametricClass`.
4. `findParametricClass` parses the mangled class name into the generic class name and type parameters. It then loads the generic class file into a byte array.
5. The byte array is passed the the *classfilestruct* package which represents the generic class as a `ClassFileStruct` object.
6. The actual types are verified to be subtypes of the bound types. This may involve loading other classes.

7. The instantiation is performed by modifying the `ClassFileStruct` object using the type parameters and parametric attribute.
8. The `ClassFileStruct` object is converted back to a byte array and passed to the `defineClass` method that constructs and returns the Class.

For the system class loader, native C code was developed to perform the same function as the *classfilestruct* package, to modify the bytes of an instantiated parametric class prior to defining the class object. These modifications to the native class loader allow any Java class, especially system classes which would not be loaded by an extended class loader, to be made parametric. Testing showed that the modifications to the class loader were successful, but also revealed nuances in the Java class loading system that must be further addressed. For instance, the *Collection* class could only take as a type parameter other Java system classes, such as *String* and *Integer*. This is due to the system loader only being aware of the location of system classes. This can be corrected by allowing the *URLClassLoader* to be used in the verification of type parameters.

6. ANALYSIS OF OTHER TECHNIQUES

Several techniques extend the simple homogeneous translation underlying GJ and JDK 1.5 with features of heterogeneous translations. LM Translator [22], [23] extends a homogeneous translation by using type descriptors to maintain information about the actual type parameters of the instantiated classes. Because the type descriptors are maintained at runtime, JCR could be modified to exploit that information, although this has not been done [23]. However, the type information of a parametric type is encoded directly into the class as static fields and methods, in essence changing the class. JCR would report these extra fields although they do not appear in the source code of the parametric class. There is also the additional cost of look-ups and method redirection.

6.1 Heterogeneous Translations

NextGen [12] uses a modified form of heterogeneous translation to address several shortcomings of GJ. A parametric class is compiled to an abstract base class, similar to the raw type of GJ. The compiler generates a lightweight wrapper class and interface for each instantiation of a parameterized class. The wrapper class inherits type-erased methods from the abstract base class and the wrapper interface contains the type information for the instantiation [12]. The usage of wrapper classes and interfaces provides correct run-time type information. Further implementation details and benchmarks are presented in [8], and the follow up work in [7] extends this technique to mixins.

However, this technique is a fairly complex representation of the simple and well understood idea of instantiation of a parametric class. Instantiation simply means substitution of the actual type parameters for the formal ones. In contrast, this technique creates a completely unexpected and confusing run-time situation for a user performing introspection by Java Core Reflection and expecting a single class object for an instantiated parametric class. Instead, JCR would report the wrapper interface, its implementing class and the abstract base class as well as the presence of the snippet methods. Not only is the user's type hierarchy represented by a more complex one, but this technique requires a number of fixes by snippets to handle a variety of problematic situations especially when this triple of types participates further in the inheritance hierarchy. For example, matching this representation with the standard Java collection class library causes problems [8]. Type casts are not eliminated either.

A heterogeneous technique reported in [1] is based on the compilation of parametric classes into a modified class file format followed by load time instantiation. The modified class file format contains information about the type parameters in a header. Each instantiation with different actual parameters would create a different class object in memory although the file system would contain one parametric class file. This implementation also removes the need for type casts. The main drawback of this technique is that the class file generated this way is not a valid class file; an extended class loader is needed to load the parametric class due to the incompatible nature of the class file. Another drawback is that the JVM is not modified to allow JCR to accurately reflect on the fact that these are generic or instantiated generic classes.

PolyJ [19] also uses load time instantiation, but provides a different mechanism for specifying the bound type [14]. While it has some advantages, changes needed to the class file cause it to break some of the requirements of JSR 14.

6.2 Reflective Technique

Particularly relevant to our approach is a reflective technique [4], [21] that recognizes the limitations of the generic Java idiom. It is based on the requirement that the run-time type information for instantiated parametric classes must be correct so that it would be reported as such by Java Core Reflection. An important implication of this requirement is that if the class object of an instantiated parametric class is promoted to persistence, it will contain correct type information. JCR would be even able to recognize the fact that there are parametric classes in the system and would report correctly the information of those classes [3]. The essence of this technique is proper management of type descriptors for fields and methods in the Java class files and usage of optional class file attributes to record correctly the information about type parameters and their bounds. All of this is done without disturbing the standard Java class file structure to ensure compatibility with standard JVM [2]. This technique was the starting point for investigating all the implementation subtleties required by a correct solution as reported in this paper.

6.3 C# Technique

The C# technique is also targeted to virtual machine environments where the traditional compile, link and run model is replaced with a more dynamic model with dynamic loading and even compilation with "just-in-time" (load-time) instantiation of parametric classes. The C# technique is also particularly important because it provides correct (exact) run-time type information.

Regrettably and contrary to the expectations based on the similarity between C# and Java, it is hard to apply any of the techniques for extending C# with generics to Java. The main reason is found in the differences between the two virtual machines, which play a major role in extending the two environments with genericity.

The whole Java platform depends critically upon the class file structure defined in the JVM Specification [18]. This is why the fundamental component of our technique is a representation of a parametric class or interface in the standard class file format. The C# technique naturally has nothing to do with this Java specific requirement. The other fundamental difference is in the specifics related to the internal representation of Java class objects. This internal representation must be maintained in order for the JVM to function properly. Java Core Reflection accesses this representation to report run-time type information. Load-time instantiation of parametric classes must produce correct class objects. Dynamic dispatch depends heavily on the class object representation.

There exists a fundamental agreement between the C# technique and our technique in the observation that the virtual machine must be adapted in order to be able to handle a variety of issues related to parametric classes or else no good solution for extending the language with parametric polymorphism will be available. However, there exists a huge difference in the views on what kinds of changes to the virtual machine are acceptable. The C# extension with parametric polymorphism is based on liberties that are simply not acceptable for Java.

The language of the virtual machine (CLR) has been changed by adding new types to the intermediate language type system, by introducing parametric forms of intermediate language declarations along with ways of referencing fields, classes, interfaces and methods, and by specifying new instructions as well as generalizing the existing ones to the parametric forms. Such modifications were used with PolyJ [19], which extended the Java bytecode with the *invokewhere* and *invokestaticwhere* instructions. Unfortunately this Java implementation had the serious drawbacks of not being backward compatible due to its highly modified class file.

This explains why our solution has been developed under much more restrictive assumptions: the instruction set, the run-time type introspection, dynamic dispatch, and class file structure all remain intact. The extensions in the Java Core Reflection, internal class object representation and the loading system have no impact on the correct functioning of the Java legacy systems. Contrary to the intuitive expectations, the C# solution is simply not a solution for extending Java with generics.

6.4 Code Multiplicity

In the technique presented in this paper, each instantiation of a generic class is itself a complete class. The generic class, in fact, does not ever appear in memory unless loaded by an unmodified class loader. Dealing with code multiplicity is a thorny problem, and all solutions have negative implications. GJ has no code multiplicity; the cost is the requirement of type casts and lack of correct runtime type information. NextGen uses inheritance from the abstract base class to reduce code multiplicity, but results in two class objects in memory for each instantiation in addition to the class object for the abstract base class. Code multiplicity is avoided by redirecting the methods from the wrapper (implementing) class to the base class. The base class contains the generic code and the implementing class contains type specific code [8]. Other drawbacks of this technique were analyzed in Section 6.1

The .NET implementation for C# [16] utilizes the fact that objects consist of a reference to a vtable, which provides for the method dispatch, followed by the fields of the object. The solution for avoiding code multiplicity in the C# solution amounts to duplicating the method dispatch tables (vtables) and attaching the type instantiation information to those tables, while the code remains the same, as illustrated by the following figure:

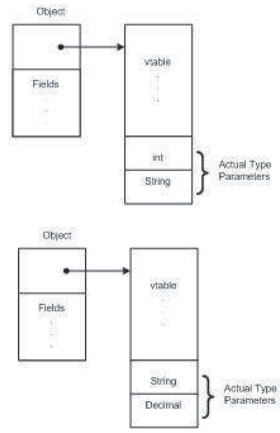

Figure 5: C# implementation of Parametric Polymorphism

However, implementing this approach in the JVM is practically impossible without far reaching changes to the JVM. This in particular applies to separating the dispatch tables from the code for methods or executing the code produced by the Java generic idiom for methods that involve objects of instantiated parametric classes. The C# technique includes a non-trivial scheme based on type dictionaries to keep track of the actual types of such objects. It is unclear how one would go about incorporating such a schema in the JVM.

The central issue in Java is that the code from the generic class can be reused with the specific constant pools of each instantiation. The changes made to the class by this implementation only include the constant pool and not the methods themselves. Therefore, when code references a position in the constant pool, that position would be filled with the correct information no matter what the instantiation. If there was a way of dispatching a method with a certain constant pool to reference, it would solve the problem. This requires a sort of special dispatch that the JVM is not prepared to utilize. Modifications to give the JVM this sort of capability would be extensive.

The question is whether the implications of implementing this far reaching extension of the JVM are worth the savings in memory. Consider that bootstrapping the JVM utilizes about 4.5 MB and the average *Collection* class requires about 8 KB of memory. The amount of memory saved would depend on the relative sizes of the constant pool and the methods. If we can assume a savings of 4 KB per instantiation, ten instantiations of a collection would

result in a 1% savings over the total memory used by the JVM. And that does not take into account the heap used by the objects in the application. We would argue that this is a reasonable cost to pay at this point in order to produce a correct solution for parametric polymorphism in Java.

7. CONCLUSIONS

The main conclusion of this paper is that there can be no correct solution for extending Java with genericity unless the Java Virtual Machine is extended appropriately. Furthermore we showed that this extension is possible without compromising the integrity of the existing Java Virtual Machine.

The implementation technique developed in this paper has several advantages over prior reported efforts. The classes instantiated from the generic classes are first class types in that they can be used anywhere a standard Java type can be used. The formal parameters can be used anywhere a standard Java type is used, a significant advantage over JDK 1.5. The classes that use parametric classes can be statically type checked and do not need type casts to be inserted. The generic classes could be used by legacy programs according to the generic Java idiom. Parametric type information is contained by the instantiated parametric classes in memory and can be accessed via Java Core Reflection.

A major advantage of this technique is that loaded class objects contain correct run-time type information, unlike JDK 1.5. This information will be reported correctly by Java Core Reflection. In addition, when objects of an instantiated parametric class are promoted to persistence the associated persistent type information in their class objects will of course be correct. Going one step further, to have Java Core Reflection recognize and report the information on parametric classes and their instantiation requires an extension of JCR implemented in this project. This extension along with the other extensions requires recompilation of the whole Java platform, but legacy packages will not be affected in any way.

Our final remark is that one of the goals of this paper was to demonstrate why the problem of extending Java with parametric polymorphism is so difficult. Our hope is that the analysis presented in the paper along with all the implementation details indeed shows all the subtleties involved in a correct implementation technique required by this non-trivial, well-known and important problem which really has not had a good solution so far. Regrettably, JDK 1.5 does not offer it either.

8. REFERENCES

[1] Agesen, O., Freund, S.N. and Mitchell, J.C. Adding Type Parameterization to the Java Language. *ACM Special Interest Group on Programming Languages* 1997, 49-65.

[2] Alagić, S. and Nguyen, T. Parametric polymorphism and orthogonal persistence. Proceedings of the ECOOP 2000 Symposium on Objects and Databases, *Lecture Notes in Computer Science* 1944, 2000, 32-46.

[3] Alagić, S. Solórzano, J. and Gitchell, D. Orthogonal to the Java imperative. Proceedings of ECOOP 1998, *Lecture Notes in Computer Science* 1445, 212-233.

[4] Allen, E. A Guide to Generics in the Java Tiger version and the JSR-14 prototype compiler. IBM Developer Works, February 2003. http://www-106.ibm.com/developerworks/java/library/j-djc02113.html

[5] Allen, E., Overcoming the Limitations in the JSR-14 Compiler. IBM Developer Works, April 2003 http://www-106.ibm.com/developerworks/java/library/j-djc04093.html

[6] Allen, E., Some Limitations of Generics in the JSR-14 Prototype Compiler. IBM Developer Works, March 2003 http://www-106.ibm.com/developerworks/java/library/j-djc03113.html

[7] Allen, E., Bannet, J., Cartwright, R. A First-Class Approach to Genericity. *Proceedings of OOPSLA* 2003, 96-114

[8] Allen, E., Cartwright, R., Stoler, B. Efficient Implementation of Run-time Generic Types for Java. *Generic Programming 2002*, 207-236

[9] Bracha, G. and the Expert Group. *Java Specification Request 14: Add Generic Types to the JavaTM Programming Language*. http://jcp.org/jsr/detail/14.prt

[10] Bracha, G., Cohen, N., Kemper, C., Marx, S., Odersky, M., Panitz, S.E., Stoutamire, D., Thorup, K. and Wadler, P. *Adding Generics to the Java Programming Language: Participant Draft Specification*. April 27, 2001. http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html

[11] Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P. Making the future safe for the past: Adding Genericity for the Java Programming Language. *Proceedings of OOPSLA* 1998, 183-200.

[12] Cartwright, R. and Steele Jr., G.L. Compatible Genericity with Run-time Types for the Java Programming Language. *Proceedings of the OOPSLA* 1998, 201-218.

[13] Close, S., Using Java Generics. *Java Developers Journal.* November 2003, 32-37, 60

[14] Day, M., Gruber, R., Liskov, B. and Myers, A.C. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. *Proceedings of OOPSLA* (Austin TX, October 1995), 156-168.

[15] Gosling, J., Joy, B., Steele, G. and Bracha, G. *The Java Language Specification, Second Edition. The Java Series*. Addison-Wesley, 2000.

[16] Kennedy, A. and Syme, D. Design and Implementation of Generics for the .NET Common Language Runtime. *Proceedings of PLDI* , 1-12, ACM 2001.

[17] Liang, S. and Bracha, G. Dynamic Class Loading in the Java Virtual Machine. *ACM SIGPLAN Notices, Proceedings of OOPSLA* 1998, 36-44.

[18] Lindholm, T. and Yellin, F. *The Java Virtual Machine Specification, Second Edition. The Java Series*. Addison-Wesley, 2000.

[19] Meyers, A.C., Bank, J.A. and Liskov, B. Parameterized Types for Java. *Proceedings of the 24th ACM Symposium on Principles of Programming Languages* (Paris, France, January 1997), 132-145.

[20] Odersky, M. and Wadler, P. Pizza into Java: Translating theory into practice. *Proceedings of the 24th ACM Symposium on Principles of Programming Languages* (Paris, France, January 1997), 146-159.

[21] Solórzano, J. and Alagić, S. Parametric polymorphism for Java: A reflective solution, *Proceedings of OOPSLA* 1998, 216-225.

[22] Viroli, M. From FGJ to Java according to LM translator, *Workshop on Formal Techniques for Java Programs, ECOOP* 2001.

[23] Viroli, M. Parametric Polymorphism in Java: an Efficient Implementation for Parametric Methods. *Proceedings of the 2001 ACM symposium on Applied Computing*, 2001, 610-619.

[24] Viroli, M. and Natali, A. Parametric Polymorphism in Java: an Approach to Translation Based on Reflective Features. *Proceedings of OOPSLA* 2000, 146-160.