Poor Man's Genericity for Java

Boris Bokowski and Markus Dahm

Freie Universität Berlin
Institut für Informatik
Takustr. 9, 14195 Berlin
{bokowski,dahm}@inf.fu-berlin.de

Abstract. A number of proposals have been made as to how Java can be changed to support parameterized types. We present a new proposal that does not try to provide more powerful constructs or cleaner semantics, but instead minimizes the changes that need to be made to existing Java compilers. In particular, we found that changing only one method in Sun's Java compiler already results in a reasonable implementation of parameterized types, which we call "Poor Man's Genericity" (PMG).
We have implemented our solution based on simple byte–code transformations both at compile–time and at load–time. The paper explains how our solution works, and compares it to other proposals. We also describe how the drawbacks of our approach can be overcome by making additional, but minimal changes to an existing Java compiler.

1 Introduction

Recently, a number of proposals for adding parametric polymorphism (generic classes) to Java [5] have been published, namely Pizza [7], GJ [2], Virtual Types [8], Genja [4], a proposal from the MIT [6], and a proposal from Sun and Stanford University [1]. These proposals differ in a number of aspects:

1. the suggested syntax extensions,
2. the expressiveness that can be achieved,
3. the translation scheme being used,
4. the level of integration with the Java type system, in particular, whether basic types may be used as actual type parameters,
5. the resulting runtime performance, and
6. implementation issues.

In this paper, we concentrate on the last aspect; in particular, we aim at minimizing the changes that need to be made to existing Java compilers. Each of the proposals that have been made so far requires a new compiler to be written, or an existing compiler to be modified extensively. Our approach yields a much cheaper implementation than any of the other proposals in terms of how many modifications to an existing Java compiler are needed. Particularly, in the

case of Sun's Java compiler, only one method needs to be changed for compiling parameterized classes.[1]

We think that trying to implement parametric polymorphism by making as few modifications as possible to an existing Java compiler is worthwile for two reasons: First, it is interesting to see that the language feature "parametric polymorphism" can be implemented in a way such that it is orthogonal to the rest of the language. Second, our solution shows an easy–to–follow upgrade path for incorporating parametric polymorphism in Java. Today, a large number of Java compilers and integrated development environments exist already. When introducing a new language feature like parametric polymorphism into Java, one important consideration for Sun should be to minimize the effort for other compiler and tool vendors to implement that feature.

The main idea of Poor Man's Genericity is to modify the way in which the compiler loads byte–code files ("`.class`" files), employing byte–code transformation to generate instances of generic classes dynamically, both at compile–time and at load–time. It was inspired by the load–time expansion technique used in the proposal of Sun and Stanford University [1]. By introducing so–called placeholder types, this idea can be used at compile–time as well. The idea can be applied to Sun's compiler as well as to any other Java compiler.

The basic idea of our implementation is described in section 2. Section 3 discusses the properties of the resulting parametric polymorphism. As will be explained in section 4, our solution has a number of drawbacks, which is why we call it "Poor Man's Genericity". However, we explain how these drawbacks can be overcome by further changes to the compiler, all of which are simple and local changes. In section 5, we compare our proposal with related work. Section 6 concludes the paper.

2 Basic idea

The problem of compiling parameterized types can be split in two parts, namely, compiling code that defines parameterized types, and compiling code that makes use of parameterized types. Before we explain our solutions to these problems, we give a short introduction explaining what parameterized types are and how we would like to define and use such types.

2.1 Parameterized Types

Parameterized types, a feature of object–oriented programming languages that is considered to be missing from Java, are a mechanism that allow classes or interfaces to be parameterized with other types. By providing actual type parameters, parameterized types can be used like any other user–defined type. A consequence of the fact that Java currently does not allow parameterized types

[1] Our implementation is available at `http://www.inf.fu-berlin.de/~bokowski/pmgjava/index.html`.

can be observed in virtually any Java program that uses generic classes like, for example, `Stack`. Each time an object is obtained from a `Stack`, a type cast is needed before methods specific to that object's type can be called on it. If `Stack` was a class that is parameterized by the type of the objects that can be stored, these casts would not be required.

Assume that we want to define a class `Stack` which can hold instances of a generic type `A`. We would write a class that is parameterized with `A`, so that we can refer to the generic type `A` in the body of the class. Using Pizza's syntax [7], one could image a simple `Stack` class being defined as follows:

```
public class Stack<A> {
  private A[] store = new A[100];
  private int size  = 0;
  public void push(A a) {
    store[size++] = a;
  }
  public A pop() {
    return store[--size];
  }
}
```

To actually use a Stack, an actual type parameter must be provided for `A`. A stack of `String` objects, for example, could be used as follows.

```
public static void main(String args[]) {
  Stack<String> s = new Stack<String>();
  s.push("Hello");
  System.out.println(s.pop().length());  // access to length()
                                         // without casting
}
```

2.2 Compiling Definitions of Parameterized Types

To solve the problem of compiling code that defines a parameterized type, the compiler need not be modified at all. Rather, we propose to allow parameterized types to be defined as normal Java source code following certain conventions. Consider what remains of the above definition of Stack, after its header has been stripped away:

```
private A[] store = new A[100];
private int size  = 0;
public void push(A a) {
  store[size++] = a;
}
public A pop() {
  return store[--size];
}
```

A Java compiler could compile this code without problems if it knew type A. But nothing prevents us from defining an empty interface named A, acting as a placeholder type, so that the compiler will accept the above code:

```
interface A {}
```

Now only a part of the solution for compiling code that defines a parameterized type remains, that is, a naming convention for distinguishing between ordinary types and parameterized types. This is accomplished by including the names of all formal type parameters in the name of a parameterized type. A class or an interface is a parameterized type if its name consists of a base name, which is followed by the names of all formal type parameters, where each such parameter name is enclosed by two '`$$`' characters.[2]

The '`$`' character is allowed to occur in Java identifiers, but its use is discouraged for normal programs, making it available for name mangling schemes. Thus, for example, the resulting name for our parameterized class Stack would be `Stack$$A$$`, and the resulting name for a class `Hashtable` that is parameterized with both a key type K and a value type V would be `Hashtable$$K$$$$V$$`. Note that this syntax, which admittedly is somehow awkward, can be hidden from the programmer by means of a preprocessor, which will be introduced in section 4.4.

To sum up, here is the complete implementation of our simple `Stack$$A$$`, which can be compiled by an unmodified Java compiler:

```
interface A {}

public class Stack$$A$$ {
  private A[] store = new A[100];
  private int size  = 0;
  public void push(A elem) {
    store[size++] = elem;
  }
  public A pop() {
    return store[--size];
  }
}
```

So far, we have described how definitions of parameterized types can be compiled, but we have not explained how these types can actually be used. In the next section, we give a solution for this second problem.

2.3 Compiling Instantiations of Parameterized Types

Assume now that we want to use our `Stack$$A$$` for storing color values, represented by objects of class Color:

[2] This name mangling scheme is compatible with the standard mangling scheme for inner classes.

```
public class Color {
  public byte r, g, b;
  public Color(byte r_, g_, b_) { r=r_; g=g_; b=b_; }
}
```

We would like to write code that looks as follows:

```
Stack$$Color$$ s = new Stack$$Color$$();
s.push(new Color(255,0,0));
s.push(new Color(0,255,0));
System.out.println(s.pop().r);
System.out.println(s.pop().r);
```

When given to an unmodified Java compiler, the first line would obviously result in an error message like "Can't find class `Stack$$Color$$`". Obviously, a definition for `Stack$$Color$$` is not available. However, we could provide such a definition dynamically, if we changed the way in which input files are loaded by the compiler. The idea is to load an instantiation of `Stack$$A$$` derived by replacing all references to "`A`" by references to "`Color`" in the loaded copy of the original file. As we will see in section 3.4, this transformation produces valid byte–code. A very similar technique was already used in [1] for load–time expansion of parameterized types instead of compile–time expansion.

In Sun's Java compiler, a single method is responsible for loading compiled byte–code files of a given name. It can be modified as follows: If the file that is to be loaded has a name that follows our naming convention for parameterized classes and interfaces, appropriate byte–code for an instantiation of a parameterized type is generated. Otherwise, the normal procedure for loading byte–code files is followed.

In the case of `Stack$$A$$`, appropriate byte–code is generated in four steps:

1. The base name "`Stack`" of the type that the compiler tries to load is extracted.
2. An existing byte–code file that has the same name, but different names for the parameters, is searched. In our case, "`Stack$$A$$`" is found. It is now known that the formal parameter "`A`" needs to be replaced by the actual parameter "`Color`".
3. Therefore, the file is loaded into memory, and all references to type "`A`" are replaced by references to type "`Color`". For this purpose, a class library for manipulating byte–code [3] is used. As has been noted in [1], the format of the byte–code file makes it very easy to perform these replacements, as all references to class and interface types are stored independently from the actual byte code instructions in the first part of the file called the "constant pool".
4. Finally, the resulting byte–code is returned to the compiler.

3 What has been accomplished

In this section, we will characterize our proposal in terms of the translation scheme being used (3.1), the interactions between ordinary types, parameterized types, and type parameters (3.2), and the possibilities for constraining genericity (3.3). Also, we will show that our approach is type–safe and that it allows static type checking (3.4).

In order to make the text more readable, we will use Pizza–style syntax in the following sections where appropriate. Section 3.2 presents examples for translating mangled names into Pizza's syntax. In section 4.2, we describe how mangled names can be hidden from the programmer.

3.1 Heterogeneous Translation

When compiling generic source code, the compiler may generate specialized byte code for every instantiation of a parameterized type (heterogeneous translation), or it may generate generic code that works for all instantiations (homogeneous translation). When compiling for the Java Virtual Machine, both approaches have their pros and cons.

A homogeneous translation scheme requires less memory at run–time because all instantiations of a parameterized type use the same byte–code. However, for the current Java Virtual Machine, runtime type checks need to be inserted into code that uses parameterized types, reducing run–time performance, although it can be ensured at compile–time that these checks will always succeed. Most importantly, primitive types (like `boolean` or `int`) cannot be used as actual type parameters, but must be put into appropriate wrapper classes. Other problems are mismatches between parameterized types and arrays (described in [7]), and the inability to refer to individual instances of parameterized types at runtime (e.g., for casts, or `instanceof`).

A heterogeneous translation scheme results in better runtime performance because no runtime checks need to be inserted into the code, at the cost of higher memory requirements. With a heterogeneous translation, it is possible to allow basic types as actual type parameters. Moreover, the semantic problems indicated above can be avoided.

Our solution is based on a heterogeneous translation scheme. However, we do not allow primitive types as actual type parameters, because the required byte–code transformations would be much more complicated.

3.2 Parameterized Types and Type Parameters

When introducing parameterized types into a language, essentially three new kinds of types are introduced, namely parameterized types, such as `Stack<A>`, instantiations of parameterized types, such as `Stack<Color>`, and formal type parameters, such as `A`. To achieve a clean integration into the language, it is desirable that wherever a normal type could appear in the original language, each of these new kinds of types is allowed. Also, all kinds of types should be

usable as actual type parameters. To make things even more complicated, it is also desirable to allow *incomplete* instantiations of parameterized types, such as Hashtable<K, Color>.

For example, interfaces may extend parameterized interfaces, a formal type parameter can be used to declare arrays with that element type, a parameterized class may inherit from an incomplete instantiation of another parameterized class, and instantiations of parameterized types may be used as actual type parameters (nesting).

With our approach, all of the above combinations are allowed, with one exception that is due to the heterogeneous translation scheme: Primitive types such as, e.g., boolean, char or float are not allowed as actual type parameters.

Note that the convention of enclosing each type parameter within two '$$' characters leads to four consecutive '$$' characters between two type parameters, as in Hashtable$$K$$$$V$$. Thus, multiple parameters of a generic type can be distinguished from cases where an actual type parameter is itself a parameterized type. For example, translating our names into valid names of Pizza classes, Hashtable$$Hashtable$$K$$$$V$$$$$$V$$ would result in Hashtable<Hashtable<K, V>, V>, whereas Hashtable$$K$$$$Hashtable$$K$$$$V$$$$ results in Hashtable<K, Hashtable<K, V>>.

It would of course be preferable to use square or angular brackets for type parameters, but unfortunately, the Java Language Specification [5] does allow only one special character ($)in Java identifiers.

3.3 Constrained Genericity

So far, we have required that all formal type parameters are declared as empty interfaces, such that on objects whose type is such a formal type parameter, only methods of class Object may be called. Clearly, it is desirable to *constrain* the possible actual type parameters for a parameterized class, making it possible to call other methods on such objects in a type–safe way. In fact, it is possible to specify such constraints by declaring the placeholder types in other ways than as empty interfaces.

Generally, there are two main approaches for specifying such constraints. One approach is to require that all actual type parameters be a subtype of a given type. This approach is chosen by most of the other proposals. Another approach for specifying constraints is chosen by the MIT proposal, called "where clauses" [6]. Interestingly, our proposal allows both approaches to coexist.

Type Constraints by Subtyping By letting the placeholder type extend other types, type constraints can be specified that require actual type parameters to be subtypes of those extended types. If actual type parameters are required to be classes, the placeholder type can be a class as well. For example, it would make sense that a class EventForwarder<E> required actual type parameters for E to be subclasses of java.awt.Event. In our solution, this constraint would be specified in the placeholder class E that the compiler needs for compiling the generic class:

```
class E extends java.awt.Event {}
```

Note that the body of this class is still required to be empty. As a second example, assume that in the implementation of a class `SortedVector<O>`, actual parameter types for `O` are required to implement an interface `Comparable`. In our approach, this constraint can be expressed by defining the placeholder interface `O` as an interface extending `Comparable`. Again, the placeholder type has an empty body.

```
public interface Comparable {
  boolean lessThan(Object other);
}

interface O extends Comparable {}
```

Type Constraints by Conformance The second approach for specifying constraints does not require actual type parameters to be subtypes of certain predefined types, but only requires that certain methods can be called on objects of that type. In our solution, it is possible to specify these kinds of constraints as well, by including method signatures in the body of the placeholder class. For example, using the MIT proposal, the constraint that an actual parameter type should include a method `void notify()`, without requiring that this type is a subtype of, say, `Notifiable`, would be specified as follows:

```
class Notifier[N] where N { void notify(); } {
  N[] clients;
  int howmany = 0;
  void start() {
    for(int i=0; i < howmany; i++)
      clients[i].notify();
  }
}
```

Using our approach, the same constraint is specified as follows. (Note that this constraint, like in the MIT proposal, does not require that actual type parameters for `N` be subtypes of `N`.)

```
abstract class N { abstract void notify(); }

class Notifier<N> {
  N[] clients;
  int howmany = 0;
  void start() {
    for(int i=0; i < howmany; i++)
      clients[i].notify();
  }
}
```

Interestingly, our approach not only allows both approaches of specifying constraints on type parameters separately, but also the combination of both.

As an aside, note that our proposal implements F–bounded parametric polymorphism; for example, it is possible to specify type constraints involving binary methods by using parameterized interfaces, as already described in [7], and type–checking for such cases is supported:

```
interface C {}

public interface Comparable<C> {
  public boolean lessThan(C other);
}

interface O extends Comparable<O> {}
```

3.4 Type safety

With regard to type safety, one can distinguish between two requirements. The first — essential — requirement is that our byte–code translation scheme does not introduce type errors by itself. The second — desirable — requirement is that type errors caused by the user should be detected at compile–time rather than at run–time. As we will see, our proposal satisfies both requirements.

No Type Errors are Introduced In the case of unconstrained genericity, it is easy to see that by replacing all references to an empty placeholder interface by another class or interface type, no type errors will be introduced, because objects of the placeholder type can only be accessed by methods defined in class `Object`. For constrained genericity, obviously not all replacements are valid, which is why we introduced constrained genericity in the first place. Thus, before actually replacing a formal parameter type by an actual parameter type, it needs to be checked that the replacement is valid. To perform this check, we first need to load both the byte–code for the actual parameter type, and the byte–code for the formal parameter type (the placeholder type). If the body of the placeholder type is empty, the constraint is of the form that actual types are required to be subtypes of a given type, and we only need to check that the actual parameter type is a subtype of all the types that the placeholder type is declared to be a subtype of. If the body of the placeholder type is not empty, we additionally have to check if for all methods and constructors of the placeholder type, methods and constructors of equal names and signatures exist in the actual parameter type or in any of its supertypes.

Static Type Checking is Retained Our proposal leads to the compiler performing type–checking both on a parameterized class itself and on all users of instantiations of parameterized classes. Thus, type errors will be detected as early as possible. Note that, unlike some implementations for templates in C++, we

do not perform macro expansion but transformation of already compiled and type–checked code. This allows us to report type errors within definitions of parameterized types independently of the uses of such types, whereas the C++ template mechanism does not allow constrained genericity and may cause link–time errors.

4 Improvements

Nothing comes for free, and if something is much cheaper than expected, there is always a catch. As might be expected, our proposal as it was described so far is no exception to this rule. It has a number of drawbacks, which is why we call it "Poor Man's Genericity". But as it turns out, these drawbacks can be overcome by further changes to the compiler, all of which are simple and local changes compared to the implementation requirements of other proposals.

4.1 Separate Compilation Required

Our solution requires that, before instantiations of parameterized types can be used, a byte–code file be available for loading in order to perform the necessary byte–code transformations. This is not a fundamental problem, though, since we know what byte–code we are looking for. All that is needed to overcome this problem is to be able to cause the compiler to compile another Java source file before we return the byte code it was asking for in the first place. We plan to implement the required functionality in the future.

4.2 Better syntax

A more severe problem of our approach is that its syntax is not very simple and easy to understand because we need to mangle information about the parameters inside standard Java identifiers. However, providing a better syntax (with angular or square brackets enclosing type parameters) would require only local changes to an existing compiler's scanner and parser, because our mangling scheme could still be used internally. In fact, we implemented a simple preprocessor that generates mangled names for parameterized classes. Similar to the hook that was needed for intercepting the loading of byte–code files, we added another hook in Sun's Java compiler for intercepting the loading of source files.

Another improvement would be to allow type constraints to be specified at the formal type parameter itself instead of in a separate placeholder type. These placeholder types could then be generated automatically, and they could be hidden from the programmer.

4.3 Fully qualified names

The most annoying problem we had with our simple solution is that all names of formal parameters and actual parameters had to be fully qualified. Even more

unpleasant was the fact that we need another mangling convention for qualified names as type parameters. This mangling translates the dots of qualified names to underscore characters, while 'real' underscore characters need to be escaped. Fortunately, this problem was not very difficult to solve. As will be seen shortly, we have extended the preprocessor to expand unqualified type names inside mangled names.

All our examples so far were based on interfaces and classes belonging to the global, unnamed package. A more realistic example of a parameterized `Stack` class would reside in a named package, say `collections`. Then, the complete mangled name for the parameterized stack class is `Stack$$collections_A$$`, and a stack of strings is called `collections.Stack$$java_lang_String$$`: [3]

```
package collections;

interface A {}

public class Stack$$collections_A$$ {
  private A[] store = new A[100];
  private int size  = 0;
  public push(A a) {
    store[size++] = a;
  }
  public A pop() {
    return store[--size];
  }
}
```

This parameterized stack class can be used as follows:

```
package tests;
import collections.*;

public class Main {
  public static void main(String[] args) {
    Stack$$java_lang_String$$ s;
    s = new Stack$$java_lang_String$$();
    s.push("Hello");
    s.push("world");
    System.out.println(s.pop().length());
    System.out.println(s.pop());
  }
}
```

[3] This is what the compiler sees. Using the preprocessor, the user may just write `Stack<A>` and `Stack<String>`, respectively. All names will be fully qualified and mangled automatically.

Again, this problem can be solved easily by subjecting formal and actual type parameter names to the same name resolution procedure as ordinary type names. This could be achieved by a local change in the compiler if, during name analysis, all formal and actual type parameter names were replaced by their fully qualified and mangled version. In our implementation, we chose to leave the compiler unchanged and instead extended the preprocessor such that it is aware of `package` and `import` statements and fully qualifies all type parameters before mangling.

4.4 Implementation Overview

In this section we will give a brief overview of our implementation.

The following figure illustrates the compilation process for code that uses a parameterized class. Symbols below the dotted line belong to the PMG system, while the other symbols depict plain files and the unmodified Java compiler, i.e. the user's view.

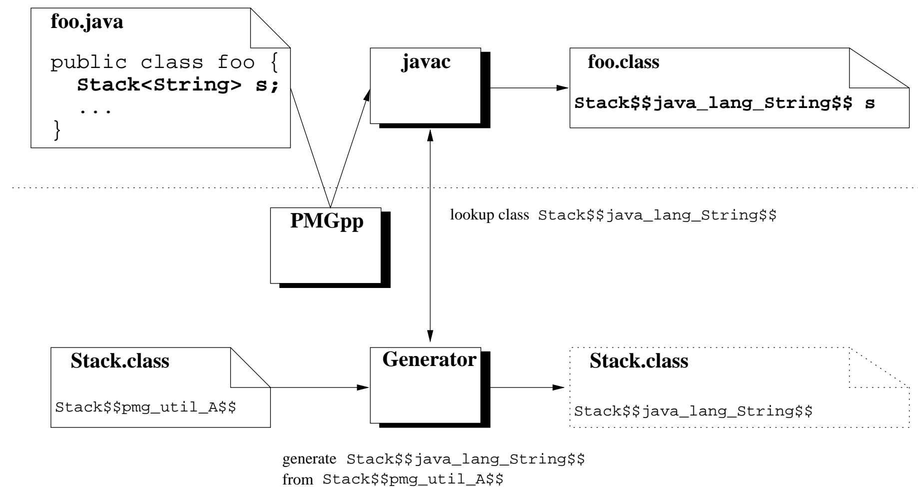

Fig. 1. Overview of the compilation process

The source file `foo.java` written in the Pizza syntax declares a variable `s` of type `Stack<String>`. In the first step, the file is parsed by a preprocessor called `PMGpp`.

It scans the file for parameterized identifiers, fully qualifies all names of actual type parameters and translates them into the name mangling scheme (mangling `.` to `_` characters). Thus, the Java compiler will see `Stack$$java_lang_String$$` instead of `Stack<String>`. It then tries to load this class. At this point, we intercept the compiler in order to use our own loading algorithm implemented

in a class named `Generator`. Since the requested class does not exist, a class with a substitutable placeholder type is searched that can be used as a template.

Given that a file "`Stack.class`" containing the class `Stack$$pmg_util_A$$` exists somewhere on the `CLASSPATH`, we load this class file into memory. Consequently, we need to ensure that the substitution is valid, i.e. that `String` can be safely substituted for `A`. This procedure was described in more detail in section 3.4. If all type checks succeed, we replace all references to the formal type `A` with references to the actual type `String`. This yields the required class `Stack$$java_lang_String$$`. This file is not created physically and thus is drawn with dotted lines. The newly created class is returned to the compiler which can now proceed and generate "`foo.class`".

We chose not to store generated files in permanent storage in order to hide these from the user, to reduce storage and transmission size of the code, and to avoid problems with inconsistent instantiations of generic types. Therefore, at load–time, the same mechanism for generating appropriate byte–code needs to be employed. In our implementation, we use a custom class loader that calls the appropriate method of class `Generator` for all classes.

For cases in which a custom class loader cannot be used, generation of actual files in permanent storage can be turned on with a compiler option that specifies a directory in which generated files are written.

5 Other proposals

In this section we briefly describe other proposals and summarize their properties in figure 2.

5.1 Pizza

Pizza [7] proposes both a homogeneous and a heterogeneous translation, but currently only the homogeneous translation is implemented by a compiler. Therefore, generic code is not duplicated for each instantiation; instead, runtime casts (that will always succeed) are inserted into the code that uses parameterized types because they are required by the byte–code verifier. However, there are some problems with arrays, casts, and the `instanceof` operator, and, as is described in [1], there are undesirable implications on Java's security model. Note that these drawbacks only apply to the homogeneous translation scheme. Pizza automatically generates wrapping code in order to allow basic types as actual (unconstrained) type parameters in the case of homogeneous translation. Constraints on formal type parameters are expressed by sub–typing. Pizza not only implements parameterized types, but adds other advanced language features such as first–class functions and algebraic classes with pattern matching to Java. Moreover, a formal type system is defined that ensures type safety of Pizza. The freely available Pizza compiler is a complete implementation that already has been proven useful in practice.

5.2 GenericJava (GJ)

A successor to Pizza, GJ [2] implements parameterized types and parameterized methods. It allows only non–primitive types as actual type parameters. Again, a homogeneous translation scheme is used, leading to the same problems with the use of parameterized types and type parameters in array allocations, and with the `instanceof` and cast operators. Some additional changes have been made to Pizza in order to allow compatibility between new code that uses parameterized classes and existing code. Not only can parameterized classes be used in a conventional way, where formal type parameters are replaced by their static type bounds, but also conventional code can be parameterized after it has been compiled, by providing parameterized signatures for the methods of a class. In this manner, the Java collection classes have been parameterized without the need to access their source code. A full compiler for GJ, and a tool that enables parameterization of existing code has been implemented.

5.3 MIT proposal

The MIT Proposal [6] uses a homogeneous translation scheme, avoiding code duplication. It suggests changes to the Java Virtual Machine in order to overcome the problem of unneeded runtime overhead by casts that are normally required for a homogeneous translation. Moreover, each instantiation of a parameterized type has a runtime representation, thus avoiding the semantic problems that Pizza has. Unfortunately, this makes generic code incompatible with normal Java Virtual Machines. Basic types are allowed as type parameters and can be accessed like other objects by defining a fixed set of methods for basic types, derived from operators for that types. Constraints on formal parameter types are specified using where–clauses, a new language construct that lists methods that need to be supported by actual parameter types. A compiler is not implemented, but a modified Java Virtual Machine was built to assess the performance gained by not requiring run–time casts. Note that although our proposal allows type constraints very similar to where–clauses, we do not need to change the Java Virtual Machine because we employ a heterogeneous translation scheme.

5.4 Stanford proposal

The proposal by Sun and Stanford University [1] employs a heterogeneous translation scheme. In order to avoid duplicated code being stored in files or transmitted over the network, generic code is compiled to a special byte–code format that is expanded at load time, using a technique very similar to ours. We apply this technique not only to the loading of classes into the Java Virtual Machine, but also to the process of compiling generic code. Basic types are not allowed as type parameters. Constraints can be specified by sub–typing. There is an implementation of the class loader that expands instantiations of parameterized types, but no implementation of a compiler.

5.5 Genja

For Genja [4], a heterogeneous translation is chosen (a homogeneous translation scheme is discussed as well); memory requirements or performance issues are not discussed in the proposal. Genja has a unique way of supporting basic types as actual type parameters, as these are accessed in a parameterized class by means of method overloading. Instead of supporting constraints for actual type parameters, Genja includes generic method parameters, a new construct that allows operations required for formal parameter types to be supplied when instantiating a parameterized class. A compiler that performs source code transformations to Java has been developed.

5.6 Virtual types

Virtual Types [8] support parameterized classes in a way that is different from all other proposals, since no distinction is made between a parameterized type and an instantiation of a parameterized type. Instead, a type `A` may contain "virtual types" that may be restricted in subtypes of `A`, leading to a covariant typing scheme. Because covariant typing may lead to type errors, runtime checks involving a virtual method call are inserted into code with virtual types. Unlike other proposals, not all of these checks can be guaranteed to succeed at runtime. Basic types are not allowed as types of those type variables. Currently, no compiler for Virtual Types for Java is implemented.

6 Conclusions

We have presented a new proposal for adding parameterized types to Java, focusing on the implementation of such a proposal. We have shown that, by changing only one method in an existing Java compiler, parameterized classes can be supported. Interestingly, our proposal compares quite well to other proposals. The resulting genericity has some minor drawbacks, all of which can be ameliorated by simple additional changes. Each of these changes can be localized to a specific part of a Java compiler, making it unnecessary to develop new compilers or to change existing compilers extensively in order to support parameterized types for Java.

As has been pointed out recently [2], the heterogeneous translation scheme leads to problems when a parameterized type is instantiated using an actual type parameter which is not `public` and belongs to a different package. In this case, the Java visibility rules do not allow the instantiation to refer to the actual type parameter. We are currently investigating this issue. One solution would be to automatically change the visibility of actual type parameters in these cases. However, we are not satisfied with this solution since it would lead to security problems similar to those exhibited by the homogeneous translation scheme.

To validate our implementation, we have successfully parameterized and compiled the new collection classes of Sun's Java Development Kit 1.2. As claimed

	Pizza	**GJ**	**PolyJ (MIT)**	**Stanford**	**Genja**	**Virtual Types**	**Poor Man's Genericity**
translation	homogeneous	homogeneous	hybrid	heterogenous	heterogeneous	inheritance	heterogeneous
basic types as type parameters	yes, automatic wrapping, unconstrained	no	accessed as if they were objects	no	yes, accessed by method overloading	no	no
duplicated code	no	no	little	at run-time	yes	no	at compile-time and at run-time
runtime casts	yes	yes	yes	no	no	using method call	no
type constraints	subtyping	subtyping	where-clauses	subtyping	generic method parameters	subtyping	subtyping and where-clauses
new compiler / type system	yes	yes	yes	yes	yes	yes	local changes to existing compiler
additional new language constructs	first-class functions, algebraic classes, pattern matching	parameterize d methods	where clauses	no	generic method parameters	virtual types	no
implemented	yes	yes	yes	Class Loader	no (?)	no	yes
disadvantages	problems with arrays, casts, and instanceof; security ?	problems with arrays, casts, and instanceof; security ?		extension of the byte-code format		not statically type-safe due to covariant typing	packages; explicit placeholder types separate compil. error handling

Fig. 2. Overview of proposals

by the designers of these classes, this mostly amounted to adding type parameters to all classes and removing unnecessary runtime casts. Interestingly, for two (inner) implementation classes this did not work, because the implementation included code that encoded the type of stored objects in a variable, making it impossible to statically prove its type–correctness. However, it was easy to change this code, not only allowing full compile–time type checking, but also resulting in a cleaner design. We successfully use several of these classes (ArrayList<A>, HashMap<K, V>, HashSet<A>), and corresponding interfaces, super–classes, and iterator classes in a Java program consisting of over 100 classes.

References

1. O. Agesen, S. N. Freund, and J. C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings OOPSLA'97*, Atlanta, GA, 1997.
2. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past : Adding Genericity to the Java Programming language. , 1998.
3. M. Dahm. *The JavaClass API*. http://www.inf.fu-berlin.de/~dahm/JavaClass/ , 1998.
4. M. Evered, J. L. Keedy, G. Menger, and A. Schmolitzky. Genja – A New Proposal for Parameterised Types in Java. In *Proceedings TOOLS Pacific*, Melbourne, Australia, 1997.
5. J. Gosling and G. Steele B. Joy. *The Java Language Specification*. Addison-Wesley, 1996.
6. A.C. Myers, J. A. Bank, and B. Liskov. Parameterized Types for Java. In *Proceedings 24th ACM Symposium on Principles of Programming Languages*, Paris, France, 1997.
7. M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. In *Proceedings 24th ACM Symposium on Principles of Programming Languages*, Paris, France, 1997.
8. K. K. Thorup. Genericity in Java with Virtual Types. In *Proceedings ECOOP'97*. LNCS 1241, Springer Verlag, 1997.