Two Ways to Bake Your Pizza — Translating Parameterised Types into Java

Martin Odersky[1], Enno Runne[2], and Philip Wadler[3]

[1] Ecole Polytechnique Fèdèrale de Lausanne[†]
`Martin.Odersky@epfl.ch`
[2] University of Karlsruhe[‡]
`enno.runne@ira.uka.de`
[3] Lucent Technologies
`wadler@research.bell-labs.com`

Abstract. We have identified in previous work two translations for parametrically typed extensions of Java. The homogeneous translation maps type variables to a uniform representation, while the heterogeneous translation expands the program by specialising parameterised classes according to their arguments. This paper describes both translations in detail, compares their time and space requirements and discusses how each affects the Java security model.

1 Introduction

Java does not provide parameterised types, but is designed to make them relatively easy to simulate. Consider `java.util.Hashtable`, the standard library class whose declaration is shown in Figure 1. The `Hashtable` design uses `Object` as the type of both keys and values. Every reference type is a subtype of `Object`, and so may be used as a key or value. It is up to the user to recall what types have been used, and perform appropriate casts. For instance, the `HashtableClient` class uses a hashtable with keys of type `String` and values of type `Class`. Lookups in the hashtable return an `Object` value that must be cast to a `Class`.

Idioms like this are widespread in the Java library, including simple utilities like stacks and vectors, the event model introduced for Java 1.1 (where events may return values of arbitrary type), the JavaBeans library (which depends on events), and the collection classes planned for Java 1.2 (which provides generic collections, sets, lists, and hashtables).

In Pizza [OW97a], we have extended Java with features common in functional programming languages. The most important extension in Pizza is its type system, which supports parameterised types. Figure 2 shows how hashtables are defined in Pizza. Here, `Key` and `Data` are type parameters for the `Hashtable` class. Access code for parameterised hash tables no longer needs a type cast to recover the types of the retrieved elements.

[†] This work was completed while at the University of South Australia.
[‡] This work was completed while at the University of South Australia.

```
public class Hashtable {
    public Hashtable () { ... }
    public Object get(Object key) { ... }
    public Object put(Object key, Object value) { ... }
}

class HashtableClient {
    private Hashtable loadedClasses = new Hashtable();
    public Class loadClass(String name) {
        Class c = (Class)loadedClasses.get(name);
        if (c == null) {
            c = Class.forName(name);
            loadedClasses.put(name, c);
        }
        return c;
    }
}
```

Fig. 1. Hashtables in Java.

There have been also several other proposals to extend Java with parameterised types [MBL97,AFM97,Bru97], with roughly similar capabilities as the Pizza approach, as well as proposals to extend Java with virtual types [Tho97], [Tor97], with capabilities quite different to it [OW97b]. Pizza differs from these other proposals in that it admits basic types as type parameters and in that it has polymorphic methods in addition to parameterised types.

Generally, there is widespread agreement that a generic type system with its benefits for the construction of reliable software systems is a useful addition to Java. It is less clear how such a type system should be implemented in the confines of the Java Virtual Machine. In earlier work [OW97a] we outlined two translations from Pizza's generic types to Java. In this paper we present the two translations in detail, discuss their security aspects, and compare their resource consumption empirically.

The *homogeneous* translation maps a parameterised class into a single class that represents all its instantiations. For example, the homogeneous translation of the Pizza code for parametric hashtables in Figure 2 yields essentially the Java code in Figure 1. The homogeneous translation is similar to schemes for implementing parametric polymorphism in ML [Ler90,SA95]. The homogeneous translation underlies the current Pizza compiler whose use has become widespread since December 1996. This translation yields compact code that can run in standard Java-enabled browsers.

By contrast, the *heterogeneous* translation maps a parameterised class into a separate class for each instantiation. For example, the heterogeneous translation of the Pizza class `Hashtable<Key,Value>` replaces the instance `Hashtable<String,Class>` by the class `Hashtable$_String_$_Class_$`, whose

```
public class Hashtable<Key, Data extends Object> {
    public Hashtable() { ... }
    public Data get(Key key) { ... }
    public Data put(Key key, Data value) { ... }
}

class HashtableClient {
    Hashtable<String,Class> loadedClasses = new Hashtable();
    Class loadClass(String name) {
        Class c = loadedClasses.get(name);
        if (c == null) {
            c = Class.forName(name);
            loadedClasses.put(name, c);
        }
        return c;
    }
}
```

Fig. 2. Hashtables in Pizza.

body is generated by replacing each occurrence of `Key` by `String` and each occurrence of `Value` by `Class`. The heterogeneous translation resembles the translation scheme employed in the TIL compiler for Standard ML [HM95], the generic instantiation in Ada [oD80] and Modula-3 [CDG$^+$88], and the template expansion process in C++ [Str86]. However, unlike in C++, all type checking in Pizza is performed before expansion, not afterwards.

The heterogeneous translation poses the problem of keeping track of the different instances of generic types that are produced at compile-time or link-time. The Java environment admits an elegant solution to this problem: One can make use of Java's dynamic class loading capabilities to generate instances on demand at run-time. Agesen, Freund and Mitchell introduce a scheme based on this idea [AFM97]. They argue that the heterogeneous translation supported by their scheme offers several benefits:

First, since the heterogeneous translation preserves more type information than the homogeneous translation, it tends to give more freedom to the language designer. Language constructs such as checked type casts to parameterised types, generic array creation, or mixins [FKF98] require an explicit representation of type variables at run time and therefore are only feasible under a heterogeneous translation. (Consequently, since Pizza was designed to support both translations, these constructs are missing in Pizza.)

Second, the heterogeneous translation may lead to better run-time performance, trading off speed for code size. By specialising a generic type with its arguments, one can eliminate all type casts and boxing/unboxings introduced by the homogeneous translation. Furthermore, the specialisations might provide additional opportunities for subsequent optimisations.

While these arguments are plausible, they have not yet been verified empirically. We therefore extended the Pizza compiler to allow the heterogeneous translation as well as the existing homogeneous translation. Moreover the new translation allows heterogeneous and homogeneous code to be mixed, which is necessary to support Pizza's generic methods which are still translated homogeneously. We then compared both translations using a range of benchmarks, including the Pizza compiler itself. The results can be summarised as follows.

– As expected, the heterogeneous translation leads to somewhat faster code for member access in generic types, and to much faster code for generic array access. Except for arrays, the performance gains are modest, however. They do not generally exceed 5%. Furthermore, code involving polymorphic methods becomes much slower since the interface between homogeneous and heterogeneous code is complex.
– The increase in code size incurred by the heterogeneous translation can be substantial. For example, the footprint of the Pizza compiler increases by about 60%.
– With Sun's current JDK, the additional classes generated by the heterogeneous translation incur a significant class loading overhead, often large enough to wipe out the performance gains obtained by specialisation.
– In the course of our experiments we also discovered a severe security problem incurred by the heterogeneous translation. Unless the access checking in the Java Virtual Machine can be refined, this problem can be avoided only by restricting the source language to a point where the usefulness of parameterisation is very much in doubt.

The rest of this paper is organised as follows. Section 2 describes the homogeneous translation from Pizza to Java, and Section 3 describes its heterogeneous counterpart. Section 4 compares the two translations in terms of their security aspects. Section 5 compares them empirically in terms of their run-time and space consumption using a number of benchmark programs. Section 6 concludes.

2 The Homogeneous Translation

The homogeneous translation from Pizza to Java proceeds in three steps, which affect types, expressions, and declarations. First, Pizza types are mapped to Java types by erasing all type parameters, and by mapping all type variables to their upper bound. Then, type conversions are inserted in expressions where needed to avoid type compatibility errors. The effect of the first two steps is illustrated by comparing the Pizza program in Figure 2 with its translation in Figure 1.

The first step, type erasure, is defined recursively as follows:

1. The erasure of a parameterised type $C\langle T_1, ..., T_n\rangle$ is its class or interface name, C.
2. The erasure of an array type $A[\,]\ldots[\,]$ whose element type A is a type variable is the abstract class `pizza.support.array`.

```
abstract public class array {
    public abstract int length();
    public abstract Object at(int i);
    public abstract void at(int i, Object x);
    ...
}

public class booleanArray extends array {
    private boolean[] elems;
    public int length() { return elems.length; }
    public Object at(int i) { return new Boolean(elems[i]); }
    public void at(int i, Object x) {
       elems[i] = ((Boolean)x).booleanValue();
    }
    ...
}
```

Fig. 3. Array classes.

3. The erasure of every other type variable is the erasure of the type variable's bound, or `Object` if no bound was given.
4. The erasure of every other type is the type itself.

Arrays are treated specially in the second rule above. Without this rule, the array type `A[]` with type variable `A` would be mapped to `Object[]`. The problem is that `Object[]` is not convertible to arrays of primitive types except by copying the whole array, which loses sharing. Instead, generic arrays are mapped to the abstract class `pizza.support.array`, from whose definition we excerpt in Figure 3. This class provides a generic interface to an array. It has a method to return the length of the array as well as *getter* and *setter* methods for accessing individual elements. The translation maps accesses to the `length` field of a generic array to calls of the `length()` method, and it maps indexed accesses of generic arrays to calls of the getter and setter methods.

There are nine subclasses of `pizza.support.array`, one for each of Java's eight basic types, plus one for type `Object` which is the generic representation of all arrays of reference type. Each subclass implements the abstract methods in class `pizza.support.array` via a private array of the corresponding element type. As an example, Figure 3 gives the subclass for arrays of booleans.

The Pizza translation of expressions inserts type conversions as needed to make the resulting Java code legal. Type conversions between reference types are simply type casts. The Pizza type system guarantees that these type casts will always succeed at run-time, but they are still necessary to let the generated classes pass the byte-code verifier, which checks programs according to Java's typing rules. Type conversions between basic types and reference types work with Java's mirror classes for basic types. For instance, a `boolean B` is converted to an `Object` with `new Boolean(B)` and the reverse conversion from an `Object` `O` is `((Boolean)O).booleanValue()`.

Type conversions between basic arrays and the generic array class involve a wrapper class for the basic array. For instance, a `boolean[]` array `BS` is converted to `pizza.support.array` with `new booleanArray(BS)` and the reverse conversion from a generic array `A` can be achieved with `((booleanArray)A).elems`.

The last step of the translation deals with the problem that type erasure can destroy overriding and implementation relationships between Pizza methods. As an example, consider a parameterised interface `Iterator<A>` and a class that implements iterators of element type `String`:

```
interface Iterator<A> {
   A next();
   void append(A x);
}
class StringArrayIterator implements Iterator<String> {
   String[] a;
   public String next() { ... }
   public void append(String x) { ... }
}
```

The translation after erasing types is:

```
interface Iterator {
   Object next();
   void append(Object x);
}
class StringArrayIterator implements Iterator {
   String[] a;
   public String next() { ... }
   public void append(String x); { ... }
}
```

This translation has two problems. First, the implementation of method `next` in class `StringArrayIterator` has a different return type than its definition in interface `Iterator`, which violates a requirement of the Java language [GJS96, Sec. 8.4.6.3]. Second, the implementation relationship between the two append methods in the original Pizza source does not translate to a corresponding relationship in their translations: After translation, `append` in class `StringArrayIterator` takes a `String` as argument, and hence does not implement `append` in interface `Iterator`, which takes an `Object` as argument.

To correct these shortcomings, the translation inserts *bridge methods* to restore overriding relationships. Bridge methods have the type after translation of the implemented or overridden method. Their body simply forwards the call to the true implementing method, converting argument types as needed.

To avoid name clashes between bridge methods and other methods, and to avoid the problem with different return types in implementations, all methods whose type refers to a type parameter are coded with the name of their enclosing class. For instance, our iterator example would be translated as follows.

```
abstract class Iterator {
   Object Iterator$next();
   void Iterator$append(Object x);
}
class StringArrayIterator implements Iterator {
   String next() { ... }
   void append(String x); { ... }
   final Object Iterator$next() { return next(); }
   final void Iterator$append(Object x) { return append((String)x); }
}
```

The main strengths of the homogeneous translation are its simplicity and the compact size of generated code. Its main disadvantage is the run-time overhead introduced by bridge methods and type conversions. Type conversions from basic types or basic array types to their generic representations are particularly expensive since they involve the creation of a wrapper object. These costs are eliminated in the heterogeneous translation, which we discuss presently.

3 The Heterogeneous Translation

The heterogeneous translation expands the program by specialising generic classes according to their arguments, so that types become monomorphic. This expansion is performed on demand at load time, using a modified class loader which produces instance classes from class templates. The translation and expansion process resembles the one described by Agesen, Freund and Mitchell [AFM97], but there are two complications.

First, parameters in Pizza can be base types as well as reference types. Therefore, our class loader must be able to instantiate a parameterised class template with a base type, which generally requires changes to the byte codes. In Agesen *et al.*'s scheme, which only considers reference type parameters, a class file's constant pool is all that needs to be changed.

Second, Pizza has polymorphic methods, which are unaffected by the class loader expansion. Consider for instance the `zip` method in Pizza's `List` class which joins two lists into a list of pairs.

```
public class List<A> {
  public <B> Pair<A,B> zip(List<B> xs) { ... }
  ...
}
```

A specialisation of `List`'s parameter `A` to, say, `String` would leave the following residual function, which is still polymorphic:

```
  public <B> Pair<String,B> zip(List<B> xs) { ... }
```

Now, one might consider expanding such polymorphic functions as well as expanding parameterised types. This is difficult, however, because at the point where we load a class, we may not yet know the possible instances of its polymorphic methods. Since we can't change the code of a class after the class is

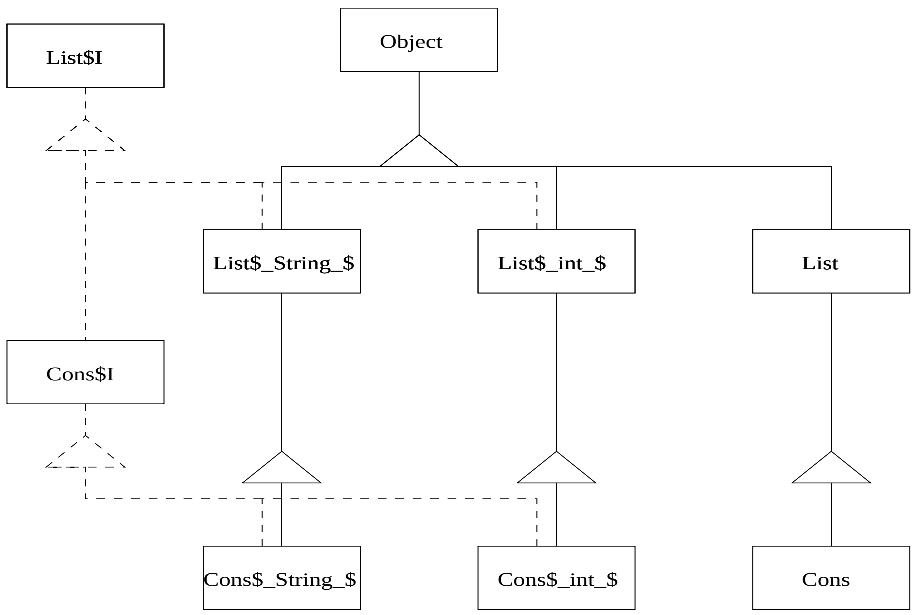

Fig. 4. Classes generated under the heterogeneous translation.

loaded, we can specialise generic methods only by placing each in a compiler-generated inner class of its own, which can then be specialised at each call. This approach would incur a heavy overhead at both compile-time and run-time.

Instead, our scheme leaves polymorphic methods as they are, using the homogeneous translation for all type variables which are not parameters. As a consequence, an object might now be accessed at the same time from code that is specialised and from code that is not. To allow this, we provide two *views* for every parameterised type. The homogeneous view maps the type's parameters to a uniform representation, namely the erasures of their bounds. The heterogeneous view is simply the specialised version of the class. The homogeneous view takes the form of a Java interface which all specialisations implement. This dual view would also provide the opportunity for more refined translation schemes, which mix homogeneous and heterogeneous translations in order to obtain a good balance between speed and code size.

We now describe the translation in more detail. For each parameterised class `C<A>`, three Java classes are generated.

– An *interface* `C$$I` which represents the homogeneous interface to the class,
– A *template class* `C$_$0_$` which the class loader will instantiate to obtain specialisations as needed,

```
public class List<A> {
    public A head;
    public List<A> tail;

    public List<A> append(A last) {
        return new List(head, tail == null
                              ? new List(last, null)
                              : tail.append(last));
    }

    public static <A> List<A> fromArray(A[] a) {
        List<A> xs = null;
        for (int i = a.length - 1; i >= 0; i--)
            xs = new List(a[i], xs);
        return xs;
    }

    public List(A x, List<A> xs) {
        head = x; tail = xs;
    }
}
```

Fig. 5. Generic list class.

– A *static base class* `C` which implements all static methods of `C`, and also provides methods to construct new instances of `C`.

If parameterised classes inherit from other parameterised classes, the inheritance relationship is translated to an inheritance relationship between corresponding components. For instance, Figure 4 shows the case of a parameterised class `Cons<A>` that extends `List<A>`. This figure shows the static base classes `List` and `Cons`, instance classes with `int` and `String` as the parameter type, and the homogeneous interfaces `List$$I` and `Cons$$I`. Class inheritance is expressed by solid lines and interface implementation is expressed by dashed lines.

To illustrate the roles of these classes and interfaces, consider the generic list class of Figure 5. Lists have `head` and `tail` fields. There is a method `append` that returns a new list with the given element appended to the elements of the current list. There is also a static polymorphic method `fromArray` which constructs a list from the elements of an array.

Homogeneous Interfaces

The homogeneous view of this class will map every occurrence of the type variable `A` to `A`'s upper bound, in this case `Object`. The view is represented by a Java interface, `List$$I`, given in Figure 6.

```
interface List$$I {
   List$$I List_append(Object x$0);
   List$$I List_tail();
   List$$I List_tail(List$$I x$0);
   Object List_head();
   Object List_head(Object x$0);
}
```

Fig. 6. Homogeneous interface for `List`.

The homogeneous interface contains one access method for every instance method whose type refers to a parameter of the class. Generic instance fields are represented by getter and setter methods in the interface.

Templates and Specialisations

A specialisation of class List with a concrete element type `X` is obtained simply by replacing the type parameter `A` with `X`. Instead of `List<X>`, which is not a legal class name in Java, we use `List$_X_$`.

In addition to all instance members of the original class, specialisations also contain implementations of all methods in the homogeneous interface. These implementations simply call the original method or load/store the original instance field, wrapping and unwrapping objects as required.

Specialisations are synthesised at run-time from a template class. Template classes are very similar to their instantiations, except that they use *place-holders* instead of concrete parameter types. Place-holders are named `$0`, `$1`, and so on, up to the number of type parameters of a class. As an example, Figure 7 presents the template class for lists.

Polymorphic Methods

Polymorphic methods are translated using a modified version of the homogeneous translation described in the last section. The erasure of a parameterised class is now its homogeneous interface. Accesses to fields and methods of such a class are translated to corresponding accesses in the homogeneous interface. Another change is that the actual instance type of a polymorphic method is made available at run time by passing implicit type parameters as additional parameters of type `Class`. For instance the method signature

```
<A> List<A> fromArray(A[] a)
```

becomes

```
List<Object> fromArray(Class A, pizza.support.array a)
```

```
public class List$_$0_$ implements List$$I {
   public $0 head;
   public List$_$0_$ tail;

   public List$_$0_$ append($0 last) {
      return new List$_$0_$(head, tail == null
                                  ? new List$_$0_$(last, null)
                                  : tail.append(last));
   }

   public List$_$0_$($0 x, List$_$0_$ xs) {
      super(); head = x; tail = xs;
   }

   /* Implementations of interface functions for homogenous access */

   public Object List_head() { return (Object)head; }
   public Object List_head(Object x$0) {
      head = ($0)x$0; return x$0;
   }
   public List$$I List_tail() { return tail; }
   public List$$I List_tail(List$$I x$0) {
      tail = (List$_$0_$)x$0; return x$0;
   }
   public List$$I List_append(Object x$0) { return append(($0)x$0); }
}
```

Fig. 7. Template class for `List`.

We have not yet explained how instances of a parameterised type are created from within a polymorphic method. The matter is trivial whenever the parameters are known types or type parameters of the enclosing class. In this case we simply invoke the corresponding constructor of the instance class (or its template). But if some parameters are local type variables of a polymorphic method, the correct instance is known only at run time, when the actual argument type is passed. In these situations, we have to resort to Java's reflection library for object construction. For example, the object allocation in method `fromArray` would involve the following three steps.

```
// Create class of object to be allocated:
Class list_A = Class.forName("LinkedList$_" + A.getName() + "_$);

// Obtain constructor given argument types:
Constructor constr = list_A.getConstructor(new Class[]{A, list_A});

// Invoke constructor:
return (LinkedList$$I) constr.newInstance(new Object[]{x, xs});
```

To make this scheme reasonably efficient, both class and constructor objects are cached in hashtables which are indexed by the argument type(s). There will be one such hashtable for every constructor of a parameterised class. Using caching, each constructor method will be looked up only once through the reflection interface. Caching was essential in our implementation since method and constructor lookup was very slow. With caching, we observed a slowdown of between 2 and 3 for programs that used polymorphic static methods exclusively, as compared to the same programs using instance methods. Without caching, the slowdown was about a factor of 500.

The Specialising Class Loader

The heterogeneous translation uses a customised class loader for expanding template classes with their actual type parameters. The class loader inherits from class `java.lang.ClassLoader`, overriding method `loadClass(String name, boolean resolve)`. Our new implementation of `loadClass` scans the given name for embedded type parameter sections, delimited by `$_` and `_$` brackets. If a parameter section is found, a template for the class constructor is loaded and expanded with the given parameter(s). Once loaded, template classes are kept in a cache for subsequent instantiations.

A classfile consists in essence of a *constant pool* which holds all constant information such as classnames, method descriptors and field types, a *fields* section and a *methods* section [LY96]. The code for all methods and for variable initialisers is stored in *code attributes* attached to the methods or the class.

Expansion of a classfile consists of two steps. The first step involves replacing in the constant pool all occurrences of a place holder name with the name of the corresponding actual type parameter. Quite surprisingly, this is all that is needed for expanding classes with reference type parameters.

If type parameters are basic types, a second step is required to adapt code blocks to the actual parameters. The Java Virtual Machine uses different bytecode instructions for accessing data of different type groups. For instance, a local variable of type `int` is accessed with the instruction `iload` whereas `aload` is used if the variable is of reference type. To help the class loader find all instructions that need to be adapted, the compiler generates a `Generic` attribute that gives the offsets of these instructions relative to the start of the code block.

4 Security Implications

The homogeneous and heterogeneous translations have quite different security implications. Under the homogeneous translation, a Pizza program is no more secure than the generated Java code after type erasure. Since type parameters are erased by the translation, we can not always guarantee that a parameterised type is used at run-time with the type parameter indicated in the source. As Agesen *et al.* [AFM97] argue, this constitutes a security risk as users might expect run-time type-checking to extend to fully parameterised types.

The heterogeneous translation does not have this problem, since all type information is maintained at run-time. Rather to our surprise, the heterogeneous translation nevertheless fits poorly with the security model of Java. We encountered two difficulties, one with visibility of instance methods, and one with visibility of type parameters.

Visibility of Instance Methods

Because the Pizza implementation mixes homogeneous and heterogeneous translations, all the heterogeneous instantiations of a class must be accessible via a common interface. For instance, in Section 3 there is a list interface (Figure 6) that is implemented by the list template (Figure 7). In Java, all methods implementing an interface are required to be public, and hence the Pizza translation works only when all instance methods are public.

If the template class has no superclass (other than `Object`), we could solve this problem by replacing the interface with an abstract class, since abstract classes do allow non-public members. Unfortunately, the template class may already have a superclass that depends on the type parameter, so the multiple inheritance provided by interfaces is essential.

If one were allowed to change the Java security model, the problem could be fixed by generalising interfaces to allow non-public methods. Since the JVM is currently being implemented in silicon, such a change seems unlikely.

Thus, the combination of homogeneous and heterogeneous translations of Pizza requires all instance methods of a parameterised class to be public, which severely restricts the utility of the Java visibility model. This problem would not arise if one adopted a pure heterogeneous translation, since then homogeneous interfaces would not be required. By contrast, the next security problem is inherent in the heterogeneous translation,

Visibility of Types

The JVM security model supports only two kinds of visibility for classes: package-wide and public visibility. It is not possible to use a class to reference objects outside a package unless the class is declared to be public. (This restriction holds even if the verifier is disabled, since the JVM specification [LY96] requires the virtual machine to throw an `IllegalAccessError` if a class refers to any other class that is in another package and not public.)

Consider an instantiation `C<D>` of a parameterised class `C` defined in package `P`, applied to a parameter class `D` defined in a different package `Q`. There are two possibilities: either class `D` must be public (in which case we can place the instantiation in package `P`), or else the body of class `C` must refer only to public classes (in which case we can place the instantiation in package `Q`).

Since classes usually refer to non-public classes in their package (otherwise, why have packages?), the Pizza compiler in its heterogeneous version limits itself to the first case: classes used as type parameters of public types defined in another package must be themselves be public. This rule severely restricts the utility of the Java visibility model.

Further, even if one were allowed to change the Java security model, it is not clear how to fix this problem.

5 Performance Evaluation

We initially believed that the heterogeneous translation would result in notably faster code than the homogeneous translation. We expected a certain loss at start time due to the class expansion, but assumed that than the execution — especially of classes parameterised with basic types — should be faster, as the heterogeneous translation does not need to convert between boxed and unboxed representations.

To evaluate the heterogeneous translation we compared the execution of several small benchmark programs. We used our own class loader in both cases. We ran each test at least 50 times. We ran all benchmarks on Sun's JVM 1.1 final on a Sun Ultra Sparc 1.

Micro Benchmarks

Our micro benchmarks try to measure one aspect of Java's execution in both translations. The code for these benchmarks is found in Appendix A.

The two versions of the *list-reverse* benchmark are designed to estimate the speedup of the heterogeneous translation for member access, as well as its slowdown for polymorphic methods.

The `List<int>` benchmarks show that polymorphic methods slow down significantly as every access to the list objects is made via the homogeneous interface. The instance reverse method shows only a slight speedup; we had expected a greater difference. We have smaller startup costs in the `List<String>` benchmarks as the class expansion is much easier for reference types.

The *cell* benchmark measures access of a variable of a parameterised type. This time, the heterogeneous translation yields a significant speedup for base types, which has to do with the fact that the variable accesses dominate all other costs. With a reference type, the heterogeneous translation shows a smaller speedup, since instead of an unboxing method call only an additional type cast is required. The cell benchmark was also used by Myers *et al.* to to test their implementation of parameterised types for Java [MBL97]. Our findings show a somewhat larger speedup for the heterogeneous translation compared to theirs (27% as opposed to their 14%).

The next set of benchmarks were designed to compare the efficiency of array accesses in both translations. We used two implementations of a reverse function for Pizza's `Vector` class. One is part of the class itself and accesses all data directly. This version does no runtime wrapping for the basic types in the homogeneous translation within the `Vector` class. But it has the array access overhead including wrapping and unwrapping in the `pizza.support.array` class.

Table 1. Micro benchmark results; times given in seconds

benchmark	iterations	homogeneous	heterogeneous	%
Cell<int>	1,000,000	4.26	3.10	73
Cell<Integer>	1,000,000	3.22	3.09	96
List<int> instance reverse	10,000	110.34	93.37	85
List<int> static reverse	10,000	110.48	403.77	365
List<String> instance reverse	10,000	109.77	94.76	86
List<String> static reverse	10,000	108.78	371.97	342
Vector<int> internal	10,000	78.71	21.60	27
Vector<int> external	10,000	125.81	42.55	34
Vector<String> internal	10,000	39.41	22.58	57
Vector<String> external	10,000	62.33	42.54	68
Hashtable<int, ..>	10,000	161.20	276.54	172
Hashtable<Integer, ..>	10,000	161.03	168.41	105

The second version is not local to the `Vector` class and has to access the data via methods. The homogeneous translation needs to introduce wrapping and unwrapping for basic types as well as for the arrays.

The vector benchmarks show a large speedup for `Vector<int>`. The homogeneous translation performs not quite as bad with a reference type as the element type, since then no runtime wrapping overhead is incurred. The differences between the two translations decrease in the external version of the benchmark since there array accesses are less frequent in the instruction mix.

A slightly more involved benchmark measured the efficiency of *hashtable accesses* under both translations. We measured the efficiency of the `get` operation for hashtables with keys of the basic `int` type. To our surprise, the heterogeneous translation performed much worse than the homogeneous translation on this benchmark. The reason for this effect is that a `get` operation on a hashtable involves several calls to methods `hashCode` and `equals` of `get`'s key parameter. In the homogeneous translation, a hashtable key will be boxed once, before it is passed as a parameter to `get`. But in the heterogeneous translation the key will be passed in unboxed form, and will then be boxed each time a `hashCode` and `equals` method is invoked. Hence, we have increased rather than reduced the number of type conversions that need to be performed. The same effect does not arise if the key parameter is of a reference type, since then all type conversions in `get` are widening casts from a reference type which do not translate into any bytecode instructions at all.

Macro Benchmark

As large benchmark we used the execution of the Pizza compiler itself. We translated it once homogeneously and once heterogeneously. Both versions then did the same job. They translated the compiler source and the sources of Pizza's API packages `pizza.lang` and `pizza.util` to heterogeneous classes.

We observed that under the heterogeneous translation the number of loaded classes increased by 85% and the total code size increased by 60%. Code size was measured as the number of bytes passed to the class loader's `defineClass` method. This includes both constant pool and code blocks of classes. We further observed a slowdown of 26% for the heterogeneously compiled compiler. This slowdown was rather unexpected. To isolate the different factors that might contribute to the slowdown we ran the compiler twice on the same data such that during the second run no more classes needed to be loaded. This showed that of the total slowdown 19 percentage points are attributable to increased class loading overhead and 7 percentage points are attributable to execution overhead. The additional class loading overhead is incurred both by the fact that many more classes need to be loaded, and by the fact the loading of the additional classes involves an extra expansion step.

A major factor in the execution overhead is the use of polymorphic methods such as `List.map`, `List.forAll`, which are heavily used in some parts of the Pizza compiler. The `map` method in particular is expensive since it involves the construction of parameterised object instances through the reflection library.

Table 2. Execution of the Pizza compiler

	loaded classes	size	system classes	duration in seconds
homogeneous	213	702 kB	28	93
heterogeneous	393	1251 kB	31	117

parameterised classes	171
templates used	16
size of the templates	53.1 kB

6 Conclusion

In summary, we found that at least for the current JVM implementation the heterogeneous translation of parametric polymorphism has not fulfilled its initial promise. A significant increase in code size did not yield a clear improvement in runtime efficiency. We also noted a severe incompatibility between the heterogeneous translation and Java's package based security model.

Sun's JVM has several peculiarities which affected the outcome of the benchmarks: The memory management is very inefficient, loaded classes are looked up in a linear array, which slows down execution as more classes are loaded, and all instructions are interpreted. It is unclear how the comparative performance of both translations would be affected with a different machine. A better memory management would work primarily in favor of the homogeneous translation, since it makes boxing operations more efficient. On the other hand, a better class loading scheme would primarily benefit the heterogeneous translation.

We also attempted to run our benchmarks on Microsoft's virtual machine for Java (JView 2.0 beta on Windows NT), but were not able to complete them, due to problems in Microsoft's implementation of the reflection library. *In the future, we hope to have additional data for both the latest Microsoft virtual machine and Sun's Hotspot machine.*

Acknowledgments

This work was supported by Sun Laboratories Academic Collaboration Grant "Measuring the Cost of Parameterised Types in Java." We thank David Detlefs and Guy Steele for their support, suggestions, and helpful criticisms.

References

AFM97. Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java language. In *Proc. ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, 1997.

Bru97. Kim Bruce. Increasing Java's expressiveness with ThisType and match-bounded polymorphism. Technical report, Williams College, 1997.

CDG$^+$88. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report 31, DEC SRC, 1988.

FKF98. Matthew Flatt, Shriram Krishnamourthi, and Matthias Felleisen. Mixins for java. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, January 1998.

GJS96. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Java Series, Sun Microsystems, 1996. ISBN 0-201-63451-1.

HM95. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, pages 130–141, January 1995.

Ler90. Xavier Leroy. Efficient data representation in polymorphic languages. In P. Deransart and J. Małuszyński, editors, *Programming Language Implementation and Logic Programming*, pages 255–276. Springer-Verlag, 1990. Lecture Notes in Computer Science 456.

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

MBL97. Andrew C. Myers, Joseph. A. Bank, and Barbara Liskov. Parameterised types for Java. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 132–145, January 1997.

oD80. United States Department of Defense. *The Programming Language Ada Reference Manual.* Springer-Verlag, 1980.

OW97a. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

OW97b. Martin Odersky and Philip Wadler. Two approaches to type structure, 1997.

SA95. Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc. 1995 ACM Conf. on Programming Language Design and Implementation*, (ACM SIGPLAN Notices vol. 30), pages 116–129, June 1995.

Str86. Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

Tho97. Kresten Krab Thorup. Genericity in java with virtual types. In *Proc. ECOOP '97*, LNCS 1241, pages 444–471, June 1997.

Tor97. Mads Torgersen. Virtual types are statically safe. Note, circulated on the java-genericity mailing list, 1997.

A Code of Micro Benchmarks

Static List Reverse

```
static <A> List<A> reverse(List<A> xs) {
  List<A> ys = Nil();
  while (true) {
    switch (xs) {
    case Nil(): return ys;
    case Cons(A x, List<A> xs1): ys = Cons(x, ys); xs = xs1; break;
} } }
```

Instance List Reverse

```
class List<A> {

  ...

  List<A> reverse() {
    List<A> xs = this;
    List<A> ys = Nil();
    while (true) {
      switch (xs) {
      case Nil(): return ys;
      case Cons(A x, List<A> xs1): ys = Cons(x, ys); xs = xs1; break;
  } } }
}
```

Cell Access

```
class Cell<A> {
  A elem;
  Cell() {}
  void add(A elem) { this.elem = elem; }
  A get() { return elem; }
}
Cell<int> c = new Cell();
c.add(42);
for (int i=0; i < iterations; i++) int j = c.get();
```

Internal Vector Reverse

```
public class Vector<A> {
  ...
  public void reverseElements() {
    for (int i = 0; i < elementCount / 2; i++) {
      A e = elementData[elementCount - i - 1];
      elementData[elementCount - i - 1] = elementData[i];
      elementData[i] = e;
    } } }
```

External Vector Reverse

```
public void reverseElementsInt(Vector<int> v) {
  for (int i = 0; i < v.size() / 2; i++) {
    int e = v.elementAt(v.size() - i - 1);
    v.setElementAt(v.elementAt(i), v.size() - i - 1);
    v.setElementAt(e, i);
  } }
```

Main Loop of Hashtable Benchmark

```
for (int i=0; i < 1000; i++) table.put(i, "A"+i);
for (int j=0; j < iterations; j++) {
  for (int i=0; i < 1000; i++) {
    String s = table.get(i);
  } }
```