Implementing Parameterized Types in Java

Master's Project Report

Hugh Vidos

Major Professor: Timothy A. Budd
Committee Members: Margaret M. Burnett, Curtis R. Cook

August 28, 1997

Implementing Parameterized Types in Java

Master's Project Report

Hugh Vidos

August 28, 1997

1. Introduction

The goal of this project was to investigate the addition of parameterized types to the Java programming language. Two different parametric polymorphism mechanisms were developed and compared. The first was a preprocessor and the second was a compiler.

Parameterized types allow a programmer to create generic programs. Much as a function parameter allows the value of a variable to be changed each time the function is called, a type parameter allows the type of a variable to be changed. This allows the creation of classes that can have the type on which they operate specified at compile time.

The principal reason for parameterized types is code reuse. Generic and efficient type-safe libraries are easily created which programmers can instantiate with a type parameter when the library is needed. This research creates a mechanism that allows two similar classes that differ only in the type of value they operate on to share the same function bodies. Two of the main benefits are reducing programming time and reducing errors in the program[Stroustrup91].

Parameterized types allow for the easy creation of reusable libraries. For example, the Standard Template Library (STL) in C++ relies heavily on parameterized types.

2. Description of Research

2.1 Background

There are three ways to compose object behavior in an object-oriented system [Gamma95]. The first is class inheritance. The second is object composition. The third is parametric polymorphism. The first two techniques are already well defined in Java, but the third technique is not and is the focus of this research. Parametric polymorphism differs from object composition in that it is only applicable at compile time. It differs from class inheritance in that the generic class does not have to be overridden.

Both C++ [Stroustrup91] and Leda [Budd95] support parameterized types. In C++ they are called templates and have the following form (Only the declaration is shown, as the class definition is unimportant in this example. In C++, comments are preceded by two forward slashes.):

```
template <class T>
class List
{
    //body of the List class
}
```

This would be instantiated as:

```
List<int> intList;
```

A List would be created which is called `intList` and contains values of type `int`. C++ handles templates using a mechanism similar to macro expansion. When the compiler sees this type of declaration, it will use the template code for List to generate a new class substituting `int` for T. If another type of List is declared, the compiler will also generate the code for that type of List as well. If there are many different declarations, the code size will increase proportionally.

Leda[Budd95] has a similar syntax, but the mechanism is much different. The major difference between the way C++ and Leda handle this mechanism is that Leda only needs one copy of the List class for every instantiation. In Leda, a parameterized type would have the following form (Again, only the declaration is shown, as the program is unimportant in this example. In Leda, comments are enclosed in curly brackets.):

```
class List [T : object]
begin
    { body of the List class }
end;
```

This would be instantiated as:

```
intList : List[int]
```

This would also create a List called `intList` that contains values of type `int`. In both examples the type parameter is T, in C++ T is of type class and in Leda T is of type object. Since both Leda and Java have a common inheritance tree rooted with the class object, Leda's implementation seems to be more natural.

2.1.1 Problems and solutions

Java is very similar to Leda in the way it treats inheritance. All classes inherit from a common class `Object`. Polymorphism [Budd95] means that a variable that is declared as holding one type can hold that type or any subtype. Since all classes are subtypes of `Object`, polymorphism can be used to simulate a parameterized type mechanism. A variable can be declared to be of type `Object` statically. At run time it can actually hold any type and then be converted back into the appropriate type using a run time-type-cast. However, this is a much slower mechanism than the proposed method. The compiler should instead perform all type checks since the class type will be known at compile time.

This mechanism will only work on objects. Java allows the use of a few primitive types instead of objects. In most cases, those types can be used as either a primitive or as an object. For instance, to represent an integer, the type `int` or the wrapper class `Integer` can be used. It is possible to perform almost all the same operations on an `Integer` as on an `int`, the only exception being arithmetic and multiplicative operations. The reverse, however, is not true. Polymorphism is an object-oriented concept and does not apply to types that are not objects. It might be possible to detect non-objects and then create new wrapper objects, but this problem can also be solved by the use of discipline on the programmer's part. As creating object wrappers would not be adding any new level of "generic-ness" to Java, it is beyond the scope of this project.

2.2 Goals

In a recent paper [Budd96], four goals of a parameterized type system in Java are proposed:

1. The addition of a parameterized type mechanism should seem like a natural extension to the existing Java language
2. The mechanism should permit the creation of type safe data abstractions. That is, the mechanism by itself should not permit the introduction of type errors into programs.
3. The mechanism should, to the greatest degree possible, be applicable at compile time, making minimal demands for run time checking.
4. Finally, while the addition of the mechanism will naturally necessitate changes to the Java compiler, it should be possible to add the mechanism to the existing Java system without requiring the introduction of any new bytecode instructions.

My primary goal will be to implement the parameterized type mechanism while also supporting the above four goals and not compromising Java security. A secondary goal is to avoid code bloat.

2.3 How objects are type cast in Java

Java stores all objects on the heap and references to those objects on the stack. Any variable in Java can be polymorphic. This makes polymorphism much easier in Java than in C++ (variables have to be declared differently to be polymorphic in C++). When a type conversion is necessary, the source object is expected to be on the top of the stack. The type conversion operator then examines the object referred to by the reference at the top of the stack. If it is the proper type or it can be resolved to the proper type, then the stack is left unchanged and that reference can now be treated as if it referred to the expected destination type. In the generic classes created by this mechanism, the types are always known. Even though the generic class has to widen the type information, the type of any value in that class is known at compile time for the compiler and at pre-compile time for the preprocessor. Any member function that returns the type specified in the type parameter to the generic class can potentially cause an unnecessary type conversion. The same is true for accesses to class data fields of the parameterized type in the generic class. With the compiler, it is possible to remove the conversion operations that would otherwise be necessary. With the preprocessor, that is not permissible. The compiler, not the preprocessor generates the conversion operations, and so the preprocessor cannot remove them. The actual impact of removing the typecast instructions is explored in section 2.11.

The Java Virtual Machine has two different methods to handle type conversion, object to object and primitive to primitive. There are fifteen total instructions that can perform type conversions. There are no explicit conversion operators from object to primitive or vice-versa.

There is only one instruction for converting an object to another object: *checkcast*. The operator checks the object reference on the top of the stack. If it is null or can be cast to the expected type, then the operand stack is unchanged; if not, a ClassCastException is thrown. The following rules determine whether it is legal to cast an object from type S to type T. [Lindholm97]

- If S is an ordinary (non-array) class, then:
  - If T is a class type, then S must be the same class as T or a subclass of T.
  - If T is an interface type, then S must implement interface T.
- If S is a class representing the array type SC[], that is, an array of components of type SC, then:
  - If T is a class type, then T must be `Object`.

- If T is an array type TC[], that is, an array of components of type TC, then one of the following must be true:
  - TC and SC are the same primitive type.
  - TC and SC are reference types and type SC can be cast to type TC by these runtime rules.

The two following code listings demonstrate this conversion. The Java source code is in figure 1. The assembly code for the source code in figure 1 is in figure 2. All the lines from the Java source also appear in the assembly in bold. Any assembly instructions generated by a line of Java source code appears afterwards (there is not necessarily a one-to-one mapping). The conversion operator has been highlighted in the assembly listing.

```
class objectCastTest {
  public static void main (String args[]) {
    Object a = new String("Hi");
    String b;
    b = (String)a;
    System.out.println(b);
  }
}
```

Figure 1

```
1:    class objectCastTest {
2:      public static void main (String args[]) {
3:        Object a = new String("Hi");
00000000   new                          java/lang/String
00000003   dup
00000004   ldc                          "Hi"
00000006   invokenonvirtual             java/lang/String.<init>(java/lang/String)
00000009   astore_1                     a
4:        String b;
5:        b = (String)a;
0000000a   aload_1                      a
0000000b   checkcast                    java/lang/String
0000000e   astore_2                     b
6:        System.out.println(b);
0000000f   getstatic                    java/lang/System.out
00000012   aload_2                      b
00000013   invokevirtual                java/io/PrintStream.println(java/lang/String)
7:      }
00000016   return
8:    }
```

Figure 2

Two of the remaining fourteen conversion operators are for converting from ints to chars and bytes. These operators are both narrowing conversions into unsigned types, so sign and magnitude information may be lost.

The other twelve are for converting from one of the four primitives, double, float, long or int to another one of those four primitives. As a conversion from a primitive to the same primitive is trivial, there is no conversion operator. The primitive conversion operators are listed in Table 1. As the type of the source primitive will be known at compile type, the compiler should always choose the correct operator. The operator expects to find the appropriate type on the top of the stack when it is executed. If the wrong type is at the top of the stack, it will be treated as the expected type; there is no error checking for types specified in the virtual machine specification. The operator pops the value at the top of the stack, converts that value into the destination type and pushes the new value onto the stack. Precision and magnitude may be lost when converting between two primitives. Specifically, conversion from a floating point number (double or float) to an integer (long or int) will result in a loss of any fractional information. The primitives can be ordered in terms of decreasing magnitude as double, float, long and int. Any conversion from a type to any type on its right may result in narrowing or loss of magnitude. Any conversion from a type to any type on its left will result in a widening conversion and will be exact. The two following code listings demonstrate this conversion. The Java source code is in figure 3 and the assembly code for code in figure 3 is in figure 4.

Destination Data Type \ Source Data Type	double	float	long	int	char	byte
double		f2d	l2d	i2d		
float	d2f		l2f	i2f		
long	d2l	f2l		i2l		
int	d2i	f2i	l2i			
char				i2c		
byte				i2b		

Table 1

```
public class castTest {
  public static void main (String args[]) {
  double x = 4.3;
  int y;
  y = (int)x;
  System.out.println(y);
  }
}
```

Figure 3

```
1:    public class castTest {
2:      public static void main (String args[]) {
3:      double x = 4.3;
00000000   ldc2_w                    4.300000
00000003   dstore_1                  x
4:      int y;
5:      y = (int)x;
00000004   dload_1                   x
00000005   d2i
00000006   istore_3                  y
6:      System.out.println(y);
00000007   getstatic                 java/lang/System.out
0000000a   iload_3                   y
0000000b   invokevirtual             java/io/PrintStream.println(int)
7:      }
0000000e   return
8:    }
```

Figure 4

Only the checkcast instruction operates on objects. The other conversion operations operate only on primitive types and primitives cannot be used as the type of a polymorphic variable. Therefore, the only bytecode instruction that must be detected for optimal performance is checkcast.

2.4 Syntax for the parameterized type mechanism

Currently C++ and Leda have a well-defined syntax for parameterized types. Adding a new keyword to the Java language would require that programmers not use that keyword in any of their programs unless they are adding a parameterized type. This could force programmers to rewrite their programs if they are to use the modified Java compiler regardless of whether or not they are actually using parameterized types. For that reason, I did not want to add a new keyword, such as *template*, to the Java language, so the implementation will actually be closer to Leda than C++.

The initial implementation of the parameterized type mechanism used the Leda syntax. However, the Leda syntax introduced a minor problem. The square brackets used for parameterized types are also used in array declarations. The JavaCC tool [Sun96] will create a parser for any LL(1) language. It is possible to specify how far ahead to look to resolve this type of ambiguity with JavaCC. If the parser looked for [ ] or [*integer value*] then it could be determined that the declaration was an array. The use of a lookahead whenever square brackets are encountered would solve this problem, but this seemed to be a needless addition to the parser since angle brackets, as used in the C++ syntax, did not introduce any ambiguity. For instance an array of objects of type GenericType, called array_x, can be declared as:

```
GenericType[] array_x;
```

If GenericType were changed to accept a parameterized type, array_x would now be declared as:

```
GenericType[object][] array_x;
```

Using angle brackets, this changes into:

```
GenericType<object>[] array_x;
```

Leda syntax dictates that the name of the parameterized type is listed first followed by a colon and then the parent type of that object. C++ syntax dictates that the keyword class is listed first followed by the name of the parameterized type. Neither method introduces any ambiguity, however as they each specify an ordering which is the reverse of the other, the choice of ordering could potentially cause confusion. There are two arguments to support the C++ ordering. First, C++ and Java both the variable type and the variable name in a declaration. Second, as a significant number of Java programmers have had exposure to the C++ language and C++'s angle brackets are used in specifying parameterized types instead of Leda's square brackets, the C++ ordering was chosen. The colon separator and the keyword class are not used. Instead, the parent type of the object must be specified. As an additional argument supporting this decision, the Java language designers took great pains to make Java similar to C++, I wanted to make the language addition as unobtrusive as possible. The C++ and Leda examples above could be written in Java using the proposed mechanism as:

```
class List <object valueType>
{
    ...  //body of the List class
}
```

This would be used as:

```
public List <Integer> intList;
```

This fulfills the first goal -- it seems like a natural extension to the Java language.

2.5 Implementation

To reiterate, the goals are: (1) The syntax of the parameterized type should seem to be a natural fit, (2) The mechanism should permit type-

safe data abstractions, (3) The mechanism should be applicable at compile time and (4) No changes should be made to the Java bytecodes. The syntax that is finally chosen will work regardless of the implementation. Therefore, if a syntax for the parameterized type mechanism is found which is a natural fit for Java, goal 1 will always be met. In addition, the decision to not modify the existing bytecode can be achieved by any of the four proposed methods, so goal 4 will also always be met. There are four possible ways of implementing this language addition. Two involve differences in the software mechanism, compiler versus pre-compiler. The other two possibilities deal with how class instantiation is supported, one instantiation of each object versus one instantiation of each object for each different type.

1. Create a pre-compiler that uses one instantiation of each object. This method will take source code and generate new source code for the compiler. This method is constrained to converting the "enhanced Java" source code into JDK 1.1.3 [Sun97b] source code. If a variable is declared to be of type object, it can later hold any type of value, but a run time-type-cast will be necessary to use the value. So, goals 1, 2 and 4 are satisfied, but goal 3 is only weakly met.
2. Create a pre-compiler that creates a new instantiation for each different type of object. This method will take source code and generate new source code for the compiler. This method is constrained to converting the "enhanced Java" source code into JDK 1.1.3 source code. It is currently possible to take a class and cut and paste different types into it to get multiple classes operating on different types. If the code is generated automatically, no new errors should be introduced. Also since there is a different class for each type, no run time-type-casts would be necessary. The drawback of this method over method 1 is that larger executables would be generated. Therefore, goals 1, 2, 3 and 4 are satisfied. C++ uses this method to create generic classes. (C++ initially used a preprocessor to convert the C++ specific code into C code. Now generic classes in C++ are most likely handled by the compiler instead of the preprocessor, but this is up to the compiler implementation)
3. Create a compiler that uses one instantiation of each object. The compiler would be modified to generate bytecode based on the enhanced Java source code. No new bytecodes would be added. The current implementation of the compiler also needs to use type conversion operations. Therefore, goals 1, 2 and 4 are satisfied, but goal 3 is only weakly met. Leda uses this method to create generic classes.
4. Create a compiler that creates a new instantiation for each different type of object. The compiler would be modified to generate bytecode

based on the enhanced Java source code. No new bytecodes would be added. The drawback of this method over method 3 is that larger executables would be generated. Therefore, goals 1, 2, 3 and 4 are satisfied. C++, Pizza and Jump use this method to create generic classes.

Many times decisions are made based on comfort and not on appropriateness. I wanted to avoid that pitfall and not choose an implementation simply because it was commonly used. I think the C++ implementation is a good, but it is not the best for this particular application. An added benefit of not using the C++ implementation is the avoidance of code bloat. After considering the above methods, methods 1 or 3 seem to be the best fit to meet the proposed goals.

2.5.1 The Parser

There are two possible ways of creating generic classes in Java using a preprocessor. The first is to create only one instantiation of any generic object and type cast that object into the appropriate type. Leda uses this mechanism, but not the preprocessor, to create generic classes. In the following code, a single List class would be created. It would have two instances, one for int_list and one for obj_list. To access any value in the int_list object, it would have to be cast into an Integer. The obj_list object would similarly have to be cast into Objects. The preprocessor takes "enhanced Java" source code and converts it into JDK 1.1.3 source code. In this case, the generated Java source code would contain a single List class and every time objects of that type are created or used, they may need to be cast into the expected type.

```
List <Integer> int_list;
List <Object> obj_list;
```

The second way is to create a new object for every type of generic class. C++ uses this mechanism, but not necessarily the preprocessor, to create generic classes. In the above example, two List classes would be created each with a single instantiation. The original List class would not appear in the generated source code. Instead, there would be two classes with internal names such as List_Integer and List_Object. The first would be a List class that operated solely on Integers and the second would be a List class that operated solely on Objects. Therefore, no type casts would be necessary to access the values in int_list or obj_list, but there would be more code generated (and loaded by the run time environment).

Goal 3, stated earlier, said that the mechanism should, to the greatest degree possible, be applicable at compile time, making minimal

demands for run time checking. The first preprocessor method does require some run time type casts, but it could be considered to be minimal. The second method requires no run time typecasts and would therefore seem to be a better implementation. However, the programs generated by this method would require more classes and the program would be larger. Any benefit gained by not requiring type casts would seemingly be lost in the additional resources to run the resulting programs. If run time execution speed is the most important consideration, then this is an invalid assumption; however, if this is the case, then most developers will choose to not use an interpreted language such as Java. For this reason, the first method was chosen when implementing the preprocessor.

At the time I started this project, there were two parser generators that generated Java source code from a grammar. There is JavaCC (the Java Compiler Compiler), written at Sun and CUP (Constructor of Useful Parsers) [CUP96] written at Georgia Tech. JavaCC is top down variable lookahead parser LL(k) and CUP is a bottom up parser LALR. CUP was in its Alpha release and had not been updated for over a year. JavaCC was the subject of frequent updates. Both have functions similar to yacc. They will take an appropriate (LL or LALR) grammar and generate source code to create a Java preprocessor which accepts the language specified by the grammar. As they both had similar functions but JavaCC was more supported, I choose to use JavaCC.

There was a downside to the benefit of bug fixes being contained in the constant updates to JavaCC. Namely, it was a constantly evolving product. It was originally called Jack, presumably to rhyme with yacc. There were two problems with using this tool. The first is that Sun released a new version of the Java language, as specified in the JDK 1.1 [Sun97b], which necessitated a new grammar. The second problem is that JavaCC is still beta quality software. New features are constantly being added and bugs are being fixed. Six different versions of JavaCC were released during the period when I created the preprocessor. (Jack 0.5, Jack 0.6-10, Jack 0.6-9, JavaCC 0.6-8 JavaCC 0.6 (Public Beta), JavaCC 0.6 and finally JavaCC 0.6.1). JavaCC 0.6 (Public Beta) was a major upgrade and included the necessary code for working with the 1.1 version of Java as opposed to the 1.0.2 version. The grammar also changed between the releases of JavaCC 0.6 (Public Beta), JavaCC 0.6 and JavaCC 0.6.1, but those were minor changes. The newer versions of the JavaCC program were more desirable due to their stability and increased parsing speed. JavaCC 0.6 would also process grammars from earlier versions of JavaCC. Unfortunately, there was no easy migration path for the JDK 1.0.2 grammar to the JDK 1.1 grammar except to reenter all changes originally made to the JDK 1.0.2 grammar.

2.5.1.1 The tJava preprocessor

The preprocessor is called tJava.Main and is a Java application. All source code files are expected to have the extension "*.tjava*". The preprocessor is being distributed as a zip file called tJava.zip. If that file is added to your classpath, then it can be run with the following command (assuming you want to process the file test.tjava):

```
java tJava.Main test.tjava
```

2.5.2 The Compiler

There are two ways for creating generic classes using a compiler that correspond to the two ways when using the preprocessor. Again, the first is to only create one instantiation of any generic object. (The method Leda uses) The second is to create a new object for every type of generic class. (The method C++ uses) The main difference between the methods described in section 2.5.1 and this section is that class files are generated instead of compilable source code. The compiler takes "enhanced Java" source code and generates Java class files directly.

Goal 3 stated that the mechanism should be applicable at compile time. Since this **is** the compile time, that goal is met. Neither mechanism needs to insert typecasts, for the runtime to evaluate, so the only difference between the two is that one generates fewer class files than the other does. So, once again, the mechanism that creates only one instantiation of any generic object was chosen.

There were two possible compilers that could be modified to create the tJava compiler, Sun's javac [Sun97b] and the JOLT project's guavac [JOLT96]. To get the source code to javac, I had to sign a non-disclosure agreement with Sun and also persuade them that I needed to have access to the source code. The source code for guavac is freely available for download on the Internet. This seemed to make guavac a more desirable compiler to modify. However, the guavac compiler was built using C++ and was not portable across platforms as was javac, which was built using Java. So, javac was chosen as the base compiler.

At first, the javac compiler seemed to be easy to compile. However, there were a few necessary changes, which made compilation much more difficult. If the compiler was not changed from the default namespace then depending on the classpath, the proper version might not execute. The binary version of Sun's javac, along with all of the other Sun Java language API's, are included with every version of the JDK in a file called classes.zip. Either the classpath needed to have the modified compiler earlier than classes.zip or the modified compiler needed to have a different namespace. It seemed to be a bad design decision to make the

ordering in the classpath determine whether the compiler would work, so the name space was changed. There are four packages that make up the javac compiler: sun.tools.asm, sun.tools.java, sun.tools.javac and sun.tools.tree. In all four cases, sun.tools was changed to tJava. However, once the namespace was changed, the compiler could no longer be compiled. There were numerous circular dependencies in most of the files, which comprised the javac compiler. When the compiler was in the sun.tools namespace, the binary in the classes.zip file could be used to compile any file when there was a circular dependency. Once the namespace was changed, either all the circular dependencies had to be removed or way to bootstrap the compilation needed to be found. Since I did not know if it was even possible to remove those dependencies, I chose to bootstrap the compilation. A new class file needed to be created, for each of the 174 classes in the compiler, which implemented the appropriate methods and was in the proper namespace. For each of the classes, I used a Java disassembler [Djava97] to create a Java assembly file. The namespace defined in that file was changed to the new namespace and then the file was reassembled using a Java assembler [Jasmin97] to create new class files in the proper namespace. Once this was done, the compiler could be recompiled into the new namespace and changes could be made to it. It turned out to be most practical to modify my classpath and leave the compiler in the default namespace for development and then move it to the new namespace when it was finished. This should not prove to be a problem for other users though, since only a binary release can be put into public distribution due to the terms of the non-disclosure agreement with Sun.

The final problem with the compiler was the removal of extra checkcast instructions. The only time that extraneous type conversions could be detected was when the source code file was being parsed. When code generation was being done, legitimate conversions cannot be distinguished from the conversions necessary for the parametric polymorphism mechanism. Unfortunately, the only way that the modified javac compiler would allow the parsing of parametric polymorphic variables was through the addition of a type conversion instead of changing the type of the internal representation. Since the impact of type conversion (as seen in section 2.11) was minimal, this was not pursued.

2.5.2.1 The tJava compiler

The compiler is called tJava.javac.Main and it is also a Java application. Source code files can have either the normal "*.java*" or "*.tjava*" file extensions. The compiler is also in the tJava.zip file. If that

file is in your path, then the compiler can be invoked (on the file test.tjava) with the following command:

```
java tJava.javac.Main test.tjava
```

2.6 Comparison of Preprocessor and Compiler

In terms of performance and ease of use, the compiler is much more convenient than the preprocessor is. It takes only one step to generate class files and if the javac compiler is used to compile code generated by the precompiler, then compilation times will be the same. However, if a different compiler is used, than the restriction of using a modified javac compiler may seem limiting. Both the compiler and the preprocessor generate the checkcast instruction but, as will be demonstrated later, this has minimal impact on the execution speed.

2.7 Scope of Parameterized Types

There are two other concerns with this introduction to the Java language. What can be defined in a generic class? Is the type parameter too limiting?

The template class cannot internally create any new objects of the parameterized type. When an object is instantiated, some portion of memory is set aside for the internal data and methods of that object. Every object needs to have its own memory for local variables; however there needs to be only one location with all the methods defined for all the instantiations of that object. The *Virtual Method Table* is the data structure that fills this need. If a class A is defined and 50 copies are instantiated, there will be 50 copies of the activation record of A, but only one virtual method table for A. Each instance of A will maintain a pointer to the virtual method table so that methods can be invoked. Consider the implications of creating a class B that is defined as a subclass of class A. When B is instantiated, it will need to have its own activation record and also its own virtual method table. Any inherited data fields will be at the beginning of the activation record and new data fields defined in class B will be at the end of the activation record. The virtual method table for class B will have a similar layout. If a variable is declared to hold objects of class A, then objects of class B can also be assigned to that variable.

```
class A {
    int e;
    int f;
    void x (void) {
         // code for method x
    }
}

class B extends A {
    int g;
    void y (void) {
         // code for method y
    }
    void z (void) {
         // code for method z
    }
}

class C extends A {
    int h;
    int i;
    int j;
    void x (void) {
         // overridden code for method x
    }
    void p (void) {
         // code for method p
    }
    void q (void) {
         // code for method q
    }
}
```

Figure 5

In the example in figure 5 classes B and C are subclasses of class A. An activation record for each class is shown in figure 6. The first entry in each activation record is a class pointer, this points to the location in memory for the virtual method table for that class. The virtual method tables for these three classes are then shown in the middle of figure 6. Every entry in the virtual method table points to the location in memory where the executable code for the corresponding method can be found.

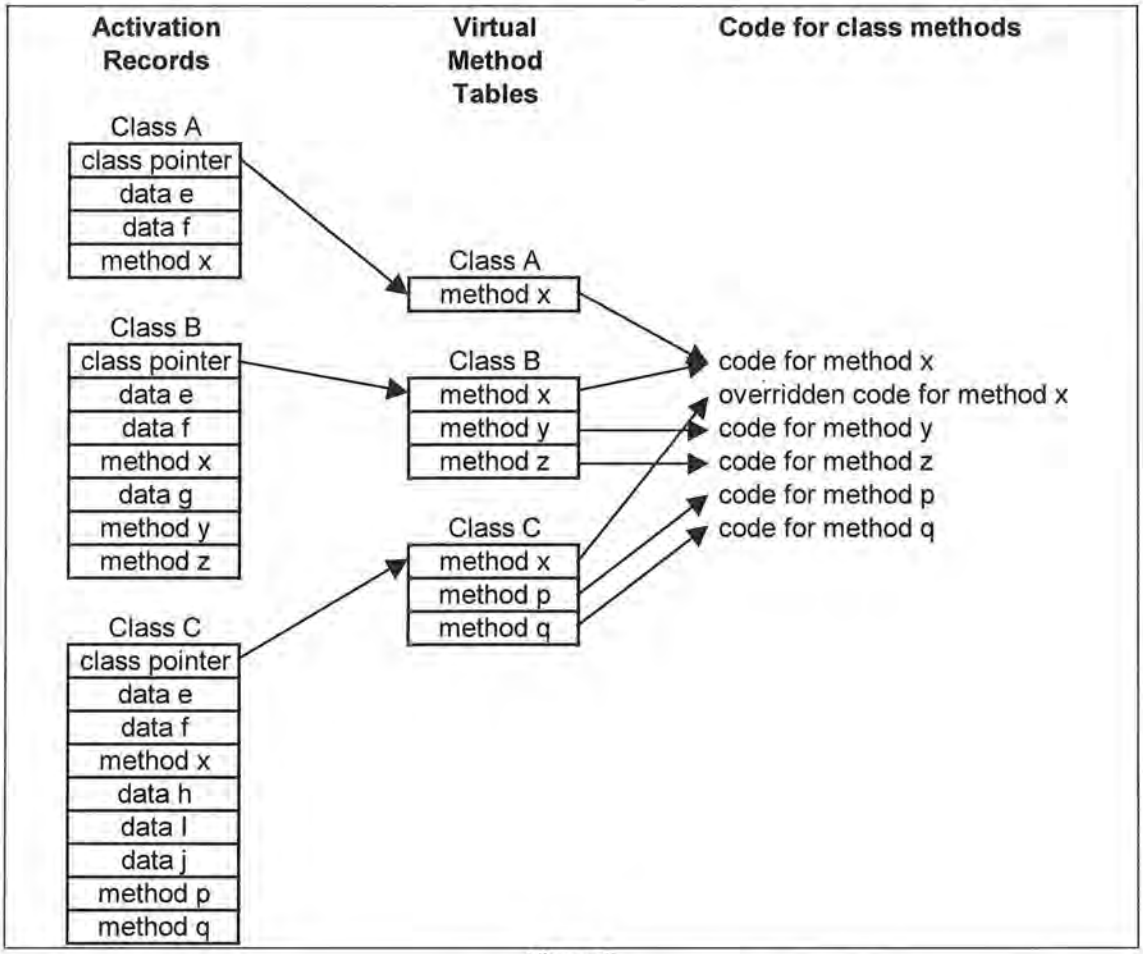

Figure 6

Java uses dynamic memory allocation for all objects, so the activation record is created on the heap. From the implementation standpoint, this means that when a variable is declared, space for that variable is set aside on the heap and a pointer to that space is kept in a variable on the stack. In Java, any variable can potentially be a polymorphic variable. A variable can hold any object of the declared type or any object of any subclass of the declared type. This is not always practical since a subclass will always have the same data fields and methods of its parent class, but the subclass may not behave in the same manner as the parent class. An object that has the same behavior as another object is a subtype of that object. In Java, objects that are subtypes **must** also be subclasses of another object; however, objects that are subclasses of another object need not be subtypes. In figures 5 and 6, classes B and C are subclasses of class A. Class B is a subtype of class A. Class C might be a subtype, but it cannot be determined without knowing what in method x was changed.

When an object is a subtype of another object, that object can be substituted for the parent object with no unexpected side effects. It can

mimic the behavior of the parent object and remain indistinguishable from any other instance of the parent object. This is known as substitutability. When creating a generic object, the instantiations will be subtypes of the original generic object.

```
class printTest <printingObject valueType> {
  private valueType val;
  public printTest () {val = new valueType();}
  public void message() {
    System.out.println ("Test "+val.message());
  }
}
```

Figure 7

```
class printTest <printingObject valueType> {
  private valueType val;
  public printTest (printingObject newVal) {
    val = newVal;}
  public void message() {
    System.out.println("Test "+val.message());
  }
}

abstract class printingObject {
  public printingObject () {}
  abstract public String message();
}

class hiObject extends printingObject {
  public hiObject () {}
  public String message() {return "Hi";}
}

class testObject extends printingObject {
  public testObject () {}
  public String message() {return "Test";}
}

public class test {
  public test () {}
  public static void main (String args[])
  {
    printTest<hiObject> hiPrint =
      new printTest(new hiObject());
    printTest<testObject> testPrint =
      new printTest(new testObject());
    hiPrint.message();
    testPrint.message();
  }
}
```

Figure 8

When a variable is declared inside a template class, the parent class of the parameterized type defines the type of the instantiated object. Therefore, the activation record and virtual method table for that parent class are used instead of the activation record and virtual method table of the parameterized type. No methods or data from the child class can be used and any gain from using a generic class is lost. For this reason, the generic class can not internally create any new objects of the parameterized type. The contrived example in figure 7 demonstrates the problem.

In the `printTest` class, when the object `val` is created, where is the code for the method `message`? Is it in the class `printingObject`? `HiObject`? `testObject`? Or some yet to be declared class that is a child of `printingObject`? The answer is that the code for the method `message` is expected to be defined in class `printingObject`. If an object of the *parameterized type* needs to be created and stored in the generic class, then it must be created externally and passed as an argument to the generic class. An implementation of the `printTest` class and supporting classes is given in figure 8:

In strongly typed object-oriented languages, there is the potential for type parameters to be overly constraining [Kim97]. It is possible to create a generic class X which takes a class of type Y, or subclass of Y, as its parameterized type. If the class Y has any attribute which is unnecessary to class X, then the use of class X has been constrained by class Y. In the above example, a generic class `printTest` is declared to take a type parameter of type `printingObject`. In this example, it is necessary to constrain the type parameter to the type `printingObject` because a method that is declared in `printingObject` is used (`message`). If the method `equals` were needed instead of `message`, then `printingObject` would be overly constraining since `equals` is defined in `Object`. In the case where a method may be needed in some uses of a generic class, but not all, then the class that defines that method may be too restrictive as a type parameter. The use of functional parameters would solve this problem. (For a more complete description to the problem, see An Approach to Type Constraints of Generic Definitions [Kim97]).

The C++ approach to function parameters is to allow a pointer to point to a function. Pointers are valid arguments and can thus be passed as arguments. The Java language does not allow functional parameters or function pointers. The addition of pointers of any type would greatly undermine the security model of Java, so that omission is understandable. The addition of functional parameters to Java would be useful and at first glance seem to not undermine the security model. However useful, this is beyond the scope of this research. Another possibility is the use of *functors*. In fact, the Standard Template Library

uses functors instead of function pointers. A *functor* is an object wrapper around a function. Technically, functors also override the () operator, but this is just to create a transparent mechanism. The idea of an object wrapper around a function is still possible in Java (since operator overloading is not possible in Java). This solution still would not be an elegant solution to this problem though. Java is a much more strongly typed language than C++. Any method that is implemented in the function object would have to be written to satisfy an interface or parent class. That parent class or interface would then be the constraint to the function, to which the functor is passed as an argument. An alternative would be to override a method that is defined in the *Object* class and is not needed in the implementation of the generic class. This is a very undesirable programming technique though. The resulting object would no longer be a subtype of `Object` and it is considered bad form to break the type – subtype relationship.

An important point to make about both implementations is that no type errors are introduced into the Java language. The preprocessor translates enhanced Java source code into JDK 1.1.3 [Sun97b] source code. The template aware compiler is simply inserting an instruction to do a run time typecast when a generic object is encountered in the source code. Even if the typecast instruction were not inserted, the object will still not be converted into another type. Only source code that is valid in the JDK 1.1.3 language specification will be compilable. Even if it were possible for a type error to be introduced into the enhanced source code, the compiler will catch that error and generate an error. If an error were to slip by the compiler, it would be caught by the run time security mechanism. Section 2.3 describes the way the Java runtime system handles object coercion. In essence, if an object is used in an improper manner, a run time type exception will be thrown.

Both of the implemented mechanism will always use a generic object as its declared type. Any new instances of that generic variable will be of the declared type. The types for generic objects are always created and used in a consistent manner based on the generic object declaration. Because of this, no type errors can be introduced into the language.

2.8 Java Security Model

Java has a well-defined 4-tier security model [Lemay96]. First, the compiler is expected to generate "safe" programs. As pointers were one of the largest causes of security problems in C/C++ (and similar languages), pointers were removed from the Java language. References to objects are still allowed, in fact, this is the only way to access objects, but these are

more secure than pointers. They cannot be forged and before an object can be cast, it must be proved to be a valid cast. Additionally, arrays have strictly enforced boundaries.

The second tier in the security model is the bytecode verifier. This subjects class files to rigorous security tests. No object can be accessed as anything other than its dynamic type; an object's fields cannot be accessed illegally. There must be no attempts to forge pointers. The run time stack must not overflow or underflow. There must be no access restriction violations. All methods must be called with the appropriate number and type of arguments. If any of these tests fail, then the byte code will not execute. So, for each basic block where the starting stack state is known, the stack can be verified. Java imposes the two following restrictions on the language: given only the stack and local variables (the type state) before the execution of any instruction, the type state afterward must be fully determined, and if there are multiple paths to arrive at the same point in the program, all paths must have the same type state upon arrival at that point. To aid in verification, Java bytecodes contain extra type information. This means that whenever possible, type casts are done statically by the bytecode verifier and not dynamically by the interpreter.

The third tier in the security model is the class loader. When a Java class is loaded into memory, it is placed in a *namespace*. Each namespace has an associated class loader. There is a namespace, and class loader, for local trusted classes and another namespace, and class loader, for untrusted classes loaded from remote network locations. There can potentially be a namespace for classes loaded from within a corporate firewall. Depending on the virtual machine, there can even be more namespaces than just these three, although their protection level will most likely be the same as the untrusted namespace as each namespace will probably be for a different network location. The following instructions are deemed as "dangerous" and methods in the untrusted namespace are not allowed to execute them [Flanagan96]:

- Read, write, rename or delete files on the local system either through the use of Java methods or system calls.
- List directory contents.
- Check for the existence of a file.
- Obtain the type, size or modification time of a file.
- Create a network connection to any other computer than the one from which the applet was itself loaded.
- Listen for, or accept, network connections on any port of the local system.

- Create a top-level window without a visible warning indicator that the window is "untrusted".
- Obtain the user's username or home directory name or read any of the following system properties: `user.name`, `user.home`, `user.dir`, `java.home` or `java.class.path`.
- Define any system properties.
- Make the Java interpreter quit.
- Load dynamic libraries or invoke any program on the local system.
- Create or manipulate any thread that is not part of the same `ThreadGroup` as the applet or manipulate any `ThreadGroup` other than its own.
- Create a `ClassLoader or a SecurityManager` object.
- Specify a `ContentHandlerFactory`, `SocketImplFactory` or `URLStreamhandlerFactory` for the system.
- Access or load class in any package other than the standard Java API.
- Define classes that are part of packages on the local system.

Classes loaded in the trusted namespace do not have these restrictions. The firewall namespace can relax a few or all of the above restrictions. One of the reasons for the namespaces is so that an anonymous applets cannot redefine the security model by replacing trusted classes with classes from the untrusted namespace.

The fourth tier in the security model is the security manager. The security manager implements the restrictions placed on classes in the untrusted namespace. In general, whenever a "dangerous" operation is about to take place the security manager is consulted. The security manager bases its approval to continue with a dangerous operation on which class loader was used to load the corresponding code. If the security manager disallows a dangerous operation, a `SecurityException` is thrown.

Both the `SecurityManager` and `ClassLoader` classes are abstract classes and must be extended in order to provide any security. Any Java distribution, such as Sun's, Microsoft's or Netscape's distributions, will have extended these classes to provide appropriate security. However, the user is still free to override these classes and change the security provided by those virtual machines. This will only apply to the local virtual machine.

Only the first security mechanism is enforced at compile time, the other three mechanisms are enforced at run time. If a compiler does not generate "safe" Java bytecode, the run time environment should catch any security violations. However, the run time environments are just

programs and can have bugs, which allow flaws in the security model to be exploited. Sun has released several updates to their Java Development Kits just to fix implementation flaws and not to introduce new features. The combination of both compile time and run time security mechanisms enhances the Java security model. If either of these mechanisms is released by an unknown third party, confidence in the overall security will drop. A compiler could be created that creates seemingly "safe" code and yet takes advantage of known security flaws in the run time environment. Or a run time environment could be created which either disregards the run time security mechanisms or subverts them entirely by causing unwanted side effects to otherwise safe Java code. Nonetheless, with a trusted run time environment, people have shown an indifference to the compiler used to create the Java programs that they actually run. If a person runs a Java applet using their favorite browser, there is no indication as to which compiler actually compiled the applet that they are running, and yet they still run the applet.

2.9 Other Concerns

If either the preprocessor or compiler break Java's security mechanism, it is unlikely that anyone will want to use them. Method 1, a preprocessor that uses a single instantiation for each different type of object, as defined above, does not change the compiler at all, so all normal security mechanisms in the compiler and the interpreter will be in effect. However, this method requires the programmer to use a pre-compiler in addition to the compiler, thus increasing the complexity of building executables. Method 3, a compiler that uses one instantiation of each object, changes the compiler, so some programmers might not be as willing to use it for this reason. This should not be a big concern because the run time environment should still catch all security violations and building executables should be the same (no extra steps).

The Sun compiler [Sun96a], *javac*, does not heavily optimize Java bytecode. Since the tJava compiler is based on *javac*, it also does not generated heavily optimized Java bytecode. Therefore, if a programmer were using an optimizing compiler, they might not want to use the tJava compiler. This would only be a concern if the compiler were used. If the tJava preprocessor were used, any compiler could then be used to generate executables.

Taking into account these concerns, it seemed that the best approach was to develop both a preprocessor and a compiler and let developers use whichever method they choose.

2.10 Success Criterion

The success of the project is based upon how well the four goals, outlined in section 2.2, were met.

1. The syntax of the parameterized type should seem to be a natural fit. This is successful if someone using or reading the mechanism is able to understand it without difficulty. I proposed a syntax in section 2.4. It is hard to empirically measure the ease with which this mechanism fits into the existing Java syntax. As Java was modeled using the syntax from C++ and the implemented parameterized type mechanism closely models that of C++, I consider this to be successful.
2. The mechanism should permit type-safe data abstractions. This is successful if the mechanism does not introduce type errors into the program. Additionally, by virtue of the fact that the compiler will be *creating* classes based on a parameter supplied by the programmer, this mechanism should help to reduce type errors. By using the type as a parameter, the compiler has the job of enforcing type safety instead of the programmer. Because the preprocessor uses only a single instance of each generic class, the preprocessor might not catch some type errors. The compiler that is used subsequently will not have all the type information available to the preprocessor and it will be unable to catch the type error. The run time environment will catch this error. This problem is exclusive to the preprocessor; the compiler will always be able to detect this situation. This problem is explained in detail in section 2.11. So, this mechanism introduces no new type errors and I consider it to be successful.
3. The mechanism should be applicable at compile time. If at all possible, the compiler should perform all changes to the program. This has the added benefit of allowing the compiler to perform type checking instead of relying on the run time environment. This is extremely successful if no run time type checks must be performed to support this mechanism. It will still be considered successful if few run time type checks must be performed to support this mechanism. Since both the preprocessor and the compiler do need to use a minimal amount of type casts to support parametric polymorphism, I consider this to be successful.
4. No changes should be made to the Java bytecodes. No new bytecodes should be introduced to support this mechanism. To allow the widest range of users to benefit from this addition, the existing Java virtual machine should be supported. If any new bytecodes are introduced or any existing bytecodes are changed, then a new virtual machine must be created to support those changes. This would preclude a vast majority of users from using Java applications and applets created with this mechanism. Consider a typical end user who is

using a web browser such as Netscape Navigator, Microsoft Internet Explorer or Sun's HotJava; all three of these browsers already have a Java virtual machine built in which adheres to Sun's Java Language Specification [Sun96b]. If Java class files are generated which change the Java bytecode, an end-user using one of these browsers will not be able to view/execute the corresponding Java programs. This is successful if no changes are made to the Java bytecodes. The Virtual Machine was not changed in any manner and no new bytecodes were added, so I consider this to be successful as well.

2.11 Timings

To demonstrate the differences between code generated by the preprocessor and then compiled by the javac compiler and code that could be generated by a template aware version of the javac compiler without type conversions, several test runs were performed. A vector was created with 300,000 `Integer` elements. Then the elements of that vector were referenced. The loop below was timed 15 times for various values of `max` (up to 300,000). Since the actual implementation of the compiler does the same manipulations to the source code as the preprocessor, it was not timed.

```
for (int i = 0; i<=max; i++) {
  temp = vec_ints.elementAt(i);
}
```

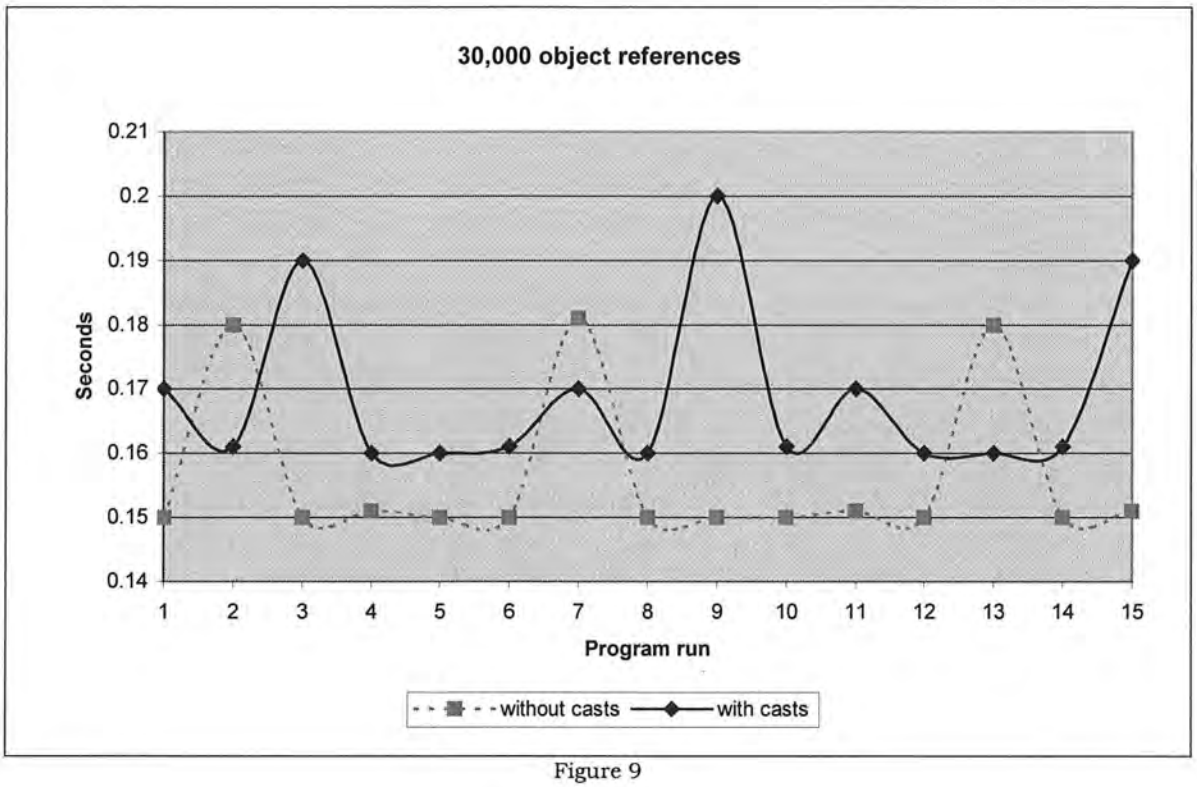

Figure 9

The standard javac compiler requires that a typecast instruction be present for this kind of operation. Since the type of vec_ints is known at compile time to of type `Integer`, the typecast is unnecessary. Therefore, a template aware compiler could dispense with the typecast. To simulate what would happen in byte code generated by the latter type of compiler, valid bytecode was generated by javac, disassembled, the assembly code was edited to remove the applicable checkcast instruction, and then the assembly code was re-assembled into a class file. The two series of data were used to compare the speedup of using a template aware compiler without type conversions.

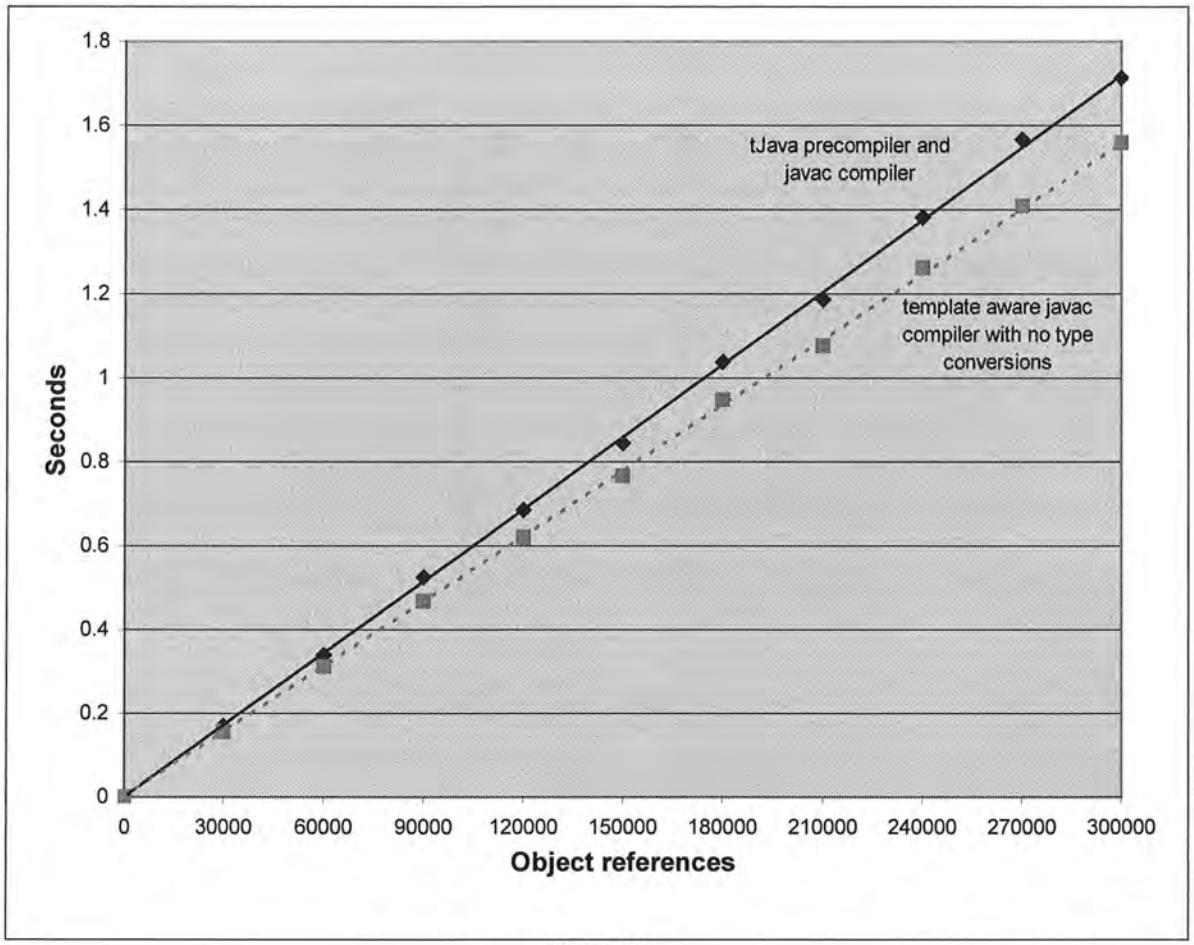

Figure 10

Figure 9 shows 15 trials for each version of the program with 30000 object references each. Figure 10 shows the average of 15 trials for each of the data set sizes listed at the bottom of the graph.

Both versions of the program have several anomalous spikes in execution time. At first glance, it could be assumed that these are caused by the Java virtual machine's garbage collection. However, disabling the virtual machine's garbage collection did not get rid of the anomalous readings. Currently, the only versions of the virtual machine, that I have access to, run under multiprocessing operating systems. An attempt was made to not run any other applications while the tests were performed, but the operating system can still interrupt any process to perform system level tasks. The tests were run under Windows NT 4.0 on a Pentium 166 with Sun's JDK 1.1.2.

As can be seen from figure 9 and figure 10, the impact of removing the typecast instruction is minimal. In figure 9, the program with casts is approximately 0.16 seconds and the program without casts is approximately 0.15 seconds. In figure 10, the timings are plotted as discrete points and a least squares linear regression is fitted to the data.

It can be observed that the growth of both data series is linear with respect to the input size and the both versions differ only in a constant multiple. The result of removing 300,000 checkcast instructions from a program results in a speedup of only 0.155 seconds.

2.12 Problems

There were several problems, both in syntax and in implementation, that were uncovered during the creation of the parameterized type mechanism. Most were described earlier along with their respective solution.

1. One of the first problems discovered during implementation was that objects of the parameterized type could not be created in the generic class.
2. The syntax was first chosen to be similar to the Leda language, but this introduced the lookahead problem with the square brackets. C++ syntax was used instead to solve this problem.
3. Another problem was the constant updates to both the JavaCC application and the Java grammar. There was no workaround and extra effort was required to keep up with the changes.
4. Also, a problem best described as the "sibling class" problem was identified. The preprocessor in an effort to be as generic as possible must "lose" scoping information. Given the three classes and one interface in figure 11, the problem can be described as follows. When the preprocessor sees the List class, all instances of T are replaced with Comparable. So if a class is declared as `List <comparableInteger> comparableIntegerList` it would be expected at compile time that any arguments passed to the `badTest` function would have to be of type `comparableInteger`. However since the preprocessor is widening the type of arguments to the parent of the parameterized type, any object which implements `Comparable` would be a valid argument to `badTest`. For instance, an object of type `comparableFloat` could be passed as an argument. The comparison of an `Integer` to a `Float` would have unpredictable results and an exception would be thrown. There are two potential solutions to this; the first is to use the method used by C++ - create a new class for every different declaration of a generic class. The second is to let the runtime environment throw an exception when this type of error occurs. Type casting seems to be the most desirable solution to this problem. There are two possible places where the cast can occur, in the generic class and in the calling method. If the cast occurred in the generic class, a different class would need to be created for each declaration of a generic class. If a cast were inserted into every instance of a method in the generic class where

the type parameter was the same as the parameterized type, then potentially many unnecessary casts would be inserted. Since the compiler uses no preprocessor, this problem does not exist when using the compiler.
Fortunately, after further thought, this turned out to be a non-issue. In the code in figure 11, what does it mean for one object to be less than another object? The `lessThan` method would need to call a method or access some data field in order to make a comparison. Since `lessThan` is operating on a `Comparable` object, the method or data field must be defined in the `Comparable` interface. The method would have to return some type and the data field would have to be specified as some type. In both cases, some concrete type must be specified and would be caught by the compiler as an implicit type conversion error.

```
public interface Comparable {
  boolean lessThan (Comparable T);
}

List <Comparable T> {
  T value;
  List (T value) { this.value = value; }
  boolean badTest (T testValue) {
    return value.lessThan(testValue);
  }
}

public comparableInteger extends Integer
       implements Comparable {
  // body of comparableInteger class
}

public comparableFloat extends Float
       implements Comparable {
  // body of comparableFloat class
}
```

Figure 11

5. A problem in the javac compiler regarding the namespace and circular dependency was discovered. Bootstrapping was used to solve this problem.
6. Finally there was a problem parsing enhanced Java source code in the compiler. Type conversions needed to be inserted to solve this problem.

3. Related Work

When I started work on this project, I was unable to find any similar research in progress. During the course of the research, numerous searches turned up other projects in various states of implementation. Currently I am aware of three other projects that are doing similar research. There are also rumors circulating that Sun is working a parameterized type mechanism to be included in a future version of the Java Development Kit, although nothing has been publicly released.

At MIT, [Bank96] Bank, Liskov and Meyers have proposed a mechanism for the creation of parameterized types in Java. Their research relies on the addition of two new bytecodes and also run time type checks to ensure type safety. This violates goal 3, (the mechanism should be applicable at compile time) and goal 4, (no changes should be made to the Java bytecodes). Also to support these changes, they propose to change the Java run time environment. This will make their method significantly less available to end-users.

Odersky and Wadler [Odersky96] have also proposed a mechanism for supporting parameterized types in Java. However, they have created a new language, Pizza, which is similar to Java, but nonetheless is a different language. Pizza also supports higher-order functions and algebraic data types. They propose to use the Pizza compiler to compile Pizza and Java code into Java bytecode. Pizza is a superset of Java much as C++ is a superset of C. While this is a noteworthy change, the language is modified enough to make the new syntax difficult to understand for current Java programmers. This compiler creates larger bytecode files then the method described in this paper. (A new instantiation is created for each different type of object, similar to C++'s STL)

Detlef Höffner [JUMP97] has also proposed a mechanism for supporting parameterized types in Java. However, he has essentially created a C++ compiler that also accepts Java source code and compiles to the Java Virtual Machine. Features such as operator overloading, templates and global variables and functions, which are part of the C++ language, are added to his compiler. These too are noteworthy additions to the language, but they break the Object Oriented Programming model of Java and allow Procedural Programming. This compiler also has the disadvantage that it creates larger bytecode files.

4 Conclusion

One of the most useful kinds of classes is the container class, that is, a class that holds objects of some (other) type [Stroustrup91]. Generic

classes allow the creation of classes that operate on a type that is unspecified at design time, but known at compile time. This mechanism allows a programmer to create generic class which will be usable with a wide variety of types, some of which may not have even been considered by the programmer. A list container class can be built using this mechanism which works for any type of object. When the list object is built, the type that it operates on is abstracted away and specified later when an instance of that object is declared.

I have modified the Java language through the inclusion of parameterized types. I have implemented two mechanisms for this, a pre-compiler and a compiler. Both approaches allow a set of container classes similar to the C++ STL. While this may not be original research, I added the type of container class library for Java and it is a welcome addition for the Java programming community. Furthermore, this mechanism does not incur the larger bytecode file penalty of the other methods. By adding parametric polymorphism to Java, the language mechanism for creating container classes is vastly simplified. By allowing the class type to be specified as a parameter, a type-safe mechanism for creating generic classes is added to the language. Furthermore, the implementation will ensure that this language addition does not require the programmer to learn obscure syntax, sacrifice execution speed or preclude end users from using the Java programs because they do not have an enhanced Java virtual machine.

References

[Bank96] Joseph A. Bank, Barbara Liskov and Andrew C. Meyers, *Parameterized Types and Java*, Technical Report MIT LCS TM-533, MIT Laboratory for Computer Science, Cambridge, MA, May 1996

[Budd97] Timothy A. Budd, *An Introduction to Object Oriented Programming*, Addison-Wesley, 1997

[Budd96] Timothy A. Budd, *Regarding Parameterized Types and Java*, November 5, 1996

[Budd95] Timothy A. Budd, *Multiparadigm Programming in Leda*, Addison-Wesley, 1995

[Cornell96] Gary Cornell, Cay S. Horstmann, *Core Java*, SunSoft Press, 1996

[CUP96] Scott Hudson, *Java Based Constructor of Useful Parsers (CUP)*, http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/home.html, March 1996

[Djava97] Shawn Silverman, *D-Java*, http://home.cc.umanitoba.ca/~umsilve1/djava, May 1997

[Flanagan96] David Flanagan, *Java in a Nutshell*, O'Reilly & Associates, Inc., 1996

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[Jasmin97] Jonathan Meyer, *Jasmin, (companion software to Java Virtual Machine)*, O'Reilly & Associates, Inc., 1997 http://cat.nyu.edu/meyer/jasmin

[JOLT96] The JOLT Project, *guavac 0.2.5 A free compiler for the Java Language*, http://www.redhat.com/linux-info/jolt/

[JUMP97] Detlef Höffner, *JUMP Compiler, a compiler for a superset of Java*, http://ourworld.compuserve.com/homepages/DeHoeffner

[Kim97] Myung Ho Kim, *An Approach to Type Constraints of Generic Definitions*, Sigplan Notices, June 1997

[Lam96] John Lam, *ATL: Rx for your COM Headaches*, PC Magazine, December 1996, pp333-338

[Lemay96] Laura Lemay and Charles L. Perkins, *Teach Yourself Java in 21 Days*, Sams Net, 1996

[Lindholm97] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Addison Wesley, 1997

[McGraw97] Gary McGraw and Edward Felten, *Java Security and Type Safety*, Byte Magazine, January 1997, pp63-64

[Odersky96] Martin Odersky and Philip Wadler, *Pizza into Java: Translating theory into practice*, to appear in: Proceedings 24th ACM Symposium on Principals of Programming Languages

[Stroustrup91] Bjarne Stroustrup, *The C++ Programming Language*, second edition, Addison Wesley, 1991

[Sun96a] Sun Microsystems, The Java Development Kit 1.0.2, http://java.sun.com/nav/download/index.html

[Sun96b] Sun Microsystems, *Java Language Specification*, version 1.0, http://java.sun.com/nav/download/index.html

[Sun97a] Sun Microsystems, *JavaCC Compiler Compiler*, version 0.6, http://www.suntest.com/

[Sun97b] Sun Microsystems, The Java Development Kit 1.1.3, http://java.sun.com/products/jdk/1.1/index.html

[Wang94] Paul S. Wang, C++ with Object-Oriented Programming, PWS Publishing Company, 1994