On Parametric Polymorphism in Java:

A NextGen Compiler Based on GJ

By

Huanling Lu

Graduate Program in Computer Science

Submitted in partial fulfillment
Of the requirements for the degree of
Master of Science

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
April, 2002

Abstract

Recently parametric polymorphism has been recognized as an important tool for modern programming languages. Languages which lack such a mechanism are considered for extension to support it. The current Java is one such language. In order to keep the Java language as simple as possible when introduced, it was designed without support for many features including the parametric polymorphism paradigm.

During the past several years, a large number of research projects have proposed to extend the Java programming language with parametric types. Some of them require modification of the Java Virtual Machine for support, but most attempt to avoid this. Two prominent proposals are GJ and NextGen, supporting a simple static mechanism and richer models of parametric polymorphism respectively.

The goal of this thesis was to explore the possibility of relying on an enhanced type erasure technique to achieve the run time parameterized classes, where a combination of homogeneous and heterogeneous translation is employed. We have developed an experimental Java compiler, based on GJ, supporting the run time parametric type information in Java. The Java Virtual Machine need not be changed to support this. To our knowledge, this has resulted in the first NextGen compiler implementation.

Keywords: Compilers, parametric polymorphism, parameterized types, Java, GJ, NextGen

Acknowledgements

I wish to first give my sincere appreciation to my supervisor, Dr. Stephen Watt, for his patience, encouragement and insightful guidance. He provided such a warm and friendly study environment and was always ready to lend a helping hand through out the investigation.

Thanks also goes to all the members of the ORCCA lab in the University of Western Ontario, for their friendship and support. I have greatly benefited from discussions, comments and encouragements from them.

Finally I would like to thank my husband Allan Z. Cheng, my daughter Amanda Cheng, my parents and my sisters for their love and understanding over the years.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and Thesis Outline

The word "polymorphism" in computer science can be explained as the ability to abstract code to work with different concrete types. For a programming language, there are generally two ways to achieve polymorphism [1]:

1. Parametric polymorphism, where polymorphism is accomplished through parameterizing components. For example, one can implement a List abstraction, standing for a list of various objects, with a parameter representing the types of objects contained in the list. Later, by specifying a concrete type for the parameter, the abstraction can be used as a generic module to provide some specific List types such as a List of Integers, or a List of Strings, etc.

2. Subtype polymorphism, which uses the top of a type hierarchy in place of variable types to simulate type parameterization. With subtype polymorphism, the programmers can

    - write code for objects of a type A
    - and have the code work for objects of type B where B is a subtype of A

    For example, the code for the Vehicle objects works for Car objects

Among the two approaches, parametric polymorphism is a widely adopted powerful form of generic programming. By abstracting the common behaviors from one or more types, it allows the programmers to write generic programs parameterized by the different types, which can later be instantiated variously according to different contexts to meet specific requirements. Through parametric polymorphism, programmers can greatly benefit from the enhanced flexibility, reusability, and expressive power of the programming environment. Recently parametric polymorphism has been recognized as an integral part of modern programming languages. Languages which lack such a mechanism are considered good targets for extension to support it.

Besides the above two common kinds of polymorphism, there is another new form of polymorphism called *correspondence polymorphism* for object-oriented languages. In correspondence polymorphism, polymorphism is accomplished by declaring a correspondence relation between methods. Therefore, the programmers need not write

in advance code to be used universally. For example, in the toy language LCP (Language with Correspondence Polymorphism) [23], $\texttt{T.m} \overset{\texttt{Corr}}{\longleftrightarrow} \texttt{S.n}$ means the method named as m relative to type T is similar to the method named n relative to type S. To our knowledge, the correspondence has not been adopted by any popular programming language.

The remainder of this thesis is organized as follows:

Chapter 2 covers the background material needed to understand the thesis. Chapter 2 starts out by introducing the concepts of unconstrained parametric polymorphism, bounded parametric polymorphism, and F-bounded parametric polymorphism. The bounded parametric polymorphism is essential in understanding the extensions to Java. Chapter 2 also discusses genericity in conventional Java. The subtype polymorphism used in conventional Java and its inadequacies are examined. The three approaches to implement generic classes in Java: virtual types, approaches with modifying the Java Virtual Machine, and translation are discussed in Chapter 2.

Chapter 3 gives a rather valuable solution Java parametric polymorphism, based on the NextGen proposal. This solution belongs to the type erasure approach and carries type information at run time. Within the enhanced type erasure model, each parameterized class is type-erased into a non-generic base class, and for each instantiation of the parameterized class, a light-weighted wrapper class and wrapper interface is generated. The wrapper classes and wrapper interfaces carry the run time

information for the parameterized class instantiation. The enhanced type erasure model is discussed in detail in Chapter 3.

In Chapter 4, the implementation details of our experimental NextGen compiler are discussed. Finally, Chapter 5 reviews the work of the thesis, and suggests some areas for the further work on this topic.

1.2 Java and Parametric Polymorphism

After its initial commercial release by Sun Microsystems in 1995, the Java programming language attracted many programmers. The original goal of the Java programming language design was to develop advanced software for consumer electronics. These devices are small, reliable, portable, distributed, real-time embedded systems. Java embodies several features that have allowed it to become a mainstream general purpose language with a widespread commercial acceptance. It is used not only to develop advanced software for a wide variety of network devices and embedded systems, but also large-scale software projects in various application areas. From Gosling and McGilton's [6] discussion, these features include:

- a simple and familiar syntax similar to that of C++, which makes it possible that the fundamental concepts of Java technology can be grasped quickly and programmers can migrate easily to Java and be productive promptly, because many programmers working at the time were using C and C++.

- an automatic memory management model, which completely removes the memory management load from the programmer. Thus, the Java language not only makes the programming task easier, it also dramatically reduces the number of bugs.

- support for a portable object-oriented programming model with safe program execution, making it possible to meet the challenges of application development in a wide variety settings, including network-wide distributed environments.

Today talk about Java seems to be everywhere. The currently defined Java language, even though it has gained some extensions, especially in the security area, remains the simple original version that has intended to be extended later [7]. The Java language has become a standard language for Internet programming and has been adopted by a large number of organizations. Nevertheless, it has some significant limitations from the perspective of software engineering, and recently it has been suggested that it is reaching the stage where some of these limitations should be eliminated by judicious language modifications and additions.

As we mentioned at the outset, parametric polymorphism mechanism is now recognized as an essential feature of modern programming languages, and has common use in mainstream applications. However, in order to keep the initial language as simple as possible, the current Java language did not provide support for

the parametric polymorphism paradigm [8]. Java does allow generic coding with subtype polymorphism, where generic types are simulated through the most general type of a type hierarchy (typically type Object) and the use of much down-casting and up-casting. As an example, in Java, a data structure such as a list of objects can be written by writing a `List` class to hold objects of a common super-type of all the objects that the programmers may want to store in the list. Typically, this is a list of Objects. When the data structure is used to hold objects of a specific type, type casts need to be done when retrieving objects from the list. This is due to the fact that the data structure does not store the types of the objects. Unfortunately, the required casts are tedious to write and error-prone. Furthermore, the type casts largely defeat the error-detection properties of Java's static typing discipline. Lack of a parametric polymorphism mechanism is considered to be a serious obstruction in implementing substantial programs in the Java programming language. We shall discuss the genericity of the conventional Java language in detail in Chapter 2.

Over the past several years, a number of research projects have attempted to extend the Java programming language with parameterized types. Some of these projects also suggest to integrate "virtual types" into Java. Virtual typing is a programming language mechanism in which a base class definition can be augmented with virtual type declarations. Class specializations narrow those virtual type members [10]. This way the functionality of the parameterized classes can be achieved. The details of the virtual type proposals remain to be worked out. Virtual types are discussed in section 2.3.

In one of the first proposals for parametric polymorphism, Myers, Bank, and Liskov [13] have designed and implemented an extension of Java called PolyJ. That extension implements constrained genericity through parameterized classes. A modification of the Java Virtual Machine was required in this proposal. In their paper [15], Agesen, Freund, and Mitchell have proposed a suggestion to extend Java with parametric types, where an extension to the byte code format and a revision of the class loader are assumed. However, the modification of the Java Virtual Machine is a considerable challenge to Java's machine-independence feature. At present, extending Java with modifications to the Java Virtual Machine is still considered to be an unjustified direction.

Other very interesting research projects are from Odersky and Wadler. They designed and implemented an extension to Java with an explicit goal: to be very conservative so that no changes to the existing Java Virtual Machine are required. Their earlier work Pizza [5] supports F-bounded parametric polymorphism (see chapter 2), as well as higher-order functions and algebraic data types. Later they collaborated with Gilad Bracha and David Stoutamire from JavaSoft and produced GJ [4] which is a successor of Pizza. Both Pizza and GJ were implemented by translation into current Java, and both the Pizza and GJ compilers can also be used as Java compilers.

1.3 GJ

Among these projects outlined in the above section, only GJ is designed to be fully backwards compatible with the conventional Java. This is useful for a software developer to evolve smoothly from the non-parametric Java programming to the parametric Java programming. In May 2001, Sun released a prototype for adding the parametric types to Java based on GJ.

GJ is translated into the conventional Java with a type erasure technique, where all type parameters are erased, all the type variables are replaced by their upper bounding types (typically Object), and suitable casts are added. We give a more detailed discussion on the type erasure technique in Chapter 2. A given GJ parameterized class is translated into an ordinary Java class which is similar to what one would write if the parametric polymorphism were not available in Java and the subtype polymorphism were used (i.e. the Java generic coding idiom with casts). In this implementation, all the instantiations of the parameterized class share the single type-erased class at runtime. The main problem with GJ is run time type information about the parametric types cannot be correctly maintained. Consequently, the type dependent primitive operations such as:

```
new  A(…)
new  A[…]
instanceof  GeneralCollect<X>
```

where A is a type parameter and X represents a specific type such as Integer, are not allowed in GJ. We say GJ lacks the integration between the parametric types and the conventional Java type system.

1.4 Objectives

The main object of this thesis is to create an extended Java compiler that supports run time parametric types. Through our experimental compiler application, we are trying to determine whether it is feasible to rely on the type erasure technique to achieve the run time parameterized classes is. The specification of our experimental compiler is based on the NextGen proposal [20]. NextGen is an extension of Java intended to address the problems of GJ described above. In NextGen, the objects of parametric types need to carry their type information at run time. This is impossible simply using the type erasure mechanism used in GJ. An enhanced type erasure translation model has been introduced for NextGen and this requires generation of stubs for each instance of the parameterized class. The details of the translation technique are discussed in Chapter 3.

Chapter 2

Background and Related Work

2.1 Notions of Parametric Polymorphism

2.1.1 Unbounded Parametric Polymorphism

In some languages, such as C++, there is no way to explicitly specify the constraints a type must have in order to be used in place of a type parameter. This feature is referred to as unbounded parametric polymorphism (unconstrained genericity). In Figure 2-1, a parametric class GeneralCollection is defined with one formal type parameter T, which represents any type (Universal type).

In the unbounded form of parametric polymorphism, any type may be provided as an actual parameter and therefore no special properties can be assumed about the operations of the type parameter. This implies that only the operations that apply to all

types can be allowed in the unbounded parametric class. In C++, this simple form of parametric polymorphism is supported through templates. When the template instantiation is done during compile or link time, the compiler or the linker can check if an actual parameter provides the required operations or not. This effectively means that generic programs based on C++ templates cannot be checked at compile time.

```
class GeneralCollection <T>{
        …
        public void addElement ( T elem){
           //… statements of the method body
        }
        …
}
```

Figure 2-1 A example of unbounded parametric class definition

2.1.2 Bounded Parametric Polymorphism

In many cases parametric polymorphism with constraints on the type parameters is more appropriate. For an example, we consider a parameterized class `GeneralCollection<T>` with a method `min()`, which compares the values of the elements within a "GeneralCollection" and returns the smallest one. Thus, the

parameterized class can only work properly with types T that provide a comparison operation. Here, we need to explicitly restrict the allowed actual type to types having some specific properties. This feature is referred to as bounded parametric polymorphism (constrained genericity) [3]. There have been two different forms suggested to provide constraints on type parameters when extending Java. The first form [5] [18] can be illustrated in Figure 2-2. Here the class `GeneralCollection` has a formal type parameter T whose upper bound is Smaller and the interface Smaller is defined elsewhere.

```
public  interface  Smaller {
    …
    public  boolean lessthan  (Object operand);
}

public class GeneralCollection<T implements Smaller> {
     …
    public T  min( ) {
         …
        //a statement calling the method lessthan;
         …
 }
```

Figure 2-2 A bounded parametric class specifying constraints with an interface

In the other form, all the required operations must be listed in the appropriate position of the defined class [3]. A corresponding example of the above would be

```
ClassGeneralCollection [T]
    where T{boolean lessthan(T  operand ) }
 {
      //...
 }
```

Here the required operation "lessthan" is listed within the class's signature.

2.1.3 F-bounded Parametric Polymorphism

In Figure 2-2, the method "lessthan" takes a value of type Object as an argument, which is inexact and leads to efficiency problems and errors. It is often convenient to allow the bounding type itself to take generic parameters. Thus, the upper bound type of a type parameter can be specified by a recursively bounded type limitation. This sophisticated form of bounded parametric polymorphism is referred as F-bounded parametric polymorphism. As shown in Figure 2-3, the bounded parametric class

GeneralCollection takes one formal type parameter whose upper bound types are also expressed as a bounded parametric interface.

```
public interface Smaller<T>
{
    ...
    public boolean lessthan ( T  operand);
}

public class GeneralCollection <T implements Smaller<T>>
{
   //implement the behaviors for the GeneralCollection
}
```

Figure 2-3 F-bounded polymorphism

In both bounded and F-bounded parametric polymorphism, an actual type parameter can be accepted only if it is explicitly declared to extend or implement the parameter bound.

2.2 Genericity in Java

As described in Chapter 1, abstracting from concrete types (i.e. polymorphism or genericity) can be achieved using parametric or subtype polymorphism. Java is designed to support subtypes directly, where class Object is the root of the class hierarchy and therefore all the reference types in Java are subtypes of type Object. In Java, a new type can be defined by either a class or by an interface. A class can extend from only one other class, but can implement many interfaces. An interface can be implemented by more than one class. As determined by the these features, parametric polymorphism may be simulated in Java by using either the universal reference type Object or an abstract type (i.e. an interface) in place of the type parameters and explicitly casting values of type Object into their intended types.

As an example, consider a generic class Pair which has two instance variables and a method min(), and compares the values of the two instance variables and returns the smaller one. A possible Java code is in Figure 2-4. A corresponding client class is illustrated in Figure 2-5.

In Figure 2-4, the required types that should work properly with Pair are defined by interface PairNode. This implies that in any class who implements PairNode (subtype of PairNode) is suitable for to use in the Pair class. Note that the definition of the constructor Pair always treats the objects passed in as the parameters as if they have the type PairNode, and the operation min() always treats the returned object as if

it is of the type PairNode. Yet the code in Figure 2-5 takes the type of MyType as the actual parameter type when using the Pair constructor. In Figure 2-5, because the return type of min( ) is always PairNode, if we need to perform some operations on the result of min(), we must cast it to its original type MyType first.

```
public  interface PairNode{
        public boolean  lt(Object operand);
}
public class Pair{
     PairNode  first_p;  PairNode  second_p;
     public Pair (PairNode p1,  PairNode p2)  {
        if ( p1.getClass()  !=  p2.getClass())
           throw new IlleagalArgumentException(
           "The two nodes of the Pair must be of the same type !");
        first_p = p1;
        second_p = p2;
     }
     public PairNode min ( ) {
        return first_p.lt(second_p) ? first_p : second_p;
     }
}
```

Figure 2-4 A generic class Pair implemented with an interface

```
public class MyType implements PairNode {
        int    MyTypeValue;
        public MyType( int  v1) { MyTypeValue = v1; }
        …
        public boolean lt(Object  o)throws illegalArgumentException
        {
           if (this.getClass() != O.getClass() )
              throw new illegalArgumentException(
                        "the type of the operand does not match");
           return this.MyTypeValue < ((MyType)o).getValue();
        public int  getValue( ){  return MyTypeValue; }
}

public class PairTest  {
      public static void main (String[] args)  {
            MyType   m1 = new MyType(1);
            MyType   m2 = new MyType(2);
            Pair  p =  new  Pair( m1, m2);
            System.out.println(" The samller one is  "
                    +((MyType)p.min( )).getValue( );
             }
  }
```

Figure 2-5 A client class of the Pair class in Figure 2-4

The most frequent example of polymorphism is the implementation of collection data types, whose behavior and features are independent from the type of the elements

of the collection. Collection classes, such as lists, sets, vectors, and so on, are fundamental data structures in a programming language which need to work with different element types. The common solution with the Java programming language is to declare a collection class with Object as its element's type. Thus the elements in the collection may have any reference type. However, as a programmer, when programming with such a collection class, one is forced to do extra work: one must keep track of the actual type one is working with in the collection. Whenever one extracts an element from the collection and does further processing on it, one must manually cast it back to its original type. The mechanism, using the universal reference type Object as the element type of collections to simulate type parameterization, is referred to as the Java generic coding idiom [20].

On the whole, the Java generic coding idiom has the following consequences for the Java programs:

1. Program coding is clumsy and error-prone because the required casts are tedious to write.

2. Down-casting and up-casting largely defeat the error detection properties of static type checking rules. For example, when programming with a Vector data structure, an Integer can be inserted into it first, and may be subsequently extracted and down-cast to an String object, this error will not be caught until run time.

3. The required casts give an extra a run-time cost. To illustrate this, consider the following: In the Java API Vector class the insertion operations such as "add" and "addElement" take an argument of type `Object`. No matter what an element's type is, the element will be up-casted to type Object to be inserted. The element's original type is completely lost. When an element is extracted from a vector at a later time, it is always treated as of type Object. Therefore, the extracted element must be explicitly down-cast to its original type before further processing. On the other hand, if the Java API Vector class were implemented as a parametric class which is parameterized by the vector's element type, the element type information would be reflected in the insertion and extraction operations, so no up-casting and down-casting would be needed.

Because of the problems with subtype polymorphism previously outlined, in the past few years there has been much done on extensions of Java for parametric polymorphism. In the following sections, we view of these research results.

Generally, the works for parametric polymorphism extensions of Java can be divided into two groups:

1. Approaches requiring extension of the Java Virtual Machine (JVM) or of parts of it.

2. Translation approaches without changing the JVM.

In the former case, compatibility with the existing Java Virtual Machine is lost, but this may lead to more flexibility and power. The second approach does not have any effect on the existing Java standard platforms but may introduce a significant overhead in both space and time.

In addition to the above solutions, virtual types have also been suggested as a way to implement parametric types. In this case, a generic class contains one or more type members that are virtual in the sense of a C++ virtual member; the derived classes are obtained by extending the base class in the usual way, and the extensions redefine the virtual type members to their specialized values.

In the remainder of this chapter, we will discuss these solutions in more detail. We shall concentrate on the implementation of translation approach, which we are more interested in.

2.3 Virtual Types for Java

Virtual types are a programming language mechanism which have evolved from the Beta programming language, and have been suggested by Thorup [10] as a means to provide the functionality of parameterized classes in the Java programming language. Virtual types are quite different from parametric types because they have no

parameters. However, virtual types and parametric types are both proposed as extensions to Java to address similar issues. When we discuss parametric polymorphism for Java, we should also describe some of the research work that is related to the extension of Java with virtual types.

In the extended language of [10], the definitions of classes and interfaces are made generic with virtual type declarations, Each declaration introduces a new type name that is associated with some existing type. In their subclasses, a virtual type may be extended to have subtype of the type that it had originally.

With virtual types, genericity is still accomplished via subtyping. As an example, consider a generic class Pair which has two instance variables, with methods for setting and getting the components of the Pair. The corresponding Java code is in Figure 2-6.

Figure 2-7 gives a roughly equivalent program using virtual types. The Pair class declares a virtual type named Ptype. Then the subclass PointPair of Pair extends the virtual type Ptype to be the type Point rather than Object. The class PointPair defines a Pair, within which the two instance variables hold the instances of type Point or its subtypes. Thus, the conventional Java code of the Pair test class `pairTest` in Figure 2-6 is implemented with virtual types in Figure 2-7, where the explicit downcastings are removed from the program text.

```
class Pair {
    Object  p_first;
    Object  p_second;
    public  void  set_first  (Object   x)  {   p_first  =  x;   }
    public  Object  get_first ( )  {  return  p_first; }
    public  void  set_second (Object  y) {p_second=y;  }
    public Object get_second ( )  {return  p_second; }
     …
}

 public class PairTest  {
     public static void main (String[] args)  {
         Pair  p =  new  Pair;
         p.set_first ( new Point (0, 0) );
         p.set_second (new  Point (2, 2);
         …
         Point   p1  =  (Point) p.get_first( );
         Point   p2  =  (Point) p.get_second( );
          …
      }
 }
```

Figure 2-6 Definition of class Pair in conventional Java

```
class  Pair {
    typedef  Ptype as Object;
    Ptype   p_first;
    Ptype   p_second;
    public void set_first (Ptype  x) {  p_first = x;  }
    public  Ptype get_first ( )  {   return  p_first; }
    public void set_second (Ptype   y){p_second =  y;  }
    public Ptype get_second ( ) {return  p_second; }
    …
}

public class PointPair extends Pair {
        typedef  Ptype as Point;
 }
```

Figure 2-7 Definition of class Pair in extended Java using virtual types

```
public class PairTest  {
      public static void main (String[] args)  {
            PointPair  p =  new  PointPair( );
            p.set_first ( new Point (0, 0) );
            p.set_second (new  Point (2, 2);
               …
            Point   p1  =  p.get_first( );
            Point   p2  =  p.get_second( );
              …
        }
}
```

Figure 2-8 A test class for the Pair class using virtual types

Generally speaking, in integrating virtual types into Java, a generic class contains one or more type members that are virtual in the sense of a C++ virtual member. The derived classes are obtained by extending the class and narrowing those virtual type members. Thus, the functionality of parameterized classes is achieved. However, with virtual types the genericity is still accomplished via subtyping. The relations between types are required to be carefully set in advance. For families of related types, the virtual types are especially useful. But, for a collection class that is generic across types that have no real family relationship, parametric types are more useful. Furthermore, the details of the virtual type proposals remain to be fully implemented. Therefore, we have taken more seriously extending Java to support parametric types.

A quite complete comparison of the relative strengths of parametric and virtual types has been made in [11]. That article also suggested that it is possible to merge virtual and parametric types, but the design and implementation complexity increased substantially. The ultimate solution of this combination remains to be worked out [11].

The remainder of this chapter will focus on the discussions of parameterized types for Java.

2.4 A Extension of Java Modifying JVM

If we had the possibility to completely redesign the JVM, we would have plenty of freedom in the implementation of parameterized types. For example, we could modify the run time representation of classes (the class "Class") and let it support parameterized classes just as JVM supports arrays. In reality, it is important that the extensions to the Java language be implemented without requiring a replacement of all the current JVM installations. Nevertheless, over the past several years, some research groups extended Java by relying on an enhanced JVM. A typical product of this approach is PolyJ.

PolyJ was designed and implemented by Myers, Bank, and Liskov at MIT to support parameterized classes, and relies on a modification of the Java Virtual Machine. PolyJ supports constrained parametric polymorphism, where a new keyword

where is introduced in to specify the bounds of the type parameters which will be instantiated by the actual types. This is very similar to Liskov's previous endeavor, CLU. For example, Figure 2-9 is a PolyJ code for the Pair class, where the **where** clause specifies that the type represented by the type parameter T must implement a method with the signature: boolean gt (T t). Otherwise, there will be a compile-time error.

The designers of PolyJ recommend some extensions to the bytecodes of the Java Virtual Machine to support parameterized class files, as well as some corresponding effects of these extensions on both the bytecode verifier and interpreter. The format of a .class file is specified in the JVM specification [14]. PolyJ modifies the JVM by adding two opcodes: ***invokewhere*** and ***invokestaticwhere*** which support invocation of the methods that correspond to the **where** clause. PolyJ also made changes to the format of the `.class` file by adding information to be supplied to the bytecode verifier to directly verify the parameterized code. PolyJ also provided an extended bytecode interpreter which is designed to duplicate very little information when the generic class is instantiated.

```
public class  Pair[T]  where T { boolean gt ( T t); }
{
    private  T   p_first;
    private T  p_second;

    public Pair(T t1,T t2) {p_first=t1; p_second=t2; }
    public void set_first (T x) {p_first = x;}
    public  T get_first ( ){ return  p_first; }
    public  void  set_second (T   y) { p_second =  y;}
    public  T  get_second ( )  {return  p_second;}
    public  T   max( ) {
          return p_first.gt (p_second)?
                p_first : p_second;
     }
     …
}
```

Figure 2-9 A implementation of Pair class in PolyJ

Another frequently mentioned research project was by Agesen, Freund, and Mitchell [15]. They designed and implemented an extension to the Java language based on the idea that the instantiation of parameterized classes can be delayed until

load time. In their work the JVM is modified by a revision of the class loader, and the `.class` file format is extended to support parametric classes. Here, the parameterized classes have the following general form:

```
class C   <parameters>
```

where *parameters* represent a list of type variables.

It appears to be a common opinion that extensions to the JVM can express parameterization more directly. The parametric polymorphism solutions based on an extended JVM should therefore be expected to be more elegant. Nevertheless, modification to the JVM is a considerable challenge to Java's machine-independence. Because Java has an established user base, it is essential that any changes to the language specification are devised with mature reflections, so that there is no doubt as to the functionality of the extension, and all other relevant parts of the Java system are able to modify their products relatively easily. If the Java byte code is to be changed, many compatibility issues come into existence. A separate issue is that the Java platform was designed to run programs securely on networks. This means that the Java platform integrates safely with the existing systems on the network. In the base Java security model the three parts of Java's security defense include the Byte-code Verifier, the Applet Class Loader, and the Java Security Manager [17]. Whenever we are talking about JVM extension, it is important to consider the effect on the Java security model whether the extensions are for the byte code verifier or for the class loader. For wide deployment, these kinds of modifications to the JVM are still

considered to be an unjustified risk. On the other hand, the solutions which translate the extended Java to the existing JVM bytecode have obtained more acceptance.

2.5 Translation Without JVM Extensions

As discussed above, the extensions to conventional Java should preferentially be implemented in a manner which does not require JVM extensions. This is the major reason for the methods of the translation approach.

Initially investigators in the Pizza project intended to translate the extended Java directly into the Java Virtual Machine [5]. They made such a decision based on the reality that the JVM is available across a wide variety of platforms, including different kinds of computers, consumer electronics, and other devices. Furthermore, it was important that the existing code compiled from conventional Java could smoothly inter-operate with new code compiled from extended Java. This would let the extended Java programms access the extensive Java libraries for graphics, networking, etc. However, as research progressed, it became apparent that "the JVM and Java were tightly coupled, and translating the extended Java into conventional Java as an intermediate stage would gain much in clarity" [5].

The translation approach is used in Pizza, GJ and NextGen implementation, where the semantics of the extended Java is given by a translation into the conventional Java code.

2.5.1 Homogeneous Translation and Heterogeneous Translation

A parameterized definition is used by providing actual types for each of its parameters. We call this an instantiation. Basically there are two forms of implementation techniques for parametric polymorphism: homogeneous and heterogeneous translation. These techniques are based on whether a single or many generated codes, respectively, for the multiple instantiations of a parameterized class.

Heterogeneous Translation

A straightforward heterogeneous translation maps a parameterized class into a separate specialized class whenever a new instantiation is encountered. This technique is generally used for C++ template implementation and has been discussed in [5] for designing and implementing the Pizza language.

Some of the implementation details of heterogeneous translation are:

1. Because the generic class files are never loaded, the class files produced through compiling the generic classes do not have to be valid.

2. Every time a reference to a new instantiation of a parameterized class is encountered, the compiler produces an additional class file with actual

type parameters instead of formal types. The instantiated files should be interpreted by the virtual machine. Therefore, it must be valid.

The main deficiency of this implementation method is that redundant compilations are produced. A considerable number of nearly identical files with strange names might be created by the compiler. The heterogeneous translation may incur a substantial increase in code size. However, heterogeneous translation can provide correct run time information about instantiated classes.

Homogeneous Translation

Homogeneous translation uses a single copy of code with a universal representation, namely the homogeneous translation maps a parameterized class into a single class to represent all its instantiations. A homogeneous translation has been implemented in Pizza and GJ, a superset of the Java Language that supports bounded and even F-bounded parametric polymorphism.

Some of the implementation details of the heterogeneous translation are:

1. When compiling a parameterized class, the formal type parameters are assumed to be their corresponding bound types. Therefore, a parameterized class is represented by simply replace the formal type parameters with their bound types respectively.

2. When a reference to an instantiated class is encountered, the compiler must interpret the generic class file that corresponds to the generic class. The compiler interprets each formal type parameter as its corresponding actual type. All the instantiated classes are represented by the same generic class produced from their corresponding parameterized class.

Homogeneous translation yields much more compact code than heterogeneous translation, by contrast, homogeneous translation leads to incorrect run time information about instantiated classes and parameter types.

Mixtures of heterogeneous and homogeneous translation are possible. In our thesis work, the NextGen proposal gives such a solution intended to make a trade-off between size and speed, by creating one type erased base class for each parametric class and one light wrapper class and one interface for each instantiated class.

2.5.2 Type Erasure Technique For Generic Java To Java Translating

The type erasure technique is a straightforward method of translation, through which a parametric class, for example List<T>, is translated into the class a Java programmer

would write to implement List with the Java generic coding idiom. During translation, type erasure replaces type variables by their upper bounding type (typically Object). In this manner, all the instantiations of a given parametric class such as List<T> are converted to a single corresponding base class List<Object>.

The type erasure technique was developed by Odersky and Wadler for Pizza [5] and GJ [18]. These are two of the most popular generic Java compilers, where GJ is a successor of Pizza. Because the design and implementation of GJ is based on the previous work of Pizza, GJ and Pizza are similar in several aspects:

- They are both supersets of the conventional Java with provision of parametric polymorphism mechanism.

- Both of them are designed to be fully backward compatible with the conventional Java programming language, in that every legal conventional Java program is still legal in the extended language.
- Both Pizza and GJ are implemented with the homogeneous translation technique. They are both explained and implemented by translation into the conventional Java programming language.

Nevertheless, there are a number of differences between the two:

- In addition to generic types, Pizza introduces high-order functions and algebraic data types with pattern matching. While GJ adds only one important idea —type parameterization to Java.

- When dealing with the legacy library code, Pizza needed to rewrite all legacy code to use a parametric version. But in GJ, a parametric type such as List<T> may be passed wherever the corresponding raw type List is expected.

- Parametric types in GJ only apply to reference types. A base type such as int cannot be used as a type parameter. In Pizza, both base types and reference type are legal as type parameters

A Pizza compiler has been implemented by adding Pizza features to EspressGrinder [5] which is a compiler for Java written in Java. A compiler implementing GJ has been written in GJ, and can also be used as a Java compiler. Our NextGen compiler is a modification of the GJ compiler, based on the NextGen proposal [20] and is implemented in GJ.

CHAPTER 3

A Solution for Dynamic Parametric Polymorphic Classes For Java –NextGen

Pizza [5] and its successor GJ [18] have shown that introducing parametric classes to Java using the type erasure technique is an interesting approach. The main problem of extending Java to GJ [18] is that the run time information of the instantiated parametric types is not be available through homogeneous translation. As mentioned earlier, GJ cannot support type-dependent operations such as object and array creation operations over type variables, or casts and instance tests operations over parametric types. For example, the following operations are illegal in GJ, where T is a type parameter.

(1) new T[]

(2) new T(...)

(3) instanceof CollectionClass<X>, each for specific X such as Integer

This situation prevents the desirable integration of the programming mechanisms introduced with the core language.

In this thesis, a NextGen compiler was built based on the discussion of an extension [20] to the GJ proposal. This uses the type erasure technique with additional run time information about the instantiations of the type parameters. In this chapter we will give a description of the design issues underlying NextGen.

3.1 How NextGen Is Intended to Work

In general terms, NextGen employs a combination of homogeneous and heterogeneous translation so that run time information about the type parameters is captured for use with a primarily homogeneous translation. Homogeneous translation avoids the creation of a large number of redundant compilations arising from the instantiation with type parameters. At the same time, the run time information about the instantiated types can only be carried or be known through the heterogeneous translation. Therefore NextGen intends to use a combined approach to address the integration issues of GJ and keep the code growth to an acceptable level.

Roughly speaking, the NextGen was designed in tandem with GJ. Just as the conventional Java programming language is a subset of GJ, GJ is a subset of the NextGen. But NextGen differs from GJ in three respects:

1. Parameterized types and type variables can be used wherever a conventional Java type can be used. In particular, NextGen can implement the operations such as:

   new T(...)

   new T[n]

   By allocating a new object or a new array with the correct run time type information, NextGen avoids the severe restrictions placed on this construct in GJ. Furthermore, NextGen can implement casting operations or instance test operations on parametric types such as GeneralCollection<T> .

2. NextGen is always type-sound when involving parametric type declaration. In contrast, GJ is only type-sound with some constraints. GJ will always generate an unchecked warning whenever it encounters the operation `new T[...]` (where T is a type parameter). In GJ, such operations cannot be executed even though they seem to be type-correct and pass compilation, since the information of the type parameters is not available at run time.

3. GJ implements all the instantiations of a given parametric class using a single run time class representation. But NextGen implements the instantiations of a given parametric class in a more complex way, with one erased base class for each parametric class. One light wrapper class and

one interface for each instantiation are used. The wrapper class inherits the type-erased methods from the abstract type-erased base class. The run time information for each parametric class instantiation is carried through the wrapper class. In NextGen, some new class files are generated whenever a new instantiation of a parameterized class is found.

3.2 The Design and Implementation of NextGen

3.2.1 Parameterized Type Declaration

A parameterized type declaration (class or interface) defines a set of types, one for each possible instantiation of the type parameter. Loosely based on the syntax of templates in C++, a parameterized type in NextGen consists of a class or interface type C and a parameter section $<T_1, \ldots, T_n>$. C is the name of the parametric class or interface, $<T_1, \ldots, T_n>$ is a list of type variable declarations delimited by < > brackets. Each type variable declaration has an optional class or interface type as an upper bound, and each actual type variable must be a subtype of the bound type. If the bound clause is omitted in a type variable declaration, the type java.lang.Object is assumed as the default bound. The scope of a type variable is all the program text of the class or interface declaration.

Figure 3-1 is a simple example of a parameterized class definition in NextGen. Note: the type-dependent operation new T[n] is illegal in GJ but legal in NextGen.

```
Public class NGVector<T> {
  Private  T[]  eles;
  Public NGVector ( int  initCapacity) {
      eles   =  new  T[initCapacity];
  }
  public T  elementAt (int  index) {
       return  eles[index];
  }
  public void  setElementAt(T newValue, int index) {
        eles[index]  =   newValue;
}
```

Figure 3-1 A parameterized class NGVector definition

In NextGen, an instantiation of a parametric class or interface can be used anywhere a conventional unparametric class is used. The type variables can also be used anywhere that a simple unparametric class is used except as the direct supertype in an "extends" or "implements" clause of a class or interface declaration.

3.2.2 Translation Maintaining Run Time Type Information

Like Pizza and GJ, NextGen use type erasure in implementing parametric types. All occurrences of type variables in a program text are replaced by their bounding type (typically Object). NextGen augments the type erasure model used in Pizza and GJ by using wrapper classes and interfaces to provide run time type information. Therefore, compared with GJ, the translation in NextGen is more complex.

We can roughly describe the translation process of NextGen as replacing all types by their corresponding erasures. The erasure of a non-parametric type is the type itself (so the erasure of String is String). The erasure of a type parameter is the erasure of its bound (in parametric class List<T> definition, the erasure of T is Object). In these cases, the translations in NextGen and GJ are similar. However, the erasure of a parametric type is quite different between GJ and NextGen. In GJ it is translated into conventional Java by type erasure, where all instantiations of parametric classes are translated to their corresponding base class. NextGen, on the other hand, supports run-time-dependent operations, i.e. all objects of parametric type carry their run time type information, so the parametric type translations of NextGen involve more diversified forms. We therefore need to consider the following three situations in parametric type translation [18]:

1. Converting each "new" operation involving a parametric type to a "new" operation for the corresponding wrapper class. Thus,

```
new   Stack<Number>
```

is translated to

```
new   $$Stack$_Number_$
```

and

```
new   Vector<Boolean>[100]
```

is translated to

```
new $$Vector$_Boolean_$[100].
```

2. Converting each casting operation to the parametric type to two casts sequentially: a cast to the corresponding wrapper interface, followed by a cast to the corresponding base class. Thus,

```
(Vector<Integer>) x
```

is translated to

```
(Vector) (($Vector$_Integer_$) x)
```

3. Converting each `instanceof` test involving a parametric type to an `instanceof` test on the corresponding wrapper interface. Thus,

```
instanceof  Vector<Integer>
```

is converted to

```
instanceof  $Vector$_Integer_$
```

We now discuss the GextGen translation from two points of view:

(1) The translation of a parametric classes

(2) The translation of the client classes of the parametric classes

For a Given parametric class $C<T_0,...T_n>$ in a package P, the NextGen compiler generates a type-erased base class C in the same package. This base class C extends the base class of the parametric super-type of $C<T_0,...T_n>$. All the attributes for each member of $C<T_0,...T_n>$ are preserved in its corresponding base class. In order to support the run time type depended operations, NextGen uses a slightly different process than GJ to construct the base class. If the run time-dependent operations do not occur in a parametric class, the NextGen compiler constructs the base class in the same way the GJ compiler translates GJ into the conventional Java language. When a run time depended operation such as

new T(...)

or new T[n]

is encountered in a method body, such an operation is replaced by a call to a generated auxiliary method. In the base class, all the generated auxiliary methods (called a "snippet") are abstract and protected with a mangled name. These auxiliary methods can be appropriately overridden in each wrapper class according the specific actual type parameter to perform the appropriate type-dependent operation.

The parameterized class NGVector<T> in Figure 3-1 is translated by the NextGen compiler into a class file for the base class shown in Figure 3-2.

```
public abstract class NGVector {
     private  Object[]  eles;
     public NGVector ( int  initCapacity) {
          eles   =  $snip$1 (initCapacity);
     }
     public Object  elementAt (int  index) {
          return  eles[index];
     }
     public void setElementAt(Object newValue, int index){
           eles[index]  =   newValue;
     }
     abstract protected  Object[]( int  initCapacity);
}
```

Figure 3-2 The type-erased base class for the parametric class NGVector in Figure 3-1

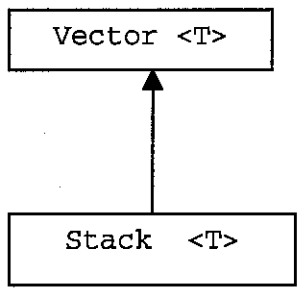
For each instantiation of a parameterized class used in a compilation unit, the NextGen compiler generates both a lightweight wrapper class and an empty interface

to carry the run time type parameter information. The reason for this is that Java does not support multiple inheritance, i.e. a Java class can be extended from only one super class. When one parameterized class such as `Stack<T>` extends another parameterized class such as `Vector<T>`, there are two super types for each instantiation of `Stack<T>` like `Stack<A>`. The two super types of Stack<A> are:

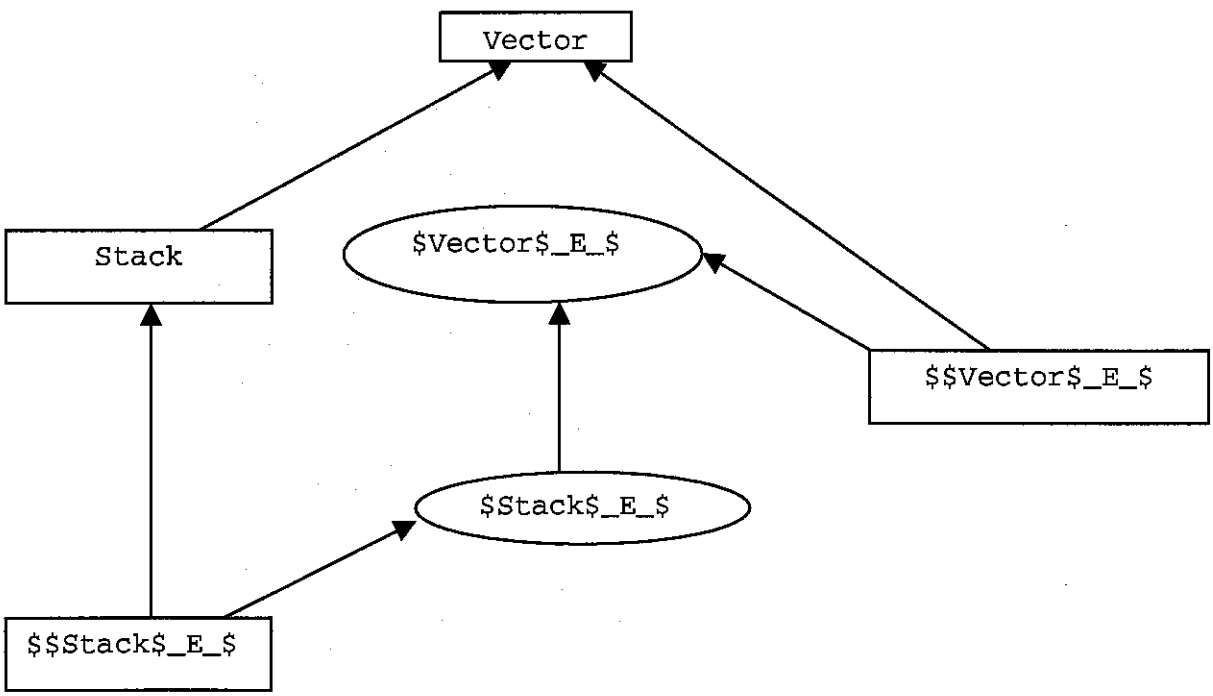
1. its corresponding base class Stack, from which it inherits nearly all of its members

2. the corresponding instantiation Vector<A> of parameterized class Vector<T>

In NextGen the type of the wrapper class can be represented by its corresponding interface. Therefore, a wrapper class `$$Stack$_E_$` can extend its base class `Stack` and implement its own interface `$Stack$_E_$`. At the same time, `$Stack$_E_$` implements the corresponding interface `$Vector$_E_$` for the super wrapper class `$$Vector$_E_$`. The resulting type hierarchy produced by the NextGen for this example is shown in Figure 3-3. In Figure 3-3, boxes representing classes and ovals representing interfaces.



(a) NextGen Source



(a) JVM Representation

Figure 3-3 A Simple Parametric Type Hierarchy and Its JVM Class Representation

The wrapper class inherits all the type-erased methods from the corresponding base class that is constructed by erasure from the parameterized class definition using type erasure. Further, if the run time type-dependent operations such as *new* `T[]` do not occur in the parametric class, the wrapper class simply duplicates the base class constructors by invoking the constructor call to the base class (using a superclass constructor call *super).* Otherwise, if the parametric class contains type-dependent operations, NextGen augments the type erasure model with additional snippets. Thus, the corresponding wrapper class not only replicates base class constructors by forwarding constructor calls to the base class but also provides the implementations for the abstract snippets methods in the base class. These additional snippet methods' definitions select the appropriate snippet code to meet the requirements of the run time type-dependent operations.

For example, for `NGVector<Integer>`, an instantiation of the parametric class `NGVector<T>` shown in Figure 3-1, NextGen generates an empty wrapper interface `$NGVector$_Integer_$` and a wrapper class `$$NGVector$_Integer_$` shown in Figure 3-4.

Figure 3-3 and Figure 3-4 show that the translation of the constructor in the `NGVector<T>` base class is postponed because the constructor invokes an operation *new T[]* that dependents on the run time type information of the type parameter T. The specification of the actual allocation operation is served as a "dynamic" method call made in the corresponding wrapper class.

```
public interface $NGVector$_Integer_$ {    }

public class $$NGVector$_Integer_$ extends NGVector
       implements $NGVector$_Integer_$
{
     public $$NGVector$_Integer_$ (int  initCapacity) {
       super (initCapacity);
     }
     public protected Object[] $snip$1((int  initCapacity)
     {
       return new  Integer[initCapacity];
     }
}
```

Figure 3-4 the wrapper interface and wrapper class for NGVector<Integer> which is an instantiation of the parametric class NGVector<T> of Figure 3-1

CHAPTER 4

Our Implementation of a NextGen Compiler

4.1 Summary of GJ Compiler

The GJ compiler has been implemented in GJ itself by Wadler et.al, and is publicly available from a number of web sites. In May 2001, Sun has put forward a proposal to add generic types to the Java programming language, and also released a prototype on Wadler's GJ implementation. Our NextGen compiler is derived from this prototype. We therefore also refer to Sun's prototype as the GJ compiler.

Roughly speaking, the GJ compiler is constructed as a series of passes over an abstract syntax tree. As illustrated in Figure 4-1, the scanner maps a source program, which is an input stream consisting of Unicode characters [12], into a token sequence. The parser then maps the token sequence into an abstract syntax tree. Subsequently,

several traversals over the syntax tree nodes are performed to evaluate the semantic rules and finally map flat Java to bytecodes.

In the GJ compiler, everything in one source file is kept in one `TopLevel` structure. The syntax tree of the compiler is represented by the recursive constructs of the syntactic subcomponet tree nodes.

In the GJ compiler, there is an abstract root class `Tree` for all the abstract syntax tree nodes. The GJ compiler provides definitions for each of GJ's syntactic constructs, such as **`if`** statements or **`for`** loops, as subclasses nested inside the root class `Tree`. A class hierarchy for representing the abstract syntax tree is shown in Figure 4-2.

The root class `Tree` itself defines fields for the tree's type and position. The subclasses used to represent individual GJ syntactic constructs are highly standardized. Each subclass typically contains only tree fields for the syntactic subcomponents of the node. For example, as shown in Figure 4-3, class `ClassDef` represents a class definition node. All variables and methods defined in this class are embodied in the field: `List<Tree> defs`.

A variable definition is represented by a tree node of `Vardef` and a method definition is represented by a `MethodDef` tree node.

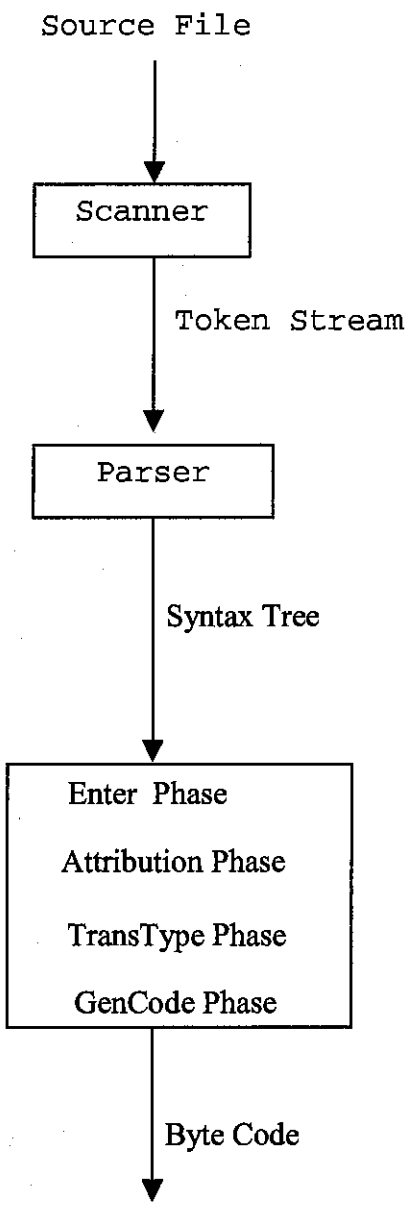


Figure 4-1 Outline of the GJ Compiler

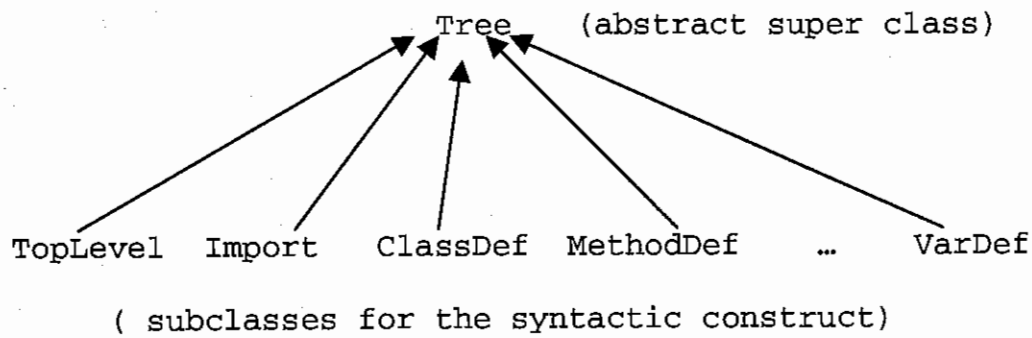


Figure 4-2 Implementation of the Abstract Syntax Tree

```
public  class ClassDef extends Tree {
     public  int                      flags;
     public  Name                     name;
     public  List<TypeParameter>      typarams;
     public  Tree                     extending;
     public  List<Tree>               implementing;
     public  List<Tree>               defs;

     public   ClassDef (…) {…}        //constructor
     public   void  visit( Visitor  v) { v._case(this); }
}
```

Figure 4-3 GJ Implementation of the Abstract Syntax Tree Node Representing A Class Definition

The root class `Tree` and each subclass define a method `visit` which takes a given `Visitor` object to the tree node. The `Visitor` object provides a traversal method *`_case`* corresponding to each individual subclass being defined. So, the actual tree processing is done by the `Visitor` classes.

The tree Visitors process the abstract syntax tree by recursively traversing the tree nodes. As shown in Figure 4-1, there are four main passes over the abstract syntax tree. Each pass is performed by its corresponding tree visitor:

1. Enter: By sweeping through all the syntax trees, the tree visitor `Enter` enters symbols for all encountered definitions into the symbol table.

2. Attr: The main context-dependent analysis phase is provided through this tree visitor.

3. TransType: Through this pass, the extended Generic Java is translated into the conventional Java.

4. Gen: the tree visitor Gen accomplishes the code generation phase.

In GJ compiler, all the tree traversal methods have the overloaded name `_case`. They are distinguished by the subclasses of `Tree`, which represent the concrete tree nodes and are taken as arguments of the traversal methods. An abstract `Tree.Visitor` class is defined as containing one `_case` method for each of the `Tree` subclasses. Each concrete visitor class is implemented as a subclass of the abstract class `Tree.Visitor` and overrides those `_case` methods. Figure 4-4 gives an overview of this.

```
public abstract class Tree {
   …
   public void  visit ( Visitor v ) { v._case (this); }
   public static abstract class  Visitor {
       public  void  _case (Tree  that) {
         throw new InternalError ("unexpected: " + that);
       }
       public void _case(TopLevel  that){_case((Tree) that);}
       public void _case(Import  that ) {_case((Tree) that);}
       public void _case(Vardef  that ) {_case((Tree) that);}
        …
    }
}

public class Attr extends  Tree.Visitor {
…
  public void  _case (ClassDef  tree) {
    //code for the attribution of a class definition tree node
 }
 public void  _case (MethodDef  tree) {
   //code for the attribution of a method definition tree node
 }
 …
}
```

Figure 4-4 Visitors in the GJ compiler

4.2 Modification of the GJ Compiler to Handle NextGen

NextGen is intended to address the type integration problems of GJ. The syntax of NextGen is similar except that it supports parametric type in any context, i.e. the type-dependent primitive operations such as

```
instanceof  Vector <X>
new    T[...]
new    T(...)
```

where T is a type parameter and X represents any specific reference type, are legal in NextGen.

NextGen compiler employs the lexical analyzer and parser from GJ. Through the lexical analyzer, a source file is mapped into a token sequence. The parser then maps the token sequence to an abstract syntax tree.

Like GJ, the NextGen compiler is structured as a series of passes over the abstract syntax tree. The actual language processing is still done by the tree `Visitor` class, but the implementation of the tree `Visitor` is more complicated.

As we have discussed in the previous sections, NextGen implements parametric types in Java by an enhanced type erasure model. The type erasure process that

NextGen uses to construct base classes generates essentially the same code for parameterized classes as the corresponding unparameterized code written using the Java generic coding idiom. This is similar to GJ. However, NextGen is designed to provide run time type information. During the translation from NextGen to conventional Java, the NextGen compiler must automatically generate the lightweight wrapper interfaces and wrapper classes. For the parametric classes involing type-dependent operations, the NextGen compiler must automatically introduce a package-private auxiliary snippet method in the type-erased base class for each one of such operations. All these snippets should be abstract in the base class. The corresponding wrapper classes then appropriately override these snippets, and therefore are able to correctly carry the run time type information.

In view of the these needs, we extend the GJ compiler by changing the behaviors of the concrete tree `Visitors` such as `Attr` and `TransType`. In a `NextGen` compiler, the program, represented as the abstract syntax tree, is changed through the series of traversal passes.

Because the Tree visitors provide an individual traversal method for each kind of tree node, we implement the traversal methods with the algorithm shown in Figure 4-5.

Step 1: Check if the processed tree node involves a parametric type. If not, go to step 5. Otherwise go to step 2.

Step 2: Check if the corresponding wrapper class and wrapper interface already exist in the same package. If not, go to step 3. Otherwise go to step 4.

Step 3: Generate the corresponding wrapper class and wrapper interface.

Step 4 : Modify the processed tree node with the corresponding wrapper class or interface. Go to step 7.

Step 5: Check if the processed tree node involves the type dependent primitive operations:

new T[...]
or new T(...)

where T is a type parameter. If not, go to step 7. Otherwise, go to step 6.

Step 6: Generate a snippet method tree node (a kind of `MethodDef` tree node) corresponding the type-dependent operation in step 5; embed the snippet method tree node into the corresponding `ClassDef` tree node; modify the processed tree node with the added snippet method tree node.

Step 7: Do the normal traversal processing as in GJ.

Figure 4-5 The Algorithm for Modifying Traversal Methods of the Tree Visitor in GJ Compiler to handle NextGen

Because a parameterized type, T<...>, can be represented as a tree node of `TypeApply`, the step 1 is easy to accomplish. For example, suppose we are passing across a `VarDef` tree node for the attribution phase, where the `VarDef` tree node represents a variable definition. The `VarDef` tree node definition (nested inside the abstract class Tree) and the Visitor method definition for a VarDef tree node look like the following:

```
public class VarDef extends Tree {
   public Name  name;          //the variable name
   public Tree  varType;       //type of the variable defined
     ...
  }

public  void  _case(Vardef  tree) {
      //codes for checking that the variable's declared type
      //is well-formed
   ...
}
```

Here, in the traversal method we need only to check if `tree.varType` is a kind of `TypeApply` tree node and we can then decide if the defined variable has a parametric type.

In step 2, there is a need to check a specific generated wrapper class and a interface in the same compilation unit. In the original GJ compiler prototype, there is a class `ClassReader` which provides operations to read a class file into a internal representation. This class can be used to check for the existence of a wrapper class by testing for the generated `.class` file.

As decided in step 3, for an instantiation of a parametric type the corresponding wrapper class and wrapper interface need be generated. Here, knowing the properties of the base class is essential. In the NextGen compiler, we use the Java reflection feature to generate the wrapper class and wrapper interface.

"Reflection" is a particular feature of the Java programming language. It allows an executing Java program to discover information about the modifiers, the fields, the constructors and methods of a loaded class, and then examine and manipulate the Java class from within itself. This feature does not exist in other conventional programming languages such as Pascal, C and C++ [22]. We use reflection to obtain information about the base class file for a parametric class $C<T_1, \ldots>$. Once the base class information is in hand, we can easily specialize the snippets in forming the wrapper class and then the wrapper interface. Figure 4-6 shows how to use the reflection, where `baseClassName` is a string representing the base class name of a parametric class.

To get the information about a class that we want to manipulate, we first need to obtain a `java.lang.Class` object for the corresponding class. Instances of the class `java.lang.Class` represent classes and interfaces in a running Java

application. The reflection classes such `Modifier`, `Constructor` and `Method` can be found in `java.lang.reflect`. The class `Modifier` provides a method to decode the class and member access modifiers; the class `Constructor` provides information about a constructor for a class; and the class `Method` provides information about, and access to, a single method of a class or interface. In Figure4-6, statement (2), we can get the Java language modifiers for the class or interface, encoded as an integer. From the statement (3) we get an array of `Constructor` objects reflecting all the constructors declared by the class represented by this `Class` object. From statement (4) we find out what methods are defined in the class.

```
Class  baseClass = Class.forName( baseClassName );        //(1)
Int modifiers = baseClass.getModifiers();                 //(2)
Constructor[] cons = baseClass.getConstructors();         //(3)
Method[]  meths = base.getDeclaredMethods();              //(4)
 ...
```

Figure 4-6 Illustration for How Reflection Works

By discovering and using the member information of a base class from a given parametric class, it is easy to define the corresponding wrapper class and wrapper interface for an instantiation of parametric class. As illustrated in Figure 4-6, we can

get a list of all the methods declared by a class. We can therefore decide if the base class has snippet methods. If the base class has no snippet method (the parametric class does not include the operation new T(...) and new T[...], where T is a type parameter), we create the wrapper class so that it inherits all the methods of the base class without overriding them. We also replicate base class constructors by forwarding constructor calls to the base class using *super*. Otherwise, in the wrapper class we have to further provide the corresponding definition of the abstract snippets methods in the base class. We use different kinds of snippet method names to differentiate the operation new T[...] and new T(...):

new T[...] → $snip$ followed by a number

new T(...) → $snipNewclass$ followed by a number

We can therefore make a decision from the snippet method name whether the allocation operation is for an array (new T[...]) or new object (new T(...)).

For example, in a parametric class C<T>, the statements:

```
T  t1;
t1 =  new  T(10);
...
T[]  elems;
elems = new T[size]; //size is variable representing a int
...
```

may be type erased to the base class C as:

```
Object  t1;
t1  =  $snipNewclass$1 ( 10 );
…
Object[] elems;
elems  =  $snip$1(size);  //size is variable representing a int
…
protected abstract  Object $snipNewclass$1 (int  intValue );
protected abstract Object[] $snip$1 (int size);
…
```

When an instantiation of C<T>, such as C<Integer>, is encountered, a corresponding wrapper class $$C$_Integer_$ and wrapper interface C_Integer_$ are created. The wrapper class will implement the corresponding abstract methods in the base class as:

```
protected   Object[]    $snip$1(int intValue)
{
    return    new  Integer[intValue];
}
protected    Object     $snipNewclass$1(int intValue)
{
    return  new  Integer(intValue);
}
```

Then, in the following step, we modify the processed tree node by updating the `TypeApply` node with the corresponding `Ident` tree node which represents the type corresponding to the wrapper class. In the compiler all the unparameterized types are represented as an `Ident` tree node during the compilation. We handle a wrapper class as if it were an unparametric class.

The discussion of the other steps has been included into the extended discussion of step 3.

4.3 Experiments and Results

Because our experimental compiler is written in GJ, we tried using our NextGen compiler to compile the compiler source code itself. In our experimental setup, we also designed 12 parameterized classes specifically containing runtime dependent operations. One of the experimental test programs is shown below:

```
public class NGVector<T>{
    private T[] elements;  //the array buffer into which the elements
                             of the Vector are stored
    private int capacity;
    private int size;      //the number of the elements in the Vector

    //Constructor
    public NGVector(int i){
      elements = new T[i];     //This is the feature of NextGen
      capacity = i;
      size = 0;
    }


    public void addElement1(T t){
      T t1;
      t1 = new T(10);            //This is the feature of NextGen
      if( size < capacity){
          elements[size] = t;
          size++;
      }
    }


    public String toString(){
      String s = "";
      for(int i = 0; i<size; i++){
          s =  s + elements[i];;
          if(i != size-1)
            s = s + ",    ";
      }
      return s;
    }
}


public class t1{

    public static void main(String[] argv){
      String s;
      t0     s1;
      s1 = new t0();

      NGVector<Integer> IntVector;
      IntVector =  new NGVector<Integer>(20);


      for(int i= 1; i<=20; i++)
          IntVector.addElement1(new Integer(i));

      System.out.println(IntVector.toString());

    }
}
```

Our study of the NextGen proposal, and implementation of a compiler prototype, lead us to believe that the type erasure technique can also be enhanced to provide run time parameterized types with run time information in Java. Just as GJ is designed as a superset of the conventional Java programming language, the NextGen proposal is upward compatible with GJ. Our experiments show that a legal GJ or conventional Java program can also be compiled by the NextGen compiler into JVM code.

The most important improvement to the GJ compiler is that the NextGen compiler has the ability to provide run time support for parametric type instantiation. This makes NextGen more expressive than GJ and better fit with Java programming mechanism, which maintains run time type information about the class of an object and the type of the elements of an array. The primary drawback of GJ is that it does not carry any run time type parameter information, so it can not support type operations over type variables, or instance tests over parametric types. However, In NextGen, the parametric type expressions can be used anywhere that conventional types can be used. Specifically, the following type dependent primitive operations can be supported in NextGen:

```
new T[...]
new T(...)
instanceof C<refType>
```

where T is a type parameter and refType is a kind of reference type.

Consider a parametric class C<T> with a field: T[] x, where x is initialized by an operations `new T[...]`. With GJ, the code can pass the compiler, but a "unchecked" warning message will be issued. Further, because Java arrays carry their element type at run time and the Java type system forces array assignments to be type-correct, an array value of type Object[] cannot be assigned to a variable of type Integer[]. So, if a variable Y has type Integer[] and Z is a object of type C<Integer>, the operation Y = Z.x will generate a run time type error even though they seem type correct at compile time. On the other hand, the NextGen compiler will postpone the initializing operation new T[...] until the object Z of `C<Integer>`, an initialization of the parametric type C<T>, is generated. The operation new T[...] is then done as new Integer[...]. In this way, NextGen is always type-sound with respect to parametric type declaration.

We have seen from our implementation of the NextGen compiler that NextGen cannot avoid the continuous creation of new class files as parameterized classes are instantiated with different types. Growth in the number of files is a problem with the heterogeneous translation approach. This is because, in Java, each compiled class requires its own file.

Our experiments on the NextGen compiler also show that, compared with GJ, NextGen has weaker compatibility properties. In GJ, because all the instantiations of a given parameterized class such as Collection<T> are translated into a single class Collection<Object> (which is identical to an unparameterized Collection class

implemented using the Java generic coding idiom), GJ achieves great compatibility with the Java legacy code. In GJ, objects created by legacy code using the Java generic coding idiom may be passed to new GJ code that expects objects of parameterized type, and also a new created objects of a parametric type may be passed to legacy code that uses the Java generic coding idiom. For example, the following code is considered to be legal in GJ.

```
LinkedList<String> LS = new  LinkedList<String>();
LinkedList  L = LS;
```

Further, in GJ, the legacy code using the Java generic coding idiom may call the new parameterized libraries and the new parameterized code may also call the legacy libraries that use Java generic coding idiom. So, only one version of the library is required in GJ. If one anticipates to eventually rewrite the Java source library with parametric types, the task can be scheduled at a convenient time.

In contrast, because the NextGen compiler translates a parameterized class such as Collection<T> into a base class Collection which may not be identical to the unparameterized class Collection (that is implemented using the Java generic coding idiom) the new NextGen code cannot use the legacy libraries directly. So, rewriting the Java source library with parametric types is preferable with NextGen. In another a separate convertation of the libraries with adaptor is absolutely necessary.

CHAPER 5

Conclusions and Future Work

5.1 Conclusions

Our experiments in developing a NextGen compiler have shown that relying on an enhanced type erasure technique to achieve parameterized classes with run time parameter information in Java is workable.

Our NextGen compiler employed a mixture of both homogeneous and heterogeneous translation. It cannot avoid the new class creation problem of heterogeneous translation, but it limits these new classes to be very small in size.

When the full integration between parametric types and Java language features is not required, GJ remains the best solution. The reason is that GJ does not cause overhead, and it also achieved a great compatibility with the Java legacy code.

5.3 Future work

NextGen has achieved the complete integration of the parameterized types with the conventional Java programming language typing at the cost of increase in number of small class files and weaker compatibility properties with the Java legacy code. For example, an object created with legacy code that uses the Java generic coding idiom cannot be passed directly to the new parametric code from NextGen. Considering a smoother process of upgrading from legacy code to the parameterized code still needs further study.

With NextGen, a new parametric code cannot use the Java legacy libraries. Rewriting the source library with parametric types is also is a considerable task.

In our experimental NextGen compiler, we used the Java reflection feature in generating the wrapper classes and interfaces. This means that the base class of the parameterized class must already exists before its client uses it. If the client is within the same file as the parameterized class, our prototype compiler cannot work properly. Further implementation with our existing compiler would be needed to solve this problem or we can try another approach such as adding extra attributes to the base class file for a parameterized class.

References

[1] Krzysztof Czarnecki and Ulrich W. Eisenecker. Generative Programming, Methods, Tools and Applications, Addison-Wesley, 2000

[2] S. Yu. Notes of the course Theory of Objects, 2001.

[3] J. H. Solorazno and S. Alagic. Parametric polymorphism for Java: A reflective solution. In conference on Object-Oriented Programming, systems, languages and Applications, pages 216-225. ACM, 1998.

[4] M. Odersky, P. Walder, G. Gracha, and D. Stoutamire. Making the future safe for the past: Adding Genericity to the Java programming language. In Conference on Object-Oriented Programming, Systems, Languages and Applications, pages 183-200. ACM, October 1998.

[5] M. Odersky and P. walder. Pizza into Java: Translating theory into practice. In Symposium on Principles of Programming Languages, pages 146-159. ACM, 1997.

[6] James Gosling, Henry McGilton. The Java Language Environment, A White Paper. May 1996. http://java.sun.com/docs/white/langenv/.

[7] Jr. Guy L. Steele. Growing A Language. Journal of Higher-Order and Symbolic Computation, 221-236, 1999.

[8] Mirko Virlli. Parametric Polymorphism in Java: an Efficient Implementation for Parametric Methods. Symposium on Applied computing. Pages 610-619. ACM SAC2001.

[9] Dominic Duggan. Modular Type-based Reverse Engineering of Parameterized Types in Java Code. In Conference on Object-Oriented Programming, Systems, Languages and Applications, pages 97-113. ACM, 1999.

[10] Kresten Krab Thorup. Genericity in Java with virtual types. European Conference on Object-Oriented Programming, pages 444-471, LNCS 1241, Springer-Verlag, 1997.

[11] Kim B. Bruce, Martin Odersky, Philip Wadler. A statically safe alternative to virtual types. In Proceedings of the European Conference on Object-Oriented Programming, page 1-27. ECOOP 1998.

[12] Cosling J., Joy B., and Seele G. The Java Language Specification. Addison-Wesley. 2000.

[13] Andrew C. Myers, Joseph A Bank, Barbara Liskov. Parameterize Types for Java. In Symposium on Principles of Programming Languages, page 132-145. ACM, January 1997.

[14] T. Lindolm and F. Yellin. The Java Virtual Machine. Addison-Wesley. 1996

[15] Ole Agesen, Stephen Freund, and John Mitchell. Adding Type Parameterization to the Java Language. In Symposium of Object-Oriented Programming: Systems, Languages, and Applications, ACM, October 1997.

[16] Alex Tomlins, Chris Jackson. Java Generics: Final Report June 1999. http://www.iis.ee.ic.ac.uk/~frank/surp99/report/caj97/

[17] Gary McGraw, Ed Felten. Securing Java: Getting down to Business with Mobile Code. John Wiley & Sons, Inc. 1999.

[18] Gilad Bracha, Martin Odersky, David Stoutamire, Philip Wadler. GJ: Extending the Java programming

[19] Gilad bracha, Martin Odersky, David Stoutamire, Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In

Conference on Object-Oriented Programming, Systems, Languages and Applications, page 183-200. ACM 1998.

[20] Robert Cartwrigh, Guy L. Steele Jr. Compatible Genericity with Run-time Types for the Java Programming Language. In Conference on Object-Oriented Programming, Systems, Languages and Applications, page 201-215. ACM 1998.

[21] Sun Microsystems, Inc. JDK 1.2: The Java Collections Framework,

[22] Glen McCluskey. Using Java Reflection. January 1998.
http://developer.java.sun.com/developer/technicalArticles/ALT/Reflection

[23] Ran Rinat, Scott F. Smith. Correspondence Polymorphism for Object-Oriented Languages. In Conference on Object-Oriented Programming, Systems, Languages and Applications, pages 167-178. ACM, 1999.

VITA

Name:	Huanling Lu
Place of birth:	Shanxi, China
Year of birth:	1967
Post-secondary Education and Degree:	Tsinghua University Beijing, China 1984-1989 B. Eng. Tsinghua University Beijing, China 1991-1994 M. Eng. The University of Western Ontario London, Canada 1998-2000 Special Undergraduate Student The University of Western Ontario London, Canada 2000-2002 M. Sc.
Honors and Awards:	Dean's honor list 1998-1999
Relate work experience:	Teaching Assistant The University of Western Ontario 2000-2002