RICE UNIVERSITY

Efficient Implementation of First-class Polymorphic Methods in Java

by

James Sasitorn

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Robert Cartwright, Chair
Professor of Computer Science

Walid Taha
Assistant Professor of Computer Science

Keith Cooper
Professor of Computer Science

Houston, Texas

April, 2005

Efficient Implementation of First-class Polymorphic Methods in Java

James Sasitorn

Abstract

This thesis describes a new implementation architecture for polymorphic methods in Generic Java using the NextGen compiler framework. The standard Generic Java (Java 1.5) compiler erases generic types at compilation. This transformation prohibits type-dependent operations, limiting generic expressivity. Type erasure causes unchecked warnings at compilation, and unexpected behavior or exceptions at runtime. Alternative reflection-based implementations of Generic Java support type-dependent operations at the cost of significant execution overhead. In contrast, this work presents an efficient implementation of polymorphic methods using NextGen. An extended NextGen compiler generates snippet environment template classes to encode type-dependent operations for polymorphic methods. A customized classloader generates specialized template instantiations on demand. This demand-driven code specialization provides an efficient mechanism to propagate runtime type information, while maintaining backwards compatibility with existing libraries and Java Virtual Machines. Benchmarks show runtime support for polymorphic methods in NextGen outperforms reflection-based approaches, with runtime overhead comparable to erasure-based Generic Java.

Acknowledgments

I would like to thank my advisor, Professor Robert "Corky" Cartwright, for his guidance and insight in my research. He embodies an uncanny fusion of the theoretical and practical aspects of programming languages and software development.

I also would like to thank Professor Walid Taha for introducing me to the more theoretical aspects of programming language semantics.

I've also had the privilege of working with many talented individuals in the Rice JavaPLT team. In particular, I would like to thank Moez Abdel-Gawad and Michael Jensen for our many insightful discussions on NextGen and software development.

Finally I would like to thank my family and friends for all their support in this arduous endeavor.

Contents

Illustrations

Chapter 1

Introduction

Java has changed the nature of software programming with its support for object-oriented design, comprehensive static type checking, "safe" program execution, and its unprecedented degree of portability. Despite these great strides, the absence of generic types prior to Java 5 has prohibited the expression of many statically checkable program invariants within the type system. Typically Java programmers simulate parametric polymorphism by using the universal type `Object`, or any more suitable bounding type, in place of a type parameter T, and then insert casts to convert the erased object back to the particular instantation type. Aside from the clutter of casting operations, this methodology obscures type abstractions and thus degrades the precision of static type checking. Generic types allow classes and methods to be parameterized with respect to type, thus providing type abstraction that could not otherwise be expressed in a statically typed language.

The most recent major release of the Java platform (J2SDK 5.0) marks an important step in the advancement of the Java language. Java 5.0 supports a second-class formulation of Generic Java called GJ. GJ supports type parameterization of classes and methods, but prohibits the use of parameterized types in type-dependent operations. The unsupported operations include parametric casts, parametric `instanceof` operations, and `new` operations of "naked" parametric type, e.g., new T(). This subtle restriction is necessary since parametric type information is erased during compilation. The compiler generates a single, type-erased class file for all instances of a generic class.

These limitations fueled the development of an alternative, first-class formulation

of generics called NextGen, originally designed by Cartwright and Steele, that is upward compatible with with GJ. NextGen overcomes the limitations inherent in GJ by introducing light-weight classes that inherit common code from a type-erased base class to represent each instantiation of a generic type. Although this mildly heterogeneous implementation of genericity is more complex than GJ, it is just as efficient and fully compatible with existing Java legacy code.

The original NextGen2 architecture supported polymorphic methods based on the assumption that possible instantiations of generic classes and polymorphic method instantiations could be statically bound. However, the design of generics in Java 5 allows cycles in the type application graph, thus enabling programs to create an infinite number of class and method instantiations. Allen and Cartwright revised the NextGen2 implementation architecture to create class instantiations on demand using a custom classloader[2]. But their design for supporting polymorphic methods was incomplete and unimplemented.

This thesis presents an efficient implementation of polymorphic methods using NextGen. This new work, called NextGen2 to avoid any ambiguity, applies the demand-driven code specialization techniques used for generic classes to provide support for type-dependent operations in polymorphic methods. Specifically, NextGen2 creates light-weight templates called `snippet environments` to encapsulate the type-dependent operations used in polymorphic methods. Thus, type-dependent operations are fully supported without any loss in compatibility with legacy code or Java Virtual Machines (JVMs).

The remainder of this thesis is organized as follows. Chapter 2 introduces the existing NextGen translation for parametric classes. Chapter 3 discusses the uses of parametric methods and their design in the NextGen2 compiler. Chapter 4 provides more technical implementation details of this work in NextGen2. Chapter 5 provides an overview of other research in the field. Chapter 6 provides performance benchmarks of polymorphic methods, and chapter 7 concludes.

Chapter 2

NextGen2 Fundamentals

NextGen2 extends GJ to propagate parametric type information to the Java Virtual Machine (JVM) runtime environment. NextGen2 supports the same basic source language `Generic Java`, but with greater expressiveness. Generic Java is ordinary Java (JDK 1.2- JDK1.4) extended to support parameterized types. The extension is detailed in the JSR14[7] proposal to add generic types in Java 5. In Generic Java, class and method definitions are augmented to include type parameters that can be referred to in the enclosing body. Class declarations are generalized from:

```
Identifer
```

to

```
Identifier < TypeParameters >
```

where

```
TypeParameters → TypeParm | TypeParm, TypeParameters
TypeParm → TypeVar { TypeBound }
TypeBound → extends ClassType | implements InterfaceType
TypeVar → Identifier
```

and { } enclose optional phrases. Interface declarations are similarly generalized. In addition, the definition of ReferenceType is generalized from

```
ReferenceType → ClassOrInterfaceType | ArrayType
```

to

```
ReferenceType → ClassOrInterfaceType | ArrayType | TypeVar
TypeVar → Identifier
ClassOrInterfaceType → ClassOrInterface { <TypeParameters> }
ClassOrInterface → Identifier | ClassOrInterfaceType.Identifier
```

and the syntax for *new* operations now includes the additional phrase

```
new TypeVar ( { ArgumentList } )
```

The new generic `RefernceType` can appear in any context a class or interface name can in ordinary Java except in the `extends` or `implements` clause of a class definition. Method declarations are generalized with a slight variation. A method declaration header is generalized from

```
{modifiers} Type Identifier ( {ArgumentList} )
```

to

```
{modifiers} {<TypeParameters>} Type Identifier ( {ArgumentList} )
```

and method invocations are modified from

```
{ScopeIdentifier .} Identifier ( {ArgumentList} )
```

to

```
{ScopeIdentifier .} {<TypeParameters>} Identifier ({ArgumentList} )
```

2.1 GJ Implementation Scheme

The GJ proposal for Generic Java by Bracha, Odersky, Stoutamire, and Wadler provides the illusion of generic types by using type erasure[3]. Each parametric class `C<T>` generates a single class file containing the erased base class `C`. Similarly, each parametric method `<T>m` generates a single erased method m. The erasure of a parametric type `T` is obtained by replacing each type parameter t with its upper bound. All references to parametric classes or methods are replaced with references to their erased versions.

2.2 Implications of GJ Type Erasure

Erasure of parametric type information causes typing inconsistencies. Consider a generic class `Stack<T>` that extends a generic class `Vector<T>`. Intuitively an instantation `Stack<E>` should be a subtype of `Vector<E>`. Correct subtyping is possible under erasure; `Stack` is a subtype of `Vector`. However, by the same logic any instantiation class `Stack<F>` is also considered a subtype of `Vector<E>`.

A more dramatic problem occurs on operations involving "naked" parametric types such as `new T()` and `new T[]`. The type parameter `T` is erased to its upper bound, in this case `Object`. So these two operations are discretely erased to `new Object()` and `new Object[]`, respectively. This results in unexpected runtime behavior and exceptions.

To minimize the problems of erasure, GJ prohibits operations that depend on runtime information. These type-dependent operations occur when using `new`, `instanceof`, or casting instructions with a naked type parameter or a generic type parameterized by a type parameter. Even with these restrictions, some operations still cannot be statically checked and cause `unchecked` warnings during compilation.

2.3 NextGen Implementation Scheme

The NextGen implementation of Generic Java corrects the inaccuracies in subtyping and eliminates the restrictions on type-dependent operations imposed by GJ. NextGen improves on GJ type erasure by making the erased base class C abstract and creating lightweight classes that extend C to represent instantiations `C<E>`. The type-dependent operations in `C<T>` are not erased in `C`, but rather translated into calls on synthetically generated `snippet` methods. The instantiation classes `C<E>` overload these `snippet` methods in C to provide specialized code encapsulating the type-dependent operations for `C<E>`.

2.4 NextGen Support for Parametric Types

While the use of lightweight template classes preserves runtime type information, it also breaks intuitive subtyping relationships. The earlier example using `Stack<T>` and `Vector<T>` illustrates the problem. The instantiations `Stack<E>` and `Vector<E>` must inherit code and thus extend the base classes Stack and Vector, respectively. Also, intuitively `Stack<E>` is a subtype of `Vector<E>`. Figure 2.1 shows this illegal hierarchy. However, this multiple inheritance is impossible in the single inheritance Object model in Java.

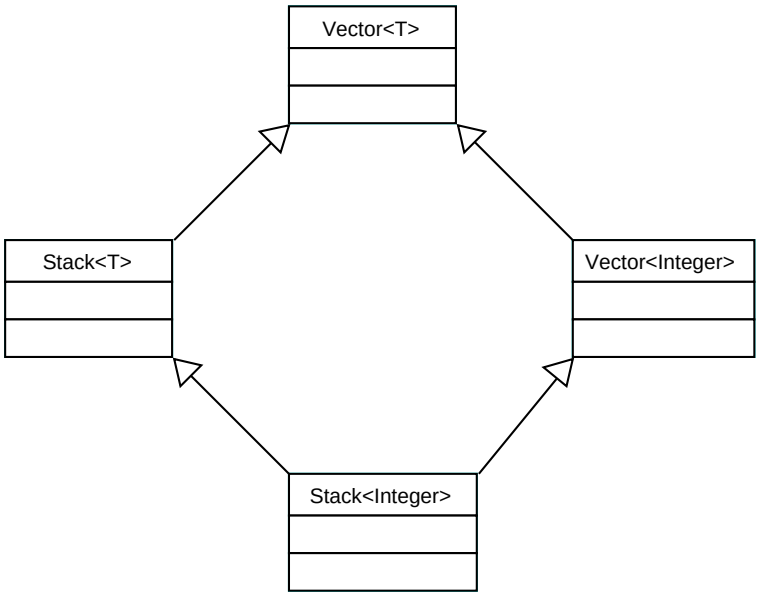


Figure 2.1 : Illegal Multiple Inheritance Class Hierarchy

In the original NextGen paper, Cartwright and Steele demonstrated how multiple interface inheritance solves this multiple inheritance predicament[4]. By introducing an empty interface `C<E>$` which is implemented by the instantiation class `C<E>`, NextGen effectively decouples the generic instance `C<E>` from its associated type. References to `C<E>` are replaced with references to `C<E>$`. Figure 2.2 shows how 2.1 can reformulated into a single class inheritance hierarchy. Since Java allows

multiple inheritance of interfaces NextGen can provide correct runtime subtyping. Furthermore, since these light-weight interfaces contain no fields or methods, their use only marginally effects program code size. A more detailed discussion is available in the earlier NextGen papers [4] [2].

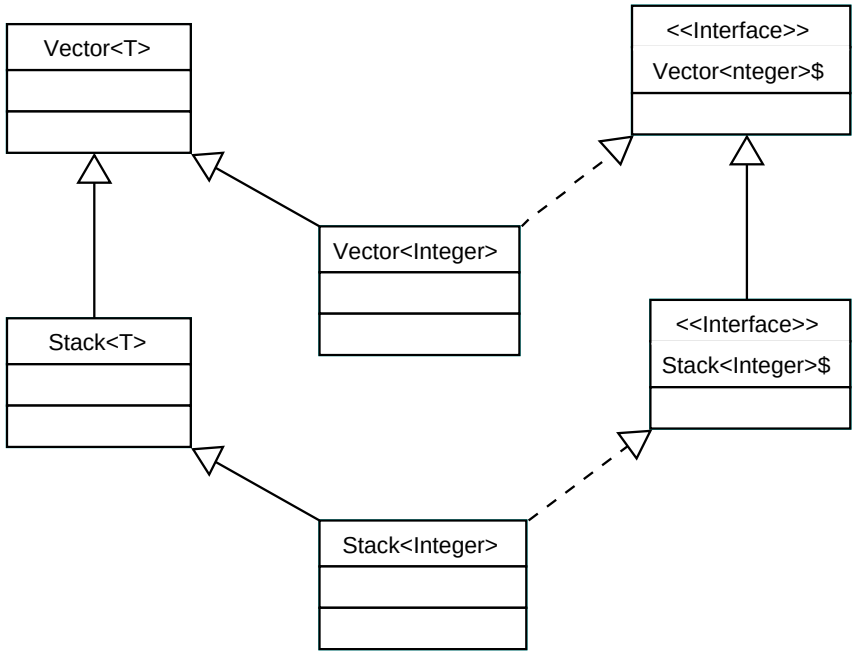


Figure 2.2 : Intuitive Parametric Type Hierarchy using Interfaces for Typing

2.5 NextGen Support for Polymorphic methods

Earlier versions of NextGen did not provide comprehensive support for polymorphic methods. The original NextGen2 design supported polymorphic methods based on the assumption that possible instantiations of generic classes and polymorphic method instantiations could be statically bound. However, the design of generics in Java 5 allows cycles in the type application graph, thus enabling programs to create an infinite number of class and method instantiations. Allen and Cartwright revised the NextGen2 implementation architecture to create class instantiations on demand

using a custom classloader[2]. But their design for supporting polymorphic methods was incomplete and unimplemented.

Chapter 3

NextGen2 Design

This chapter discusses how the NextGen compiler described in chapter 2 is extended to support polymorphic methods in first-class genericity. Sections 3.1-3.2 discuss the uses of polymorphic methods and the current support in the JVM. Sections 3.3-3.4 details the translation use in NextGen2 to support static polymorphic methods, methods where dynamic dispatch is not possible. Finally, Section 3.5 extends the static translation to support the case of dynamic polymorphic methods.

3.1 Polymorphic Type-dependent Operations

Polymorphic methods expand the number of generic types available in the body of a method. This allows for type-dependent operations based on the parametric type information of other generic class instantiations. As discussed in section 2.2, type-dependent operations include `new`, `instanceof`, or casting instructions with a naked type parameter or a generic type parameterized by a type parameter.

Figure 3.1 presents a simple example of `List.zip`, the function that creates a new List containing a lexicographic pairing of the elements from the two input lists. The type variables `T` and `U` are used in the type-dependent instantiations of `new Pair<T,U>` and `new List<Pair<T,U>>`.

3.2 Erasure of Polymorphic Methods

Under type-erasure, the parametric source code in figure 3.1 is converted into the version shown in figure 3.2. In the erased code, the parameter and return types are all replaced by their parametric upper bound `List`. The most significant change is

```
class List<T> {
   T first;
   List<T> rest;

   List(f, r) { first = f; rest = r; }

   static <T, U> List<Pair<T,U>> zip (List<T> left, List<U> right) {
       ...
      return new List<Pair<T,U>>( new Pair<T,U>(left.first, right.first),
                      List.<T,U>zip(left.rest, right.rest));
   }
}

class Pair<A,B> {
   A x;
   B y;
   Pair(x,y) { this.x = x; this.y = y; }
}

class Client {
   public static void main(String[] args) {
      List<Integer> i = new List<Integer>(new Integer(1), ... );
      List<String> s = new List<String>("A", ... );

      List<Pair<Integer, String>> p = List.<Integer, String>zip(i, s);
   }
}
```

Figure 3.1 : Static Polymorphic Method with type-dependent operations

the conversion of `List<Pair<Integer, String>>` to `List`; this represents a loss of two levels of type parameterization. In general, type erasure generates a hole in the type system at each return from a function. As a result, erasure hinders concise and expressive object-oriented design patterns in Generic Java.

```
class List {
   Object first;
   List rest;

   List(f, r) { first = f; rest = r; }

   static List zip (List left, List right) {
       ...
      return new List( new Pair(left.first, right.first),
                       zip(left.rest, right.rest));
   }
}

class Pair {
   Object x;
   Object y;
   Pair(x,y) { this.x = x; this.y = y; }
}

class Client {
   public static void main(String[] args) {
      List i = new List(new Integer(1), ... );
      List s = new List("A", ... );

      List p = List.zip(i, s);
   }
}
```

Figure 3.2 : Type-erasure of a Static Polymorphic Method

3.3 Naive Implementation of Polymorphic Methods

One possible implementation of polymorphic methods that supports runtime type-dependent operations is to translate each method into a parameterized inner class

with a single execution method[10]. The inner class is parameterized by all the type parameters of the original method and its enclosing class. At each related call site, a generic instance of the related parametric inner class is generated and invoked. A translation of `List.zip` is shown below.

```
static class List$zip<T,U> {
   zip() { }

   List execute (List<T> left, List<U> right) {
       ...
      return new List<T,U>( new Pair(left.first, right.first),
                            zip(left.rest, right.rest));
   }
}
```

The heavy-weight translation shown above parallels the heterogeneous translation of template classes in `C++`. Each distinct use of `List.zip` would generate a complete copy of the `List$zip` template, resulting in excessive code duplication and unnecessary time overhead.

So while this is not an ideal translation, this approach brings polymorphic methods into more familiar territory. Since NextGen already provides an efficient implementation of generic classes that support type-dependent operations, an efficient implementation of polymorphic methods should follow a similar approach. Specifically, it should leverage light-weight instantiation templates and snippetized code to encapsulate type-dependent operations.

3.4 NextGen2 Translation of Static Polymorphic Methods

This section describes the NextGen2 transformation to support static polymorphic methods. Static polymorphic methods include all polymorphic methods declared

private, static, or final. Then section 3.5 shows how this translation can be extended to the dynamic case.

NextGen2 supports static polymorphic methods through a primarily homogenous scheme of translating only the type-dependent operations into calls on a specialized `snippet environment`. The `snippet environment` is a light-weight singleton containing only snippet methods for type-dependent operations, and a static field bound to the only instance of the class. A `snippet environment` instance is created at each polymorphic call site and passed as an argument to the method. The name of the instantiated `snippet environment` encodes the instantiated type parameters, as well as typing information related to the statically inferred receiver type. For the sake of brevity, this chapter uses the naming convention of `[ClassIdentifier]$env` to denote a `snippet environment`. Figure 3.3 shows an implementation of `zip` using this approach.

While this transformation is more complicated than type erasure, it performs with minimal additional overhead. The overhead of light-weight `snippet environments` can be easily removed by specialized optimizations in the Just-In-Time (JIT) compiler.

While Generic Java programs create a large number of generic instantiations, these instantiations are based on a small defined set of parametric types. Since instantiation of `snippet environments` is deferred until runtime, NextGen2 generates instances of a `snippet environment` only on demand. This demand-driven approach prevents the accidental creation of infinite hierarchies that may arise in chains of polymorphic recursion.

Since different type instantiations use distinct `snippet environments`, NextGen2 prefixes polymorphic method signatures using `snippet environment` interfaces. This interface specifies the snippet calls used in the body of the method. The related snippet environment template, and thus all snippet environment instances, implement this interface to ensure correct static typing. This differs from the support for generic

classes, where interfaces are used simply to ensure correct typing of class hierarchies. For polymorphic methods, the name of the interface encodes the upper bounds of the polymorphic type parameters it represents. For simplicity, this chapter uses the naming convention of `[ClassIdentifier]$env$` to denote these interfaces.

3.4.1 Propagation of Runtime Types

A subtlety in the NextGen2 translation is the propagation of `snippet environments`. In figure 3.3, the recursive call is a polymorphic call using the same type parameters. So in this case, passing the current `snippet environment` propagates the polymorphic parameterization.

Direct passing of a `snippet environment` characterizes a larger set of invocation call graphs typical in Object-oriented paradigms. NextGen2 passes a `snippet environment` anytime a method invocation instantiates the same exact type parameters. This applies to recursive methods, helper methods, and chains of overloaded methods that provide additional initialization.

In general, `snippet environments` provides the glue to carry polymorphic type information from a call site to a subsequent method invocation. When the parametric types can be inferred statically, NextGen2 can explicitly generate the correct `snippet environment`. When the parametric types cannot be inferred statically, NextGen2 snippetizes the construction of `snippet environments`.

3.5 Dynamic Polymorphic Methods

In this section, the translation described in section 3.4 is extended to support the dynamic case, a polymorphic method in the presence of object-oriented dynamic dispatch. In other words, a polymorphic method in a class that can be overridden by a subclass*.

*This is a strong semantic reason why static inner classes will not suffice: polymorphic methods can be overriden in subclasses while inner classes cannot.

```
class List<T> {
   T first
   List<T> rest;

   List(f, r) { first = f; rest = r; }

   static <T,U> List<Pair<T,U>> zip (List$env$ $env, List<T> left, List<U> right) {
       ...
      return $env.new$List<Pair<T,U>>(new Pair<T,U>(this.first, other.first),
                      zip($env, left.rest, right.rest));
    }

class Pair<A,B> {
   A x;
   B y;
   Pair(x,y) { this.x = x; this.y = y; }
}

class Client {
   public static void main(String[] args) {
      List<Integer> i = new List<Integer>(new Integer(1), ... );
      List<String> s = new List<String>("A", ... );

      List<Pair<Integer, String>> p =
                 i.<String>zip(List$env<Integer,String>.ONLY, s);
   }
}

interface List$env$ {
    List<Pair> new$List<Pair<T,U>> (Pair f, List<Pair> rest);
}

class List$env<Integer, String> {
    List<Pair> new$List<Pair<T,U>> (Pair f, List<Pair> rest) {
        return new List<Pair<Integer, String>> (f, r);
    }
}
```

Figure 3.3 : NextGen2 translation of Zip static polymorphic method

Static polymorphic methods have a receiver type and parametric types, all of which can be inferred statically from the call site. Dynamic polymorphic methods, however, are associated with an object instance. So polymorphic type information originates from two sources: the call site and the receiver type. While the call site information is static, the receiver type information is dynamic.

As stated earlier, a `snippet environment` encodes the statically inferred receiver type. Therefore, it cannot encapsulate type-dependent operations that depend on the dynamic receiver type. NextGen2 could prohibit type-dependent operations involving both sources of parametric types. Type-dependent operations using class-level parameterization could then be handled using the basic NextGen approach: these operations can be snippetized in the related generic `instantation class`. However, this restriction seems severe since it prevents many intuitive uses of polymorphic methods.

To better understand the complexities caused by dynamic dispatch, this section will examine a dynamic version of `List.zip` originally introduced in figure 3.1. Suppose a subclass `RevList` overrides the method `zip` to return a different ordering of elements, e.g. a reversed list:

```
class List<T> {
   ...
   <U> List<Pair<T,U>> zip (List<U> other) {
         return new List<Pair<T,U>>( new Pair<T,U>(this.first, other.first),
                        rest.<U>zip(other.rest));
   }
}
class RevList<T> extends List<T> {
   ...
   <U> List<Pair<T,U>> zip (List<U> other) {  ... }
}
```

A naive translation would prefix each method's parameters with a distinct `snippet environment`. This hierarchy is concisely described by the UML diagram in figure 3.4.

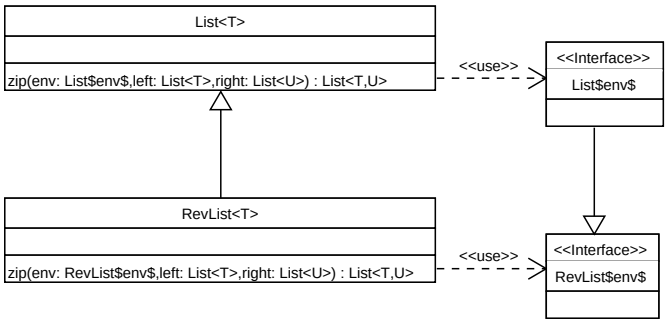


Figure 3.4 : Snippet Environment Hierarchy for Method Overrides

Now consider the call site:

```
List<Pair<Integer, String>> p = i.<String>zip(s);
```

Modifying the method signatures this way introduces two inconsistencies. First, method overrides in Java require invariance; the signature of a method override must match the signature of the original method.[†] So by prepending a different `snippet environment` to each method, this breaks the type invariance necessary for the subclass `RevList` to override the implementation of `zip` in its parent. Second, the `snippet environment` created at the call site is instantiated with respect to the static receiver type and may not accurately reflect its dynamic type. In the example above, if the variable `i` is statically typed to `List<Integer>` the actual receiver type could either be `List<Integer>` or `RevList<Integer>`, and the snippet environment would be of type `List$env$` or `RevList$env$`, respectively.

These two problems are closely intertwined. The latter is crucial since it is the mechanism by which NextGen2 propagates polymorphic method types.

[†]As of Java 1.4, invarience was relaxed slightly to allow a method override to narrow its return type.

3.5.1 Bridge Methods

NextGen2 introduces bridge methods in each subclass to restore the polymorphic method overrides from the parent class.[‡] Bridge methods must generate the correct `snippet environment` and invoke a forwarding call to the real method.

Figure 3.5 shows a bridge method in `RevList` that correctly overrides `zip` defined in `List`. Since `RevList` preserves the type parameterization in `List`, it should be possible to cast, and thus, forward a `List$env snippet environment`. However, for this to work properly, `List$env$` must contain all the type-dependent operations, or snippets, present in `RevList$env$`. In other words, this means that `List$env$` must be a subtype of `RevList$env$`.

In general, dynamic dispatch forces a class's `snippet environment` to be a subtype of all of its subclasses' `snippet environments`. NextGen2 uses `snippet environment` interfaces to provide correct typing. Then, after type checking NextGen2 propagates the snippets in each `snippet environment`, upward the class hierarchies, to its related `snippet environments`. This is the reverse of the scheme used to propagate snippets through the class heirarchy for parametric classes.

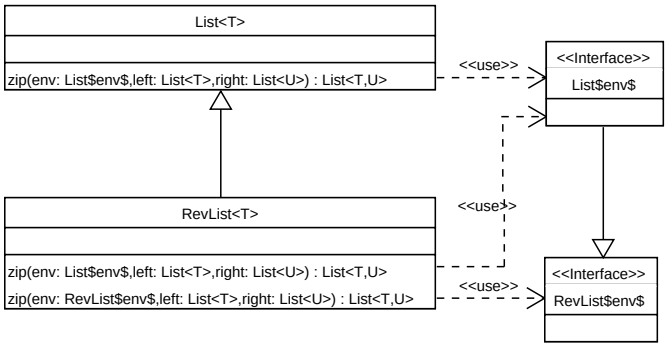


Figure 3.5 : Snippet Environment Hierarchy for Method Overrides

[‡]Bridge methods are already used in Java 5 to correct inconsistencies of method signatures caused by type erasure

3.5.2 Consolidate snipet environments

When a subclass preserves class-level parameterization, NextGen2 can provide a more optimized solution. Specifically, when the statically inferred receiver type parameterization expresses all the available parametric types, a single `snippet environment` can correctly account for all possible dynamic dispatches and encapsulate all necessary type-dependent operations. In the `zip` example, if `RevList<T>` is declared as `RevList<T> extends List<T>`, then the instantation of type `T` can always be inferred at the call site. Thus, as shown in figure 3.6, a single `snippet enviornment` of type `List$env$` satisfies both executions of `zip`.

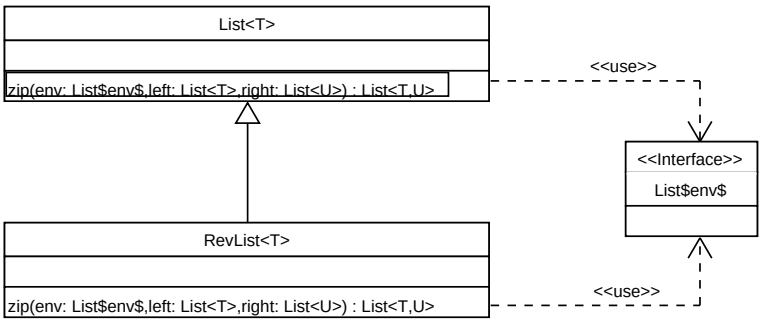


Figure 3.6 : Snippet Environment Hierarchy for Method Overrides

3.5.3 Pathological Use of Reflection

However, if a subclass introduces new class-level type parameterization, a single `snippet environment` may not be able to encode all the necessary type-dependent operations. The following conditions must occur between a class A and its subclass B:

- B introduces new class-level parameterization `T`.
- T is not mapped to parameterization in A via extends clause.

- B overrides polymorphic method `m` defined in A.
- Call site is statically typed receiver A.
- Call site is invoked with a dynamic receiver of class B.

Any `snippet environment` based on the statically inferred receiver type A at the call site will not be able to encode type-dependent operations related to the type parameter `T`. An example of this pathological case is shown in figure 3.7.

```
class Pair<S,T> {
    S s; T t;
    public Pair(S s, T t) {
        this.s = s;
        this.t = t;
    }
    public <X> Pair<S,X> changeSecond (X ob) {
        return new Pair<S,X>(s,ob);
    }
}

class Triple<S,T,U> extends Pair<S,T> {
    U u;
    public ReflectTriple(S s, T t, U u) {
        super(s,t);
        this.u = u;
    }
    public <X> Pair<S,X> changeSecond (X ob) {
        return new Triple<S,X,U>(s,ob,u);
    }
}

//call site
Pair myPair = ...;
myPair.changeSecond(new Boolean(true))
```

Figure 3.7 : Pathological reflection required for Method changeSecond

In these cases, NextGen2 uses reflection to determine the parameterized types of the receiver, and to instantiate the corresponding `snippet environment`. There

are three points of modification where NextGen2 can perform reflection: (1) at the call site, (2) in the method incatiion, and (3) in the snippet invocation.

1. **At the call site**: At a pathological call site, there is no way to distinguish between receivers that require reflection and those that don't. Therefore, under this approach a `snippet environment` must always be generated using reflection immediately before dynamic method invocation. This would slow down runtime performance, including the common non-reflection case.

2. **In the method invocation**: This approach requires using bridge methods, as outlined in 3.5.1, to provide the glue for reflection. The bridge method must use reflection to create the correct `snippet enviornment` and then pass it to the real method. Thus, only the pathological case will slowdown runtime performance. As a result, the performance penalty would depend on the number of pathological snippets actually executed in the body of the polymorphic method.

3. **In the snippet method**: On unsupported type-dependent operations, NextGen2 could use reflection to invoke the correct snippet method. However, since the `snippet enviornment` is generated in the snippet operation, it cannot be reused. In otherwords, the execution of multiple pathological snippets in succession must each perform reflection. So runtime performance would be affected only in the pathological case per each unsupported snippet invocation.

   Since a `snippet environment` has no reference to the pertinent receiver object, each snippet method signature must be altered to pass in the active receiver. Unfortunately, passing the receiver on all snippet calls would further impact runtime performance.

NextGen2 follows the second approach listed above. First, the NextGen2 compiler consolidates all `snippet environments` that can be shared in a class hierarchy. Then, it generates bridge methods for any remaining pathological cases. In

these cases, the NextGen2 compiler outputs compiler warnings specifying a possible performance degradation. Thus, a programmer can detect pathological cases, and possibly reformulate them using different design patterns.

Practically speaking, the pathological case rarely occurs in production level code.

Chapter 4

Implementation

NextGen2 uses an extended Java 5 compiler and a customized `classloader` to provide the framework necessary to propagate runtime type information. Previous versions of NextGen compiler were derivatives of the GJ compiler. The core functionality of NextGen was ported from GJ to Java 5. Since Java 5 is also a derivative of the GJ, the organization of the NextGen2 compiler follows that of NextGen. By building on Java 5, NextGen2 can immediately (1) Perform accurate type checking against the latest JSR14 specification, (2) Use support libraries in the JSR14 compiler to minimize code redundancy and improve readability, and (3) Take advantage of a more systematic compiler API with fewer bugs and inconsistencies.

4.1 Code Maintance

The cost of maintaining a full compiler is beyond the capabilities of the Rice JavaPLT research group. To understand the javac compiler requires not only a wide breath of knowledge on the diverse stages of compilation, but also a profound understanding of the peculiarities of the underlying implementation. Furthermore, it is infeasible to replicate the daily development of the javac compiler. There are numerous updates in the underlying javac implementation to fix bugs and to improve the interoperability between generic and non-generic code. Also, since the Java Programming language is under constant evolution, ie the recent addition of wildcards, it would be impossible to maintain a compatible compiler.

An optimal solution would decouple the NextGen2 extensions from the maintenance of the underly java compiler. One option is to implement NextGen2 as a

source-level preprocessor used before `javac` [2]. However, this approach is infeasible because NextGen name-mangling techniques are illegal in GJ identifiers. A second option is to implement NextGen2 as a post-processor on Java bytecode. Unfortunately, most of the run-time type information has already been removed by the compiler, and the resulting bytecode relies on erased types. A third option is to perform only a minimal amount of direct modifications to the Java Compiler and then perform the remaining transformations on the bytecode level. In this implementation, the NextGen compilation phase encodes the runtime type information needed for bytecode translation. This frees NextGen from complete depending on the javac compiler, and the quarks and complexities of its data structures and APIs.

The NextGen2 compiler follows the third approach listed above. Details are provided in the following sections.

4.2 NextGen2 Compilation Model

From the user's point of view using NextGen2 requires a minimial adjustment from their current java model: The command `ngc` replaces `javac`, and `nextgen` replaces `java`. The compilation of generic java code generates extra class files for the NextGen2 templates.

Internally, NextGen2 adds two stages of processing to the normal Java 5 compilation process: a "type flattening" stage to encode parametric types, and a "snippet patching" stage to collect snippetized type-dependent operations. Both stages follow a mutation-based Visitor pattern to destructively transform segments of code. Since unrelated code is unaltered, this approach eliminates the need for a post-processing stage to patch jump targets used in the original NextGen compiler. In conjunction, these two stages snippetize type-dependent operations and generate the template classes used to represent uninstantiated parametric classes and methods environments.

A template class has strings in its constant pool that contain embedded references to the type parameters related to the class's instantiation. These references specify

an index of the form {0},{1}, etc, referencing a class-level parameterization. The NextGen2 classloader replaces these references with the actual type parameters (represented as mangled strings) to generate specific template instantiations.

All classfiles genered by NextGen2 contain a special NextGen Attribute for versioning. This versioning marker is orthogonal to the major and minor version used to mark JVM compliance. This allows NextGen2 to distinguish between different versions of NextGen2 modified files, and also other third-party modifications.

The next three sections will explain the details of type flattening, snippet patching, and classloading.

4.3 Type Flattening

NextGen2 performs type flattening after type checking. In this stage, NextGen2 replaces parametric types with their NextGen mangled representation, and snippetizes type-dependent operations. The basic type flattening scheme is as follows:

1. Recursively traverse the code.

2. Encode type-dependent operations into a snippet method call. NextGen2 hashes the generated method names to ensure uniqueness.

3. Generate the corresponding snippet method in the related template class. For class level snippets, NextGen2 stores the snippet in the class's `template class`. For parametric methods, NextGen2 stores the snippets in the corresponding `snippet enviornment`.

4. When translating polymorphic methods, prepend the method parameters with the corresponding `snippet environment` interface.

5. On method invocations of polymoprhic methods, generate code to instantiate a `snippet environment` and pass it as a parameter into the method call.

After traversing the code, NextGen2 stores the newly generated template classes for parametric classes and the `snippet environments` for polymorphic methods.

4.3.1 Encoding Parametric Types

NextGen2 must generate distinct class identifiers for the templates used to encode runtime type information. The class identifiers include the character sequence "$$", which by convention does not appear in Java source code. The use of two "$" characters distinguishes NextGen2 classes from mangled inner classes which use a single "$". For a generic class `A<S,T>`, NextGen2 creates the following two additional classes:

- `A<S,T>`, the light-weight template class
- `A<S,T>$`, its corresponding typing interface

where S, T are fully quantified class identifiers. All references to generic types are then encoded to valid java identifiers using the following translation scheme:

- Left angle bracket '<' to "$$L"
- Right angle bracket `'>'` to "$$R"
- Comma ',' to "$$C"
- Period (dot) "." to "$$D"

So the class

```
Pair<java.lang.String, java.lang.Integer>
```

would be encoded as:

```
Pair$$Ljava$$Dlang$$DString$$Cjava$$Dlang$$DInteger$$R
```

The encoding for `snippet environments` builds on the encoding for generic types. A polymorphic method `m<T>` in class `A<S>` generates two classes: a `snippet environment` and its corresponding interface. While the previous chapter used the identifiers `[ClassIdentifier]$env` and `[ClassIdentifier]$env$`, the precise naming scheme is:

- `A`≤`S`≥`<bounds(T)>$$E<T>`, a `snippet environment`
- `A`≤`S`≥`<bounds(T)>$$E`, its corresponding typing interface

where `bounds(T)` are the full class identifiers for the upper bounds of the type parameters T. The indentifiers are then encoded using the following translation scheme:

- Left small angle bracket '≤' to "$$l"
- Right small angle bracket '≥' to "$$r"

and `$$E` can be either:

- "$$ES" denotes a static method environment
- "$$ED" denotes a dynamic method environment

Recall the dynamic `zip` example introduced in section 3.5. The the call site:

```
List<Pair<Integer, String>> p = i.<String>zip(s);
```

will produce the following snippet environment and interface:

```
List<java.lang.Integer>≤java.lang.Object≥$$ED<java.lang.String>
List<java.lang.Integer>≤java.lang.Object≥$$ED
```

which is encoded as:

```
List$$Ljava.lang.Integer$$R$$ljava.lang.Object$$r$$ED$$Ljava.lang.String$$R
List$$Ljava.lang.Integer$$R$$ljava.lang.Object$$r$$ED
```

As shown above, the NextGen2 encoding scheme for polymorphic methods generates snippet environments based on the enclosing class and the type parameters in scope. Therefore, methods that have the same type parameters, and probably a related function, share a common `snippet environment`.

4.4 Snippet Patching

After NextGen2 flattens parametric types and snippetizes type-dependent operations, a post-processing "snippet patching" stage traverses class hierarchies and propagates snippets in related templates.

Since a class inherits code from its superclass, snippets in parametric classes must be propagated down the class hierarchy. If the type-dependent operation is defined in a polymorphic method, the snippets related to the method maybe needed in the `snippet environment` of the superclass. This of course depends on the typing relationship between the snippet classes as discussed earlier in section 3.5.1. The example used in this earlier section involved a class `Revlist` that overrided a polymorphic method `zip` defined in its superclass `List`. If these two classes used separate `snippet environments`, all the snippets in `RevList$env$` would be propagated to `List$env$`.

4.4.1 Snippet Environment Representation

NextGen2 generates an auxiliary class files for each template class. Therefore, each polymorphic method generates one class file for its snippet environment and another for the snippet environment interface.

Since all NextGen2 templates are extremely light weight, an alternative representation could be to store snippet environments as attributes in the original base class. While there are variations of this scheme, the primary objective would be to trade disk access for load-time computation. The former would require loading additional classfiles, while the latter requires a more sophisticated classloader to perform code generation.

4.4.2 Implementation

Snippet patching uses the Byte Code Enginnering Langauge (BCEL) to operate on the bytecode generated by the compilation process. Manipulating snippets at the bytecode level provides a more simplistic interface over an AST-based algorithm. To copy a snippet method from one template to another, the Snippet Patacher needs to make a single pass over the constant pool and then copy each method and its related constants to the destination template.

The algorithm for propagating class-level snippets follows a straight-forward descending search specified in [2]. Support for polymorphic methods uses a more intricate, upward search based on method overrides. This search is used to determine the class hierarchy of the newly-compiled `snippet environments`, and to propagate snippets downward through these hierarchies. The algorithm performs the following:

1. Search the the list of recently compiled classes for polymorphic methods. The first parameter in the method's signature indicates the `snippet environment` generated by type flattening. The pair (method, snippet env) is hashed for future reference.

2. For each method, check if the method definition is new, or if it implements or overrides a definition in a super type. If an earlier definition exists, hash the pair (super env, sub env). These results define the class hierarchies of the newly generated `snippet environments`.

3. Then, for each distinct `snippet environment`, propagate the snippets downwards using the hierarchies from the previous step.

4. For each pair in (super env, sub env), check if the sub environment interface is defined as an interface to the super environment interface. If it is not, add it.

5. Save each completed `snippet enviornment` and interface.

The algorithm above uses a subtle property of the classloader: Elements in the classpath shadow earlier definitions. At compilation, `ngc` writes new versions of each `snippet environment` to the build directory. Storing new versions in the build directory is ideal since there is no guarantee that existing libraries or jars are writable. This strategy works since the classpath used for compilation is nearly identical to the one used for execution. Thus, more recent `snippet environments` definitions shadow earlier definitions.

4.4.3 Cross Package Instantiation

The NextGen2 design presented in chapter 3 does not specify where the instantiation classes for `snippet environments` are placed in the Java namespace. The most intuitive location would be in the same package.* However, this placement can cause a cross-package instantiation problem where a private type is used by a `snippet environment` that resides outside the package boundary[10].

The simplest solution to this instantiation problem is to automatically widen a private class to public visibility if it is passed as a type argument in the instantiation of a generic type residing in another package. This tradeoff for simplicity at the cost security has precedent in Java. When an inner class refers to private members of its enclosing class, javac automatically widens the visibility of the relevant members by generating getters and setters with package visibility. Although more secure implementations are possible, the Java designers decided to sacrifice some visibility for the sake of performance.

One solution proposed in [4] is to use an initialization protocol to pass a environment containing the necessary snippets to the constructor of the instantiation `snippet environment`. However, in NextGen2, `snippet environments` are themselves containers with minimal typing restraints: each environment instance implements a single typing interface. Thus, the actual location of the snippet environment

*A static inner class would be ideal, but they are translated into toplevel classes by the compiler.

is not important. So the NextGen2 compiler can widen the private class to package visibility and then instantiate the snippet environment in the same package as the private class. This approach requires the same number of snippet environments as simply widening visibility from private to public. Therefore, NextGen2 can support cross package instantiation problem for polymorphic methods without any additional overhead.

4.5 Pathological Reflection Implementation

As discussed in section 3.5.3, the reflection code is located in a bridge method defined in the template class of a parametric class. The reflection code must infer the polymorphic types of the relevant receiver and `snippet environment`. To minimize the impact of reflection in pathological method invocations, the NextGen2 `ClassLoader` includes a `HashMap` of `snippet environment` class identifiers to their singleton objects. Shown below is the core functionality to support pathological polymorphic methods.

```
String senv  = snippetEnv.getClass().toString();
String clazz_suffix = clazz.substring(senv.indexOf("$$l"), clazz.length());
String s4 = "$$SNIPPET_STRING$$" + s1;
Object env = edu.rice.cs.nextgen2.classloader.
                        NextGenLoader.reflectionTable.get(s4);
if (env != null) {
    return meth2( (Integer) env, a1, a2);
}
env = Class.forName(s4).getDeclaredField("ONLY").get(null);
edu.rice.cs.nextgen2.classloader.NextGenLoader.reflectionTable.put(s4,env);
return meth2( (Integer)env, a1, a2);
```

where `"$$SNIPPET_STRING$$"` is filled in with the identifier of the current template class.

4.6 NextGen2 Classloader

At runtime, the NextGen2 classloader intercepts JVM requests to load a class with a mangled identifier. It then loads the corresponding template files to generate the requested class or interface. It searches the identifiers in the constant pool replacing each reference tag {n}, where n is an integer, with the corresponding type specified by the mangled class indentifer. The NextGen2 classloader sticks closely to the original NextGen classloader defined in [2].

Instantiating a `snippet environment` used by a polymorphic method follows the same protocol as that of instantiating a class used to represent a generic type. The only caveat pertains to providing support for the pathological case of polymorphic invocation. In this case, the NextGen2 classloader must also replace {n} occurances in the `String` used by the reflection code. This corresponds to the `"$$SNIPPET_STRING$$` placeholder used In the reflection code shown in section 4.5.

Chapter 5

Related Work

To properly qualify the mildly heterogeneous approach used in NextGen, its necessary to look at other research in generics and polymorphic methods. The current research falls into three basic groups: 1) heterogeneous translations, 2) homogenous translations, and 3) adding extensions to the underlying runtime virtual machine.

5.1 Heterogeneous Translations

The first support for generics in Java was developed by Agesen, Freund, and Mitchell[1]. Their approach followed a heterogeneous technique similar to the preprocessing done with C++ templates. To support the Java compilation model, they pushed the generation of template instances from linking, a stage absent in the compilation of Java source, to load time. While this reduces the number of classfiles necessary at runtime, it significantly slows down runtime performance. Any purely heterogeneous translation must generate complete, exact copies of a template per each distinct instantiation type. This code explosion also hinders potential performance optimization by the Just-In-Time (JIT) compiler. Furthermore, this simplistic approach breaks intuitive subtyping relationships between generic instantiations.

In regards to polymorphic methods, their initial research excluded any implementation or design.

5.2 Homogenous Translations

While the GJ erasure-based infrastructure is insufficient to propagate runtime type information, a homogenous translation proposed by Viorli and Natali, called LM

(Load-Time Management), uses the Java Reflection APIs to carry parametric type information[12]. LM supports parametric methods by passing an extra parameter on parametric method invocations, and then using reflection to implement type-dependent operations.[11]. The extra parameter indexes into a `Virtual Parametric Method Table` (VPMT) associated with every potential, dynamic receiver type, and stores the parametric type information available at the call site.

While the use of reflection in LM avoids code duplication, its results in a significant performance penalty. Implementing parametric types through reflection creates a heavy-weight representation of types. The benchmarks in Chapter 6 includes an analysis comparing LM's homogenous approach to NextGen2's heterogeneous one.

5.3 Modifications to JVM

The PolyJ extension of Java, developed at MIT, provides runtime support for generic types by modifying the underlying Java Virtual Machine (JVM)[9]. They propose adding two additional bytecode operations to provide the framework necessary to maintain runtime types. While their research focuses on parametric classes, an extension to support polymorphic methods could use these two bytecode operations and/or possibly other modifications to the Java bytecode or JVM. From a design perspective, these modifications are necessary to cleanly reflect the abstraction introduced by generic types. However, any dichotomy in the runtime VM would violate the Java paradigm of "Write Once, Run Anywhere". The version of PolyJ distributed on their website maintains compatibility with the JVM by using a heterogeneous approach similar to NextGen2. It uses "`trampoline`" classes to encode runtime type information and perform type-dependent operations.

A critical evaluation of these techniques used in Generic Java can be found in research to support generic types on the .NET Common Language Runtime(CLR) by Kennedy and Syme[6]. Since they were free to modify the CLR, their design is less constrained with compatibility and more with efficiency. Surprisingly, their

implementation is very similar to NextGen's base class code-sharing and lightweight template classes. They follow a mainly homogenous approach to maximize code and representation sharing across instantiations, but use distinct `vtables` for each generic type instantiation. The `vtable` is also modified to include a dictionary of type handlers used to store the instantiated type parameters and type-dependent operations. This dictionary of type handlers is used also to support type instantiations of static polymorphic methods. However, their current implementation does not support dynamic polymorphic methods.

PolyJ and C# provide support for generic types and polymorphic methods at the JVM level. LM supports polymorphic methods using Java Reflection; In other words, maintaining types at the meta-level. In contrast, NextGen is implemented as close as possible to the JVM level, without violating the Java "Write Once, Run Anywhere" paradigm. This approach provides efficient support for polymorphic methods without breaking backwards compatibility.

Chapter 6

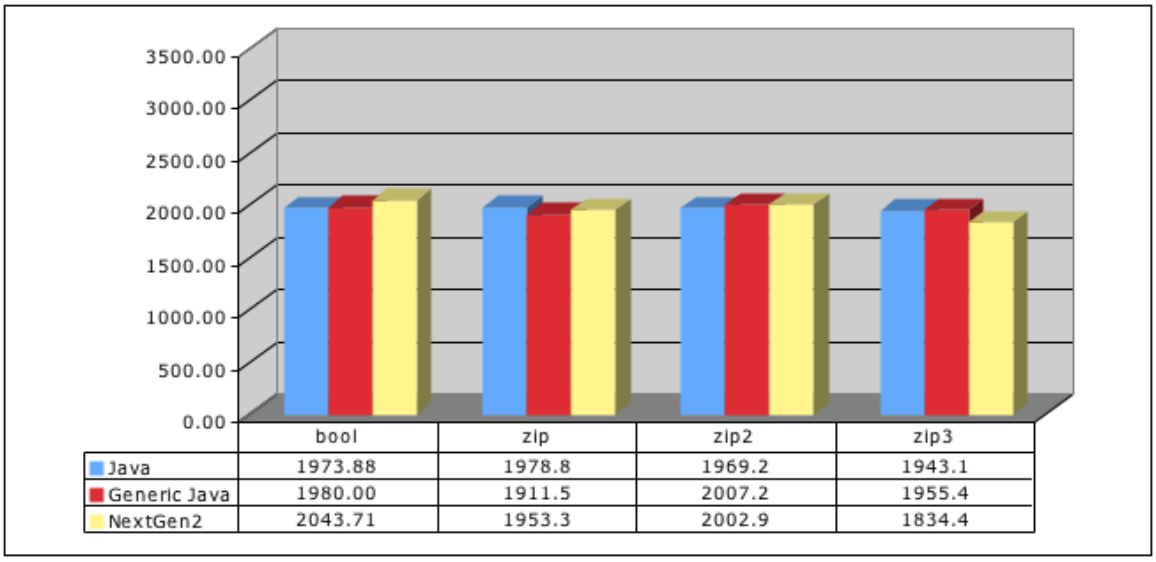
Performance

The NextGen2 benchmarks measure the overhead of supporting polymorphic method in the context of first-class genericty. Since no established benchmark for Generic Java currently exists, we had to develop our own specialized benchmark to analyze the performance of NextGen2. Each test was engineered in Generic Java. The benchmarks were then hand-translated into an equivalent non-generic Java source to provide a fair comparison with standard Java code. While the JSR14 compiler has an option to output non-generic source code, the output is not necessarily valid Java source code. The benchmarks consists of the following programs, all of which involve generics:

- **Bool:** A Boolean expression simplifier. This program parses in a large number of boolean expressions into an AST and simplifies them. The expressions are simplified through a series of passes by a generically typed visitor. This benchmark consists of 730 lines of code in 25 classes, 7 of these classes make heavy use of generics. While all the generically type visitor makes extensive use of polymorphic methods, it does not perform any type-dependent operations.

- **Zip1** Small program that repeatedly constructs small lists of 100 elements, and then zips them together. This case analyzes the initial overhead of calling a polymorphic method. This test consists of 80 lines of code in 3 classes, 2 of these make heavy use of generics and polymorphic methods.

- **Zip2:** Small program that repeatedly constructs lists of 1000 elements, and then zips them together. This case shifts focus more toward the performance of



	bool	zip	zip2	zip3
Java	1973.88	1978.8	1969.2	1943.1
Generic Java	1980.00	1911.5	2007.2	1955.4
NextGen2	2043.71	1953.3	2002.9	1834.4

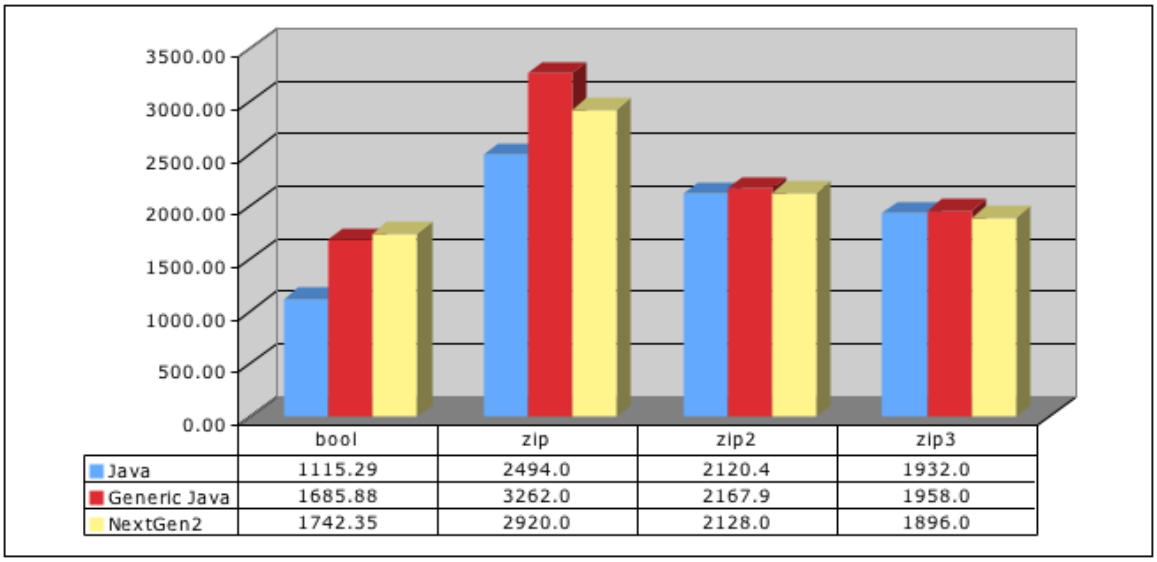
Figure 6.1 : General Performance Results (ms)

recursive methods. This test consists of 80 lines of code in 3 classes, 2 of these make heavy use of generics and polymorphic methods.

- **Zip3:** Constructs a single pair of 1000 element lists, and then repeatedly zips them together. This case analyzes the potential of current Just-In-Time (JIT) optimzations. This test consists of 80 lines of code in 3 classes, 2 of these make heavy use of generics and polymorphic methods.

- **Reflect1:** Baseline test analyzing the performance of the case of pedagogical reflection. It constructs a list of pairs, and calls a functional operation `changeSecond` to change the type of the second element in each pair. This test consists of 100 lines of code in 4 classes, 3 of these make heavy use of generics and polymorphic methods.

- **Reflect2** Second test analyzing the performance of the case of pedagogical reflection. This test inserts a 5% probability of pedagogical reflection. It performs



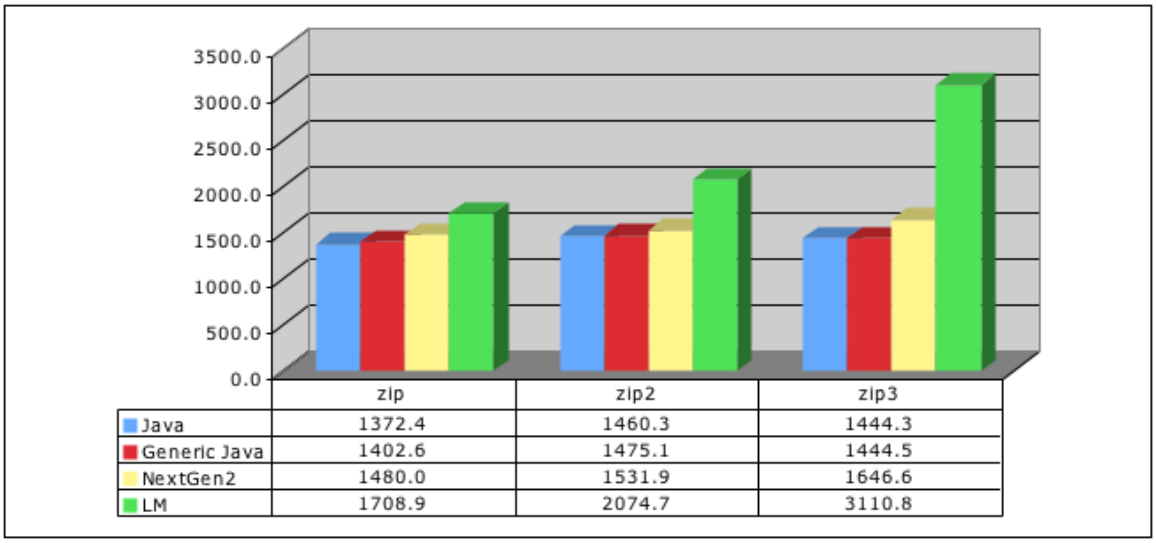
	bool	zip	zip2	zip3
Java	1115.29	2494.0	2120.4	1932.0
Generic Java	1685.88	3262.0	2167.9	1958.0
NextGen2	1742.35	2920.0	2128.0	1896.0

Figure 6.2 : First Iteration of General Performance Results(ms)

just like **Reflect1**: It constructs a list of pairs, and calls a functional operation `changeSecond` to change the type of the second element in each pair. Approximately 5% of the list are a subtype of pair that introduces an additional class-level type parameter. This test consists of 100 lines of code in 4 classes, 3 of these make heavy use of generics and polymorphic methods.

- **Reflect3** Third test analyzing the performance of the case of pedagogical reflection. This test inserts a 5% probability of pedagogical reflection. It performs just like **Reflect1**: It constructs a list of pairs, and calls a functional operation `changeSecond` to change the type of the second element in each pair. Approximately 10% of the list are a subtype of pair that introduces an additional class-level type parameter. This test consists of 100 lines of code in 4 classes, 3 of these make heavy use of generics and polymorphic methods.

- **JSR14 2.2** The task of compiling a compiler represents a very robust use of generics. The JSR14 compiler uses a series of Tree generic visitors to compile



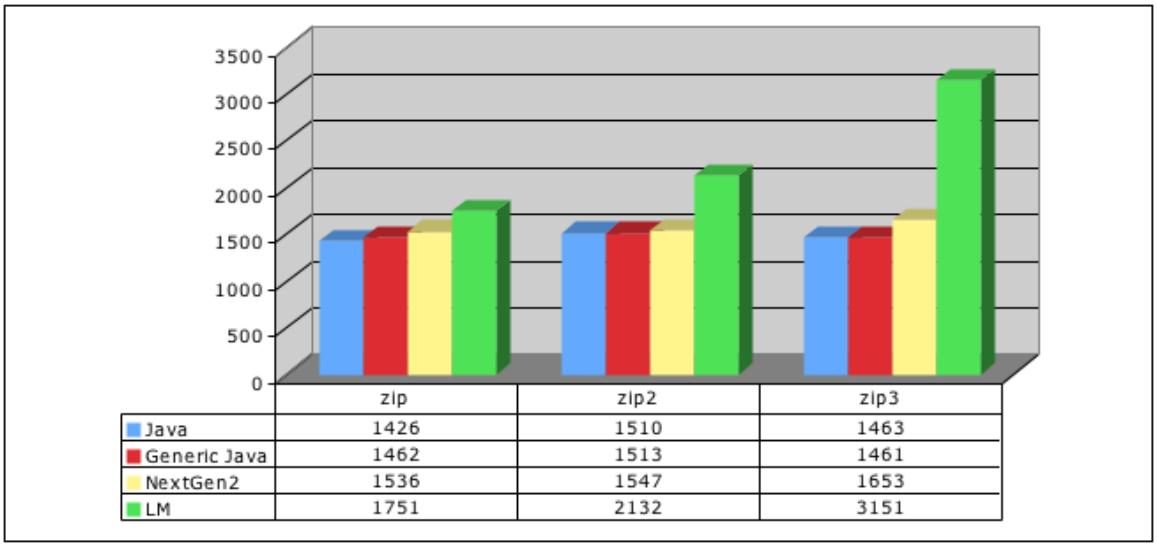
	zip	zip2	zip3
Java	1372.4	1460.3	1444.3
Generic Java	1402.6	1475.1	1444.5
NextGen2	1480.0	1531.9	1646.6
LM	1708.9	2074.7	3110.8

Figure 6.3 : Performance of Polymorphic Method Recursion (ms)

a Java source program to bytecode. The JSR14 compiler consists of approximately 25,784 lines of code in 225 classes. Nine classes make light use of generics, 59 classes make moderate use of generics, and 8 of these make heavy use of generic type and polymorphic methods.

Each benchmark was executed twenty-one times using the Sun 1.4.1 server Java Virtual Machine (JVM) on a 2 Ghz, Intel Pentium 4 with 512 MB of RAM running Redhat Linux 8.0. The NextGen2 benchmarks focus on the server JVM since the client JVM does not perform any Just-In-Time (JIT) optimizations. The expectation is that by design, the NextGen2 code augmentations can be offset by specialized JIT optimizations.

The first iteration of each benchmark was dropped because it deviated significantly from the remaining 20 iterations. We believe this deviation results from the overhead of JVM startup and initial JIT compilation of the code, both of which are not directly relevant to what we are trying to measure. After dropping the first run, the variance



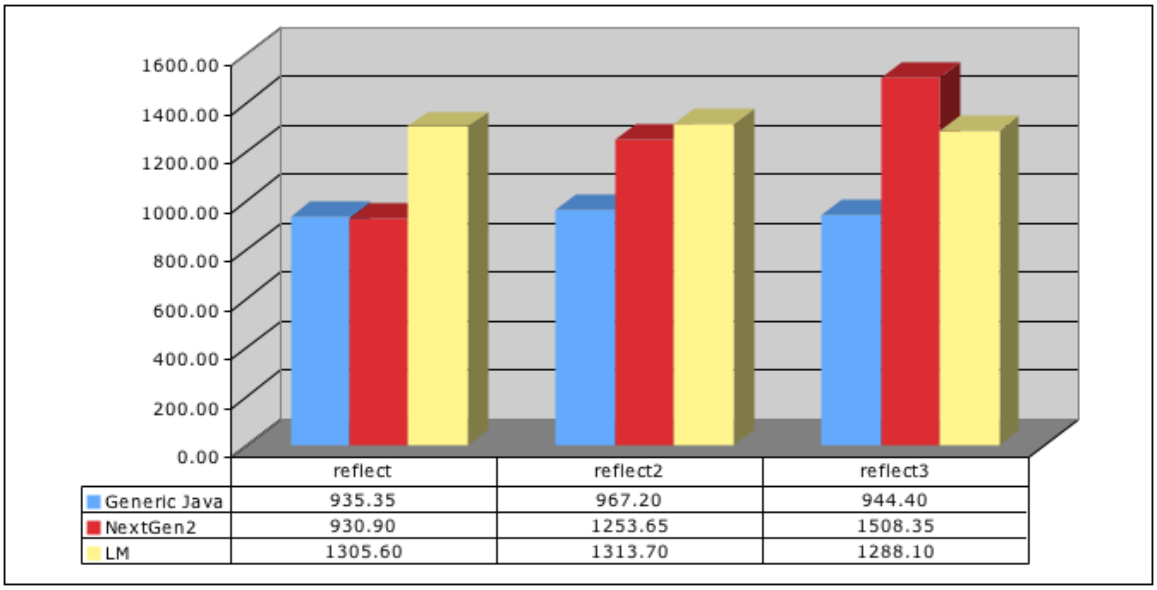
	zip	zip2	zip3
Java	1426	1510	1463
Generic Java	1462	1513	1461
NextGen2	1536	1547	1653
LM	1751	2132	3151

Figure 6.4 : First Iteration of Performance of Polymorphic Recursion (ms)

among iterations for each benchmark was less than 10%. For thoroughness, this thesis also includes performance metrics of the first iteration of each test.

The slower performance of NextGen2 reflects the overhead of polymorphic methods, and also the overhead of generic instantiation classes. This is to be expected since an implementation of polymorphic methods under first-class genericity requires a parametric representation of types. In general though, the majority of the operations in a program will not depend on runtime type information, and therefore, support for first-class genericity is not costly when amortized over a large number of instructions [12].

Figures 6.1-6.2 show that, given the current level of JIT optimizations, NextGen2 performs competitive to Java and Generic Java. Surprisingly, the results for `zip3` show a 5% performance gain in the case of recursive polymorphic methods. An analysis of the metrics for zip1, zip2, and zip3 suggest this speedup results from JIT optimizations. In comparison to zip1 and zip2, the zip3 benchmark focuses only on the recursive invocation of zip; successive iterations reuse the same set of `Lists` to



	reflect	reflect2	reflect3
Generic Java	935.35	967.20	944.40
NextGen2	930.90	1253.65	1508.35
LM	1305.60	1313.70	1288.10

Figure 6.5 : Performance of Pedagogical Reflection (ms)

invoke `zip`. Since the trend in the NextGen2 performance differs from that of GJ or Java, its safe to assume that the NextGen2 translation provides a basis for new JIT optimizations.

Figures 6.3-6.4 compares NextGen2 against LM using the tests zip1,zip2, and zip3. Because of the larger space requirements for classes in LM, the structure of the test zip3 was rewritten to improve garbage collection during execution. This change is reflected in the different results for zip3 shown in figures 6.1-6.2 and 6.3-6.4.

Since these zip benchmarks use only new operations and polymorphic method invocations, they provide a fair comparison between NextGen2 and LM.*. LM new operations require an extra parameter to store runtime types, while NextGen2 must load specialized instantiation classes. Polymorphic method invocation in LM requires passing a method descriptor (an integer index). This should be as fast, if not faster

*Instanceof tests in LM require an O(n) worst case iteration, where n is the level of parameterization. NextGen2 can perform instanceof tests in O(1) time.

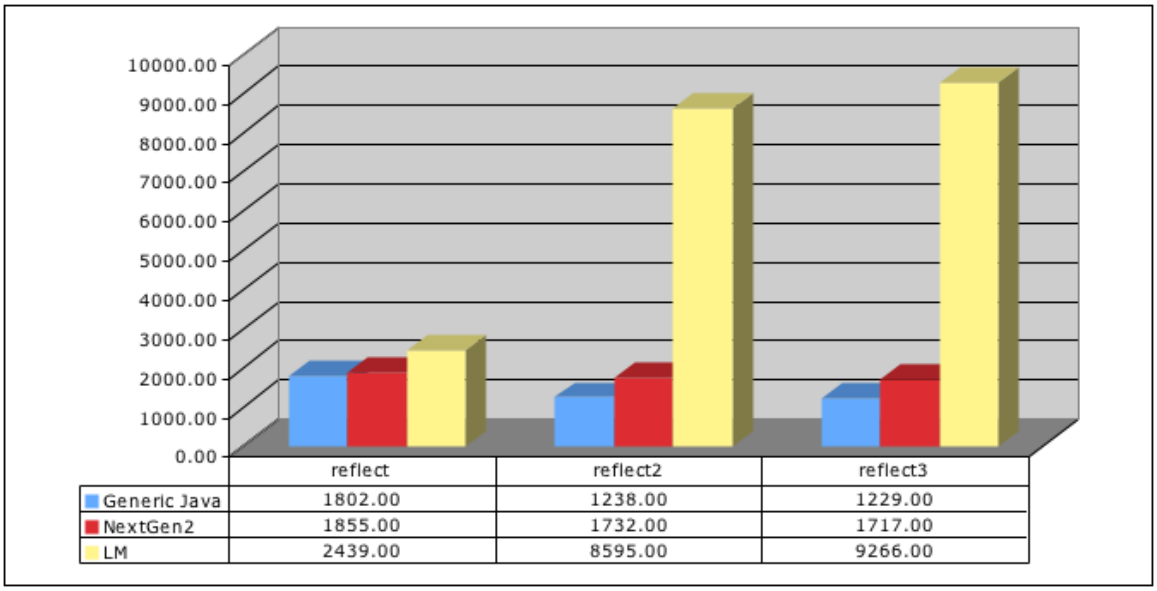


Figure 6.6 : First Iteration of Performance of Pedagogical Reflection (ms)

than passing a local final field in NextGen2.

In general, NextGen2 performs at least 20% faster than LM. The overhead of the LM approach can be attributed to two factors. First, LM classes maintain a reference to a `type descriptor` object that stores generic type informatiion. For a small collection class like `Pair` with only two fields, this increases object size by 50%. Second, the dynamic nature of LM preempts many JIT optimizations. During the execution of zip, a new `Pair` is created on each call. At each constructor invocation, LM must perform a lookup to retrieve the `type descriptor` for the new object, and thus determine its type parameterization:

```
((MD)td.MDs[0].elementAt(p)).friendTD[1]
```

Although this value never changes, LM must recompute it at each iteration. NextGen2 on the other hand, relies on static class identifiers, allowing for potential JIT optimizations.

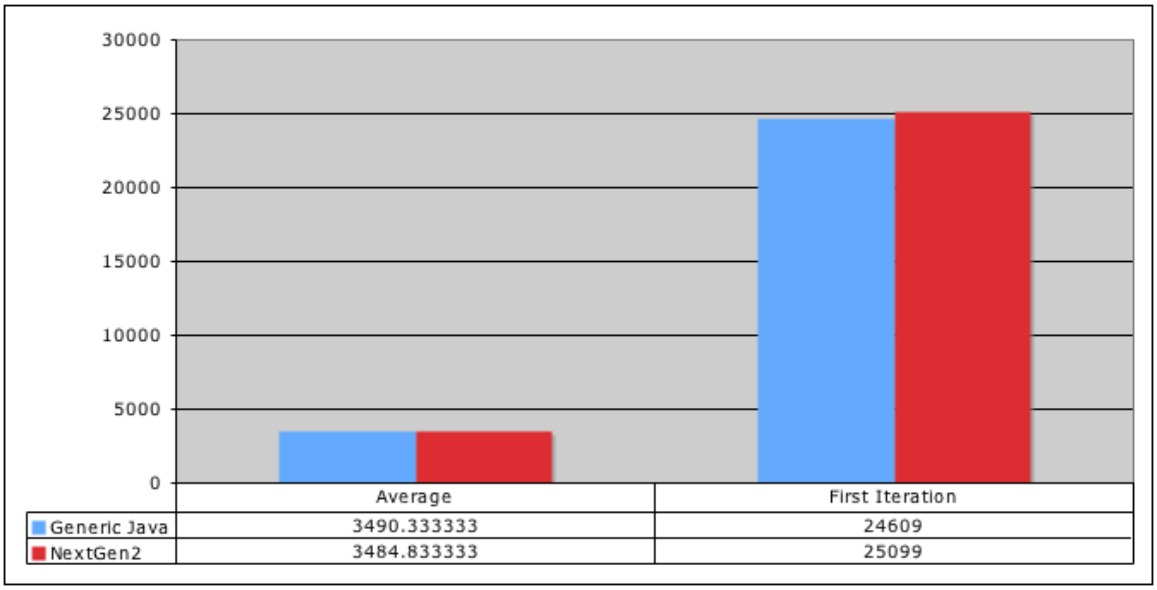


Figure 6.7 : JSR Performance Results(ms)

In the case of zip3, LM requires twice the time as GJ or NextGen2. A possible explanation for the extremely high overhead is that the higher space needs of LM resulted in some objects being stored in virtual memory and swapped to disk.

Figures 6.5-6.6 shows how the performance of NextGen2 degrades in the pedagogical case. NextGen2 performance degrades linearly since reflection code is executed in the bridge methods for a polymorphic method. Figure 6.5 implies that NextGen2 produces an inefficient implementation when the probability of reflection is greater than 5%. On the average though, NextGen2 is still more efficient than LM when the probability of reflection is less than 7-9%. An analysis of only the first run performance, shown in figure 6.6, show a perplexing result. LM performs 4-5 times slower than NextGen2 and GJ in the tests reflect2 and reflect3. A conjecture is that constructor supercall chains are inefficiently handled in LM; each supercall stores a reference to the "farther" of the current type descriptor. Since the exact cause of this slowdown cannot be determined by this benchmark alone, a more in-depth analysis

will be included in future work.

The JSR14 benchmark in figure 6.7 shows that a JSR14 compiler compiled with NextGen2 performs competitive to a GJ version. This result affirms the belief that the cost of maintaining runtime type information, and thus supporting first-class genericity in Java, is not costly in large applications, since it is amortized over a large number of operations.

Chapter 7

Conclusion

This thesis has presented a comprehensive design and a solid implementation of first-class polymorphic methods in Generic Java using the NextGen compiler framework. The NextGen2 design supports static and dynamic polymorphic methods with minimal runtime overhead. Most importantly, NextGen2 support for first-class genericity works on existing Java Virtual Machines and is compatible with all legacy code. While previous versions of NextGen were mainly proof of concept, the latest version provides a complete, production quality alternative to the current erasure-based, second-class genericity found in Java 5.

7.1 Future Extensions

A comprehensive implementation of polymorphic classes and methods provides a framework for a richer level of genericity than currently possible in Generic Java. This section discuss some of these features, and how the NextGen2 compiler can be extended to support them.

7.1.1 Performance Optimizations

Since NextGen2 now has a complete implementation of Generic Java, it's an opportune time to reflect and improve on our current implementation. On a high-level, we need to see if hashing, or other space-time tools, can speed up code in the compiler, classloader, and the template classes generated by NextGen2. On a low-level, we need to explore the internals of the JVM, and research possible JIT optimizations in NextGen2.

7.1.2 Autoboxing of Primtives

One of our long range goals is to unify Java's disjoint typing system of value types and reference types. The JSR201[8] specification for autoboxing uses the defacto heavy-weight wrapper classes: Integer, Float, Double, etc. This implementation ties autoboxing not to an interface, but to an actual implementation, thus eliminating the ability to determine an autoboxed constant from a manually boxed one. Therefore under this approach, it is impossible to provide intuitive semantics for the double-equals notion of equality on primitives. NextGen2 provides and opportunity to implement autoboxing in an intuitive way. Our approach would be to use light-weight wrapper classes for autoboxing: JInteger, JFloat, JDouble, etc. The distinction between JInterger and Integer allows us to provide a double-equals semantics consistent with primitive equality. For example:

```
if (a == z)  ...
```

could be translated into the following:

```
if (  (a instance of JPrimitive)
      ?  a.equals(z)
      :  a == z  )
```

7.1.3 Primitives as Type Parameters

NextGen2's heterogeneous implementation of generic types provides hooks that allow the inclusion of primitives as type parameters. A naive approximation would be to use a primitives boxed representation in current generic templates. A more sophisticated approach would be to provide optimized versions of generic code to take advantage of primitive bytecode operations.

Since Java has only eight primitive types, mapped to only four distinct JVM representations, code explosion would be minimal. To minimize code duplication, the NextGen2 classloader could specialize the standard class template at load time. In

any case, any space concerns are minor in comparison to the gains resulting from the elimination of autoboxing constructs.

Most importantly, more research is needed in determining intuitive subtyping between generic classes using primitives and those without. For example, should List¡int¿ be considered a subtype of `List<Object>` and `List<Integer>`? In both cases, NextGen2 would need to generate bridge methods that would autobox a primitive value into the Object world.

Despite these issues, the ability to abstract classes and methods over all types, and to perform meaningful operations on these parametric types would prove a boon to developers.

7.1.4 Mixin Classes

While NextGen2 provides support for runtime type-dependent operations, it currently does not provide complete support for every intricacy of first-class generictiy. Specifically, the NextGen framework disallows mixins, classes that extend one of their type parameters. Mixins provide a disciplined alternative to multiple implementation inheritance. The mixin class itself defines a uniform class extension that can be applied to any class that fulfills its inheritance requirements. Although similar results can be obtained using object composition (e.g., the Decorator pattern [5]), mixins provide precise typing and a more accurate interface. For example, a decorated object can be typed only against the original decorator class. Mixins, on the other hand, can be typed against both the mixin class and the mixed-in parent class. This essentially provides a mechanism to add incremental functionality to existing classes. A mixin that time stamps object creation is shown below:

```
class TimeStamped<T> extends T {
  long time;
  TimeStamped() {
     super();
```

```
    time = System.currentTimeMillis();
}
```

Most importantly, mixins provide the infrastructure to support a true module system in Java. Currently, the Java package system uses hard-coded references to specific external class names. These references prevent the reuse of a package in different contexts, and prevent programmers from using or even testing a package in isolation. With mixins, a programmer can remove these hard-coded references, and connect modules parametrically. In other words, mixins allow an application to be decoupled into logically independent components. Thus, a module's contextual requirements could be completely captured by its visible interfaces.

Bibliography

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 49–65, Atlanta, GA, 1997.

[2] E. Allen, R. Cartwright, and B. Stoler. Efficient implementation of run-time generic types for Java. In *IFIP WG2.1 Working Conference on Generic Programming*, 2002.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[4] Robert Cartwright and Guy L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Vancouver, British Columbia*, pages 201–215. ACM, 1998.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] A. Kennedy and D. Syme. Design and implementation of generics for the .net common language runtime. In *PLDI*, 2001.

[7] Sun Microsystems. Sun Microsystems, Inc. JSR 14: Add generic types to the Java Programming Language, 2001.

[8] Sun Microsystems. Sun Microsystems, Inc. JSR 201: Extending the Java Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import, 2004.

[9] A. Myers, J. Bank, and B. Liskov. Parameterized types for Java. In *POPL*, 1997.

[10] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[11] Mirko Viroli. Parametric polymorphism in java: an efficient implementation for parametric methods. In *Selected Areas in Cryptography*, pages 610–619, 2001.

[12] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. *ACM SIGPLAN Notices*, 35(10):146–165, 2000.