LOOJVM: Type Checking Java Language Extensions in the Virtual Machine

by

Robert N. Gonzalez

A Thesis
Submitted in partial fulfillment of the requirements
for the Degree in Bachelor of Arts with Honors
in Computer Science

WILLIAMS COLLEGE
Williamstown, Massachusetts

May 26, 2003

Abstract

Since Java's release in 1996, researchers have been exploring ways to improve its type system. On feature that has received a great deal of consistent attention is F-bounded parametric polymorhpism [CCH+89]. After years of debate, Sun Microsystems is finally adding support for parametric polymorphism to the Java language specification in its next major release.

However, there are other type system enhancements that add expressiveness to Java. LOOJ, a language extension to Java developed at Williams by Kim Bruce and his students, includes support for F-bounded parametric polymorphism as well as for exact types and the `ThisType` and `ThisClass` constructs. Foster's undergraduate thesis [Fos01] explained how these language extensions together add considerably to the expressiveness of the Java language.

Until now, all proposed modifications to the Java language specification have had to work around the inflexible type system of its target platform, the Java Virtual Machine. For example, GJ's compiler inserts extra type casts and NextGen produces a complicated type hierarchies to convince the Java bytecode verifier that the bytecode they produce for their Java extensions is safe to execute in the JVM.

This thesis presents LOOJVM, a modified JVM that is able to efficiently run code produced by the LOOJ compiler. LOOJVM includes an enhanced verifier whose static type system is the same as the LOOJ programming language and allows code lacking the traditional superfluous type casts to be verified and run safely. LOOJVM also optimizes bridge method calls for greater efficiency.

Acknowledgements

I cannot imagine this thesis would have been successful without the help of my advisor, Kim Bruce. His knowledge and encouragement, the occasional gentle push he provided, and, above all, the patience and understanding he has shown throughout this process for my many mistakes, my 11th hour work habits, and for me in general, gave me the ability to continue the project when I otherwise could not. Thank you, Kim.

I send my thanks out to Steve Freund, my second reader. I attempted to make his job as difficult as possible by submitting unforgivablely late drafts, yet he still responded with invaluable feedback.

This work largely builds on the previous work of John Foster '01 (a.k.a. Nate), to whom I owe considerable gratitude. Regardless of how busy his life was at any time this past year, he was always available to answer my often silly questions on LOOJ's type system and on his implementation. I must also thank him for writing extremely legible source code in the LOOJ compiler. It really made my life much easier than it could have been and I can only hope to produce such nice code for the next person to work on this project.

I would also like to thank the people on the Kaffe[1] development team (kaffe@kaffe.org) and Dalibor Topic in particular. I could never have figured out how to modify the virtual machine in any reasonable amount of time without their help.

Finally, I want to thank my friends and family for understanding why I stayed on campus during dead week to program, for being patient with me when sleep deprivation had greatly diminished my ability to be nice and fun, and for always cheering me up when things were not going as well as I would hope.

[1]Kaffe is a trademark of Transvirtual Technologies.

Contents

Chapter 1

Introduction

We shall not cease from exploration and the end of all our exploring will be to arrive where we started and know the place for the first time.

– T.S. Eliot

The Java[1] Programming Language is one of the most popular programming languages in use today. Since its release in 1996, numerous proposals have been made to improve its type system. One feature that has received considerable attention from researchers is parametric polymorphism. In many common programming situations, such as the use of generic data structures, parametric polymorphism has many advantages over dynamic approaches, which usually require the programmer to insert type casts whenever accessing data in the structure. These advantages include "safety (more bugs caught at compile time), expressivity (more invariants expressed in type signatures), clarity (fewer explicit conversions between data types), and efficiency (no need for run-time type checks)" [KS01, 1].

1.1 Motivation For Change

Unfortunately, without modifying the Java Virtual Machine (JVM) specification, the promise of greater efficiency has yet to be realized. The designs adding parametric polymorphism, or generics, to Java with little or no JVM modification have either decreased the computational speed of the resulting code or increased the amount of space needed at run-time. Both of these tradeoffs are unreasonable as Java runs on many different server platforms

[1] Java is a trademark of Sun Microsystems.

where its speed is already too slow and on a large variety of of portable devices with limited memory. The one approach exploring JVM modification for the purpose of efficiently executing code with generics is PolyJ, designed by Myers, Bank, and Liskov [MBL97]. With generics implemented in the JVM, they found a 17% speedup in some cases and only suffered a 2% slowdown in cases not using generics, though they contend the slowdown is probably due to a lack of time spent on optimization [MBL97]. The speed increase in their modified VM is due largely to a lower number of run-time type casts in the bytecode.

Aside from increased efficiency, a JVM with built-in support for parametric polymorphism is a better platform for other source languages that support generics. Due to its prevalence and the continually increasing speed of modern implementations, the JVM is targeted by compilers for dozens of languages, such as ML, Haskell, Scheme, and Eiffel, many of which have to make sacrifices when translating to Java bytecode. For instance, MLJ (an ML to Java bytecode compiler) does not support functors largely because the JVM lacks support for generics [BKR99].

The inability for these languages to compile easily to Java bytecode will become increasingly important as Microsoft's Common Language Runtime (CLR), a competing virtual platform, becomes more prevalent. Microsoft is marketing the CLR as the de facto virtual machine to be targeted by many important languages, including C# and Java. Moreover, there is already a working version of the CLR that includes support for parametric polymorphism [KS01]. Thus there are advantages to changing the JVM specification beyond increased efficiency for just the Java Programming Language.

1.2 Initial Objectives

We contend that the JVM specification should be enhanced to include support for parametric polymorphism and potentially for other enhancements to its type system, such as `ThisType` and exact types [BPF97]. With that goal, we can create an initial list of criteria that our JVM specification should meet.

- **Conceptually Simple**: It should be easy to make the specified modifications to existing JVMs. Companies and open source groups have invested too much time and money into JVMs to start from scratch.
- **Bytecode Compatibility**: All existing Java class files should run correctly on the enhanced JVM according to the current JVM specification. Therefore some desired enhancements to the Java type system, such as making the current covariant subtyping of arrays illegal, unfortunately should not be considered.

- **Efficient Legacy Code**: All existing Java bytecode should run efficiently on the enhanced JVM. Efficiency means that it should run just as fast (or very close to as fast) as on an unmodified JVM.
- **Efficient Parameterized Code**: Code using new language features should also run efficiently. In this case, efficiency means that it should run *faster* than code containing the casts used in current "generic" data structures.
- **Easy Just-In-Time Compilation**: Any JVM modification should fit into current JIT compilation techniques with little modification to existing JIT compilers. Even a great interpreter cannot outperform a mediocre JIT compiler, so any JVM enhancement that rules out easy JIT compilation is unreasonable.

The underlying idea is that our JVM with an enhanced type system should be able to do everything the old JVM specification allows while providing support for parametric polymorphism and, potentially, for other enhancements to the type system as well.

1.3 Thesis Overview

This thesis proposes a modification to the JVM specification that includes support for F-bounded parametric polymorphism, exact types, `ThisType`, and `ThisClass`[2]. We present a language extension to Java, LOOJ[3], that targets our enhanced JVM, called LOOJVM, and analyze its performance compared to other methods.

In the next chapter we consider other projects that add generics to Java and argue that the design and feature set of LOOJ make it the most amenable to integration with a modified VM. In this discussion, we see how support for `ThisType` and exact types increases the expressiveness of the JVM's type system and should therefore be included in our modified JVM specification.

In the following chapters we discuss the design modifications that enhance the JVM's type system to include these features. We purposefully restrict our modification to the class loader and bytecode verifier.

Finally, we analyze our solution, discuss the state of implementation, and propose future work.

[2] `ThisType` and exact types were first introduced in [BPF97] as a means to solve the binary method problem. This is discussed in Chapter 2.

[3] LOOJ is not an acronym. Instead, it is simply a combination of LOOM, an object oriented language developed at Williams College by Kim Bruce and his students, and Java. The name reflects the fact that LOOJ is a Java extension whose type system is derived from LOOM. LOOJ was previously called Rupiah, the currency used on the Indonesian island of Java. See [Bur98] and [Fos01] for a complete description of LOOJ/Rupiah.

Chapter 2

Contrasting Java Extensions

Understanding the future depends upon understanding the past.

– Janet Moser

Many research groups have proposed adding parametric polymorphism to Java. Most have concentrated on enhancing the source language specification and have avoided modifications to Java's target architecture, the Java Virtual Machine. Changing the JVM specification should be done reluctantly as it would require dozens of commercial, open source, and even silicon implementations to be updated in order to run software taking advantage of any new bytecode language features. Therefore, before committing to a particular extension we must convince ourselves that the current proposals are lacking in some way. Furthermore, we should also consider what other language extensions should be supported in an enhanced JVM.

In this chapter we analyze the most influential designs that add parametric polymorphism to Java and discuss which language extensions should be supported by an enhanced JVM. Some designs, such as PolyJ, already include JVM specification changes. In these cases we compare the design of the project to our list of goals for our JVM modification.

Before considering the different extensions, we will first explain which enhancements to Java's type system we are considering and show how they are represented in the LOOJ source language.

2.1 Potential Type System Enhancements

The enhancements to Java's type system that we are considering are F-bounded parametric polymorphism, exact types, and the `ThisType` and `ThisClass` constructs. This section gives a crash course in these concepts.

2.1.1 Parametric Polymorphism

Parametric polymorphism is a language feature that is well understood and is included in many modern programming languages, such as C++ and ML. In object-oriented terminology, a **polymorphic type** or **polymorphic class** (also referred to as a **generic type** or **generic class**) is a function from types to a set of **polymorphic instantiations**. An **instantiation of a polymorphic type** refers to a specific version of the generic template in the range of the function.

Analogously, a **polymorphic method** is a function from types to a set of concrete, specialized versions of that method. Experienced ML programmers make heavy use of polymorphic methods.

Consider the following generic class specification:

```
public ListNode<T> {
    T value;

    public T getValue() {
        return this.value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

`ListNode<T>` is not an actual type, but rather a general specification that can be specialized for different situations. `T` is called a **type parameter** because it is a parameter to the function whose range is a set of specialized, concrete classes. One example of such a specialization is:

```
ListNode<String>
```

An object of type `ListNode<String>` behaves as if it were programmed as the following class:

```
public ListNode<String> {
    String value;

    public String getValue() {
        return this.value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

To make a type parameter more useful, it can be **bound** in its declaration. Whenever we encounter a type parameter we treat it in type checking as if it were an extension of its bound. In the following example, the local variables `a` and `b` are treated as if they were of a type that extends `Comparable` because they are declared to be type `T`, which is bounded by `Comparable`.

```
interface Comparable {
    // returns whether this is >,<,= to other
    public int compareTo(Comparable other);
}

class MergeSort<T implements Comparable> {
    void merge() {
        T a, b;
        ...
        if (a.compareTo(b) == 0) { ... }
        ...
    }
}
```

So we see that parametric polymorphism allows us to create generic data structures that can behave differently in different situations. This is a dramatic improvement over the current expressiveness of Java's type system. Most researchers agree on this, but many disagree on the way it should be implemented in an extension to Java.

Heterogeneous versus Homogeneous Implementations of Polymorphism

Implementations of parametric polymorphism come in two distinct forms: **hetergeneous** and **homogeneous**. In a purely heterogeneous implementation, an actual class or method is created for every different instantiation encountered. To continue with our `ListNode<T>` example, when the compiler encounters a `ListNode<String>` it would actually produce a full copy of the code in which every occurrence of `T` is replaced by `String`. The C++ template system is an example of this type of implementation.

The largest drawback associated with heterogeneous implementations is the large possibility of code-bloat occurring because every instantiation of a polymorphic type has its own source code and run-time memory requirements. There are a few optimization strategies used to limit the amount of redundant code. Two of these optimizations are presented later in this chapter with two of the heterogeneous designs, NextGen and Agesen et al.'s proposal. Further run-time overhead with heterogeneous implementations

occurs while loading program code into memory. It takes time to load each specific version of a polymorphic type from disk or the network.

In a purely homogeneous implementation, the generic code is the only copy of the code physically present. Instantiations only exist to allow for static type-checking. Thus `ListNode<String>` and `ListNode<Integer>` are only treated as different types statically, while at run time they both use the same compiled code.

Homogeneous implementations do not suffer from code-bloat, but they do have their own shortcomings. For example, `ListNode<String>` and `ListNode<Integer>` are different types, and so each should have its own set of static variables. However, in a purely homogeneous implementation they would share static variables because they are the same type at run time.

In addition to purely heterogeneous and homogeneous implementations, there are also designs that are a mixture of both. The JVM modification we present in this thesis is an example of such a hybrid design.

2.1.2 F-bounded Parametric Polymorphism

F-bounded parametric polymorphism for object-oriented languages was introduced by Canning et al. [CCH+89]. This particular type of polymorphism refers to the ability to use a type variable in its bound. For example, `T` is an F-bounded type in `class A<T extends B<T>>`.

LOOJ's syntax for F-bounded polymorphism comes from GJ [BOSW98], which inherits its polymorphic syntax from the template system of C++. Class type parameters are declared in the class declaration while method type parameters are declared in the method declaration. The following is a (useless) example of LOOJ source code that contains type parameters. For a full description of LOOJ syntax, see Foster's undergraduate thesis [Fos01].

```
public class A<T> { ... }

// W is an example of an F-bounded polymorphic type
public class B<T, W extends A<W>> {

    // a polymorphic method's syntax
    public <N> void m(N n, T t) { ... }
}
```

2.1.3 A Consistent Example

Thus far we have been speaking primarily in theoretical terms. For the rest of this chapter, it is going to be helpful to have a concrete and consistent code example. We will use the following basic, generic linked list. Many of the implementation details are unimportant and have been omitted.

One thing to take from the example is that there currently is no way to create a new object of the same type as a type parameter (much as there is no way to directly create a new object with the same type as an interface). That is, if `T` is a type parameter then the following code is illegal:

```
new T();
```

To solve this we use the factory design pattern, as shown in figure 2.1. As we discuss other language features, such as exact types and `ThisType`, we will improve the initial generic design for `ListNode<T>`.

2.1.4 Exact Types

A variable has an **exact type** if the only value it can hold is exactly the type declared, not an extension of it. An exact type is declared by using the character '`@`' in front of the normal type, as in `@ListNode<T>`.

To illustrate the type checking rules, consider the following set of classes.

```
interface I {
    public void foo();
}

class A implements @I {
    public A() { ... }
    public void foo() { ... }
}

class B extends A {
    public B() { ... }
    public void bar() { ... }
}
```

Now consider the following code.

```
A  a = new A();
A  a = new B();
@A a = new A();
@A a = new B();  /* ERROR */

I  i = new A();
   i = new B();
@I i = new A();
@I i = new B();  /* ERROR */
```

The first error occurs because exactly an `A` is required by the left hand side of the assignment whereas an extension of `A` is provided on the right hand

```
public interface Factory<T> {
    // return a new T
    public T produce();
}

public class ListNodeFactory<T>
    implements Factory<ListNode<T>> {

    public ListNode<T> produce() {
        return new ListNode<T>();
    }
}

public class ListNode<T> {
    protected T value;
    protected ListNode<T> next;

    public T getValue() {
        return this.value;
    }
    public void setValue(T value) {
        this.value = value;
    }

    public ListNode<T> getNext() {
        return this.next;
    }
    public void setNext(ListNode<T> next) {
        this.next = next;
    }
}
```

Figure 2.1: A first attempt to write a generic node for a linked list in LOOJ.

side. The second error occurs because `B` does not implement `I` exactly, since it added the method `bar()`.

Thus a class only implements an interface exactly if:

- it declares that it implements the interface exactly (that is, even if a class accidentally implements an interface exactly, it still must declare the implementation to be exact),
- it contains only the methods required by the interface (as well as any number of constructors).

Exact types are used to help solve the binary methods problem, which we discuss next.

2.1.5 `ThisType` and Binary Methods

In Java, the keyword `this` refers to the currently executing object. In LOOJ, this concept is extended to include the keywords `ThisType` and `ThisClass`. We define `ThisType` and `ThisClass` as follows:

- `ThisType` refers to the public interface of the currently executing object. To be consistent with Java's use of interfaces, `ThisType` has access to all public methods but not to public fields of the currently executing object.
- `ThisClass` refers to the actual class of the currently executing object. It has access to everything in the currently executing object, including private variables (though this property is likely to change in the next release of the LOOJ language specification).

What follows in this section is a very brief description of `ThisType` that illustrates how it adds expressiveness to Java. `ThisClass` is covered in more detail in the next section. `ThisType` is described more completely in [BPF97] and its implementation in LOOJ is fully described in [Fos01].

`ThisType` is needed to solve problems arising from the use of binary methods. A **binary method** is a method that has a parameter of the same type as the object receiving the method call. In our linked list example, the `setNext()` method in `ListNode<T>` is a binary method.

Many problems with binary methods arise when they are inherited. For example, suppose we define a class `DoubleListNode<T>` for a doubly linked list as follows:

```
class DoubleListNode<T> extends ListNode<T> {
    protected DoubleListNode<T> previous;

    public void setNext(DoubleListNode<T> next) {
        this.next = next;
```

```
            if (next != null)
                next.previous = this;
        }

        public DoubleListNode<T> getPrevious() {
            return this.previous;
        }
        public void setPrevious(DoubleListNode<T> previous) {
            this.previous = previous;
            if (previous != null)
                previous.next = this;
        }
    }
```

There are a number of problems with even this simple example. Conceptually, `DoubleListNode<T>` *overloads* `setNext()` instead of *overriding* it because its parameter is a different type than that of the original `setNext()` method. Overriding the method is conceptually what the author of such a class would intend in order to avoid the following run-time possibility, in which the version of `setNext()` called dynamically would be the one defined in `ListNode<T>`.

```
DoubleListNode<String> a = new DoubleListNode();
ListNode<String>       b = new ListNode();

// statically type-safe, but now a singly linked node
// has found its way into a doubly linked list
a.setNext(b);
```

Furthermore, the following basic code does not pass static type checking.

```
DoubleListNode<String> a = new DoubleListNode<String>();

// STATIC TYPE ERROR:
//    the type returned by getNext() is ListNode!
a.getNext().getPrevious();
```

By using `ThisType`, we can avoid these problems. The code in figure 2.2 is revised source code for `ListNode<T>` and `DoubleListNode<T>` that takes advantage of the `ThisType` language extension.Note that the mutually recursive code in `setNext()` and `setPrevious()` reflects the fact that variables of type `ThisType` do not have access to an object's instance variables. Thus the statement:

```
previous.next = this;
```

```
class ListNode<T> {
    protected T value;
    protected ThisType next;

    public T getValue() { return this.value; }
    public void setValue(T value) { this.value = value; }

    public ThisType getNext() { return this.next; }
    public void setNext(ThisType next) {this.next = next;}
}

class DoubleListNode<T> extends ListNode<T> {
    protected ThisType previous;

    public void setNext(ThisType next) {
        this.next = next;
        if (next != null && next.getPrevious() != this) {
            next.setPrevious(this);
        }
    }

    public ThisType getPrevious() {
        return this.previous;
    }
    public void setPrevious(ThisType previous) {
        this.previous = previous;
        if (previous!=null && previous.getNext()!=this) {
            previous.setNext(this);
        }
    }
}
```

Figure 2.2: A second, not yet correct attempt to write a generic node for a linked list in LOOJ.

would be illegal because `previous` is of type `ThisType`.

When `DoubleListNode<T>` inherits `next` and `getNext()`, both will be of type `ThisType`, which now correctly refers to the public interface of `DoubleListNode<T>` and *not* `ListNode<T>`'s.

Using `ThisType` in our new class definitions ensures that `ListNode<T>`'s can no longer find their way into doubly linked lists, and that the example that fails static type checking now passes. However, our implementation still has some problems as we will illustrate next.

Matching Classes and Exact Method Receivers

The use of the `ThisType` construct alone does not fully solve the binary method problem. Consider the following method.

```
void ThisMightFail(ListNode<T> a, ListNode<T> b) {
    a.setNext(b);
}
```

If we invoke this method with a `DoubleListNode<T>` as the first parameter and a `ListNode<T>` as the second, we get an error when we try to execute `setNext()`, which requires a `DoubleListNode<T>` and not a `ListNode<T>`. That is, `DoubleListNode<T>`'s `setNext()` method takes a `ThisType`, which refers to the public interface of `DoubleListNode<T>`'s. Thus a `ListNode<T>` clearly cannot be passed as in the above situation.

We note that `DoubleListNode<T>` therefore cannot be used wherever a `ListNode<T>` is required. We say that `DoubleListNode<T>` **matches** `ListNode<T>`. In general, wherever the latter is needed, the former can be used, but *not when it is the receiver of a binary method call*.

We would like our language extension to be statically type safe, and thus at compile time catch errors like one found in `ThisMightFail()` above. Therefore LOOJ places the following restriction on the receivers of binary methods.

> The object receiving a binary method call, where the method being called uses `ThisType` or `ThisClass` as parameters, must be an exact type.

Following this rule, we can rewrite our `ThisMightFail()` method to be correct.

```
void ThisMightFail(@ListNode<T> a, ListNode<T> b) {
    a.setNext(b);
}
```

Notice that only the method receiver, `a`, needs to be an exact type. There is no problem in this method with setting the `next` variable in `a` to point to a `DoubleListNode<T>` and not a normal `ListNode<T>`.

```
class ListNode<T> {
    ...

    protected ThisType next;

    public ThisType getNext() {
        return this.next;
    }
    public void setNext(@ThisType next) {
        this.next = next;
    }
}

class DoubleListNode<T> extends ListNode<T> {
    protected ThisType previous;

    public void setNext(@ThisType next) {
        this.next = next;
        if (next != null &&
            next.getPrevious() != this) {

            next.setPrevious(this);
        }
    }

    public ThisType getPrevious() {
        return this.previous;
    }
    public void setPrevious(@ThisType previous) {
        this.previous = previous;
        if (previous != null &&
            previous.getNext() != this) {

            previous.setNext(this);
        }
    }
}
```

Figure 2.3: A third, still not completely correct attempt to write a generic node for a linked list in LOOJ.

Although the `ThisMightFail()` method is now statically type safe, we see that other methods in `ListNode<T>` and `DoubleListNode<T>` violate our new rule. We correct the source by ensuring that no receiver of a binary method is an inexact type in figure 2.3. That code is perfectly legal LOOJ code and will statically type check. However, statements such as the following are still not legal.

```
listnodeA.getNext().setNext(listnodeB);
```

Here, `getNext()` returns an inexact type, which is then the receiver of the binary method call to `setNext()`. A final solution to the `ListNode` problem is presented in figure 2.4. This design ensures that a list will be composed entirely of `ListNode<T>`'s or `DoubleListNode<T>`'s, but not a mix of the two.

So we see that exact types and the `ThisType` construct add expressiveness to the Java language by allowing for clear, type safe implementations of binary methods. However, an object of type `ThisType` cannot access an object's instance variables and there are some situations where that functionality is useful.

2.1.6 ThisClass

Recall that while `ThisType` refers to the public interface of the currently executing object, `ThisClass` refers to the class itself, including all private and protected data. Therefore, `ThisClass` cannot be used in an interface as its usage does not make sense in that context. That is, the following code is illegal:

```
interface I {
    public void illegalParameterType(ThisClass foo);
}
```

In practice, `ThisType` solves most tricky inheritance problems for Java. However, consider the following situation in which it is insufficient.

```
class A {
    protected int x;

    public boolean equals(ThisType t) {
        // ERROR!  ThisType is the public interface of
        //         A, and x is an instance variable.
        return x == t.x;
    }
}
```

Unlike an object of type `ThisType`, an object of type `ThisClass` has access to all private and protected information, so this corrected version of `A` is legal.

```
class ListNode<T> {
    protected T value;
    protected @ThisType next;

    public T getValue() { return this.value; }
    public void setValue(T value) { this.value = value; }

    public @ThisType getNext() { return this.next; }
    public void setNext(@ThisType next) {
        this.next = next;
    }
}

class DoubleListNode<T> extends ListNode<T> {
    protected @ThisType previous;

    public void setNext(@ThisType next) {
        this.next = next;
        if (next != null &&
            next.getPrevious() != this) {

            next.setPrevious(this);
        }
    }

    public @ThisType getPrevious() {
        return this.previous;
    }
    public void setPrevious(@ThisType previous) {
        this.previous = previous;
        if (previous!=null && previous.getNext()!=this) {

            previous.setNext(this);
        }
    }
}
```

Figure 2.4: A final, correct implementation of a generic node for a linked list written in LOOJ.

```
class A {
    protected int x;

    public boolean equals(ThisClass t) {
        return x == t.x;
    }
}
```

`ThisClass` can be used in place of `ThisType` whenever access to a class's instance variables is helpful, although, again, it cannot be used in interfaces.

`ThisClass` Constructors

In addition to providing a specification for a modified JVM, this thesis is also acting as an update to the previous specification in [Fos01]. Therefore, wherever discrepancies exist between the two works, this work should be considered to contain the more recent specification.

The last LOOJ language specification included `ThisClass` constructors. That is, the following statement would have been legal LOOJ source.

```
new ThisClass( ... );
```

In the latest version of LOOJ `ThisClass` constructors were abandoned in favor of the factory design pattern, which takes very little time to code and is considered good software engineering practice.

2.1.7 First Class Types

We have seen how the addition of F-bounded parametric polymorphism, exact types, `ThisType`, and `ThisClass` add expressiveness to the Java programming language. However, that expressiveness becomes limited if an object of one of these new types has fewer language features available to it than a normal Java class or interface.

A **first class type** is a type that can be used in all the standard ways. In Java, a first class type can be used in type casts and `instanceof` operations. Furthermore, unless the type is primitive, it should be able to take advantage of Java reflection. For example, if polymorphic types are treated as first class in a language extension, the following lines of code would be legal:

```
o instanceof ListNode<String>
(ListNode<String>)o
```

We introduce this concept here because some extensions do not treat polymorphic classes as first class and this is certainly a characteristic of polymorphic classes that we would like to see supported in any language extension adding parametric polymorphism to Java.

2.2 GJ

GJ [BOSW98] is the most popular extension adding F-bounded parametric polymorphism to Java. Indeed, the current JDK Java compiler was largely taken from the GJ compiler and Sun is working with Odersky and others from the GJ project to officially include generics in Java's next major release.

Much of GJ's success arises from its design simplicity. It is a homogeneous implementation of parametric polymorphism. It translates source code with generics into bytecode by replacing type variables with their bounds (or `Object` if no explicit bound is given) and inserting casts that are guaranteed to succeed so that the resulting bytecode passes verification. This process is called **erasure** because it essentially erases GJ-specific type information and replaces it with standard Java. This process is illustrated in subsection 3.4.1.

One consequence of using erasure to translate GJ source code to Java bytecode is a need for extra run-time casts. Another consequence of the loss of GJ-specific static type information is that the language does not treat parameterized types as first class. That is, it is illegal to write the following in GJ:

```
o instanceof ListNode<String>
```

as no information about parameterized types exists beyond compilation, and in particular at run time. This costs associated with erasure are also discussed in subsection 3.4.1.

Despite the loss of GJ-specific type information during compilation, separate compilation for polymorphic classes is still possible because class files generated by GJ contain an attribute with the necessary type information. This process is described in depth in Chapter 4.

The major complication that GJ has to deal with occurs when a bound is restricted in a subclass. For example, consider two classes, `A` and `B`, defined below.

```
class A<T> {
    void foo(T t) { ... }
}

class B<T extends Comparable> extends A<T> {
    void foo(T t) { ... }
}
```

This is perfectly legal, but when GJ translates it to standard Java the signature of `foo()` in `B` erases to `foo(Comparable t)` while that in `A` erases to `foo(Object t)`. Thus in the source language `foo()` is *overridden* in `B` while in the compiled bytecode `foo()` is *overloaded* in `B`. This causes method binding difficulties that are solved by constructing a **bridge method** to

guarantee that the correct method is called at run time. Bridge methods and their impact on the performance of GJ code are discussed in more detail in Chapter 3.

One major advantage to GJ is that existing library classes can be easily retrofitted as polymorphic classes by creating a "wrapper" class representing its methods. For instance, suppose we have an existing, non-polymorphic class `LinkedList`. We can create a small parameterized wrapper class `LinkedList<T>`, consisting only of method declarations but no definitions, that erases to the original class and that we can use to type check instantiations of the polymorphic class at compile time. Thus without modifying or re-implementing `LinkedList`, and even without creating a new class that is needed beyond compile time, we have effectively turned the original, fixed data structure into a polymorphic class. As an enormous amount of unparameterized library code and data structures currently exist, such as all the classes in the java.util package, this has been a big selling point for GJ. No other proposal boasts this functionality.

As GJ does not treat parametric types as first class types, it does not include full parametric polymorphism. If possible, we would rather use an extension that treats parameterized types as first class.

2.3 Regarding Heterogeneous Approaches

GJ is an example of a homogeneous implementation of parametric polymorphism because all polymorphic types erase to the same type at run time. In the previous section, we showed how this loss of information caused GJ to disallow certain operations on polymorphic instantiations, such as `instanceof` and type casts.

One general approach that makes it relatively easy to treat instantiations of a parameterized type as first class without JVM modification is through heterogeneous compilation. As described in section 2.1.1, a heterogeneous compiler would create a new class for every new instantiation of a parameterized class. Thus the first major benefit to using the purely heterogeneous implementation in a Java extension is that instantiations of polymorphic types are automatically first class, as they are just regular Java classes.

Another benefit to heterogeneous compilation is that it could potentially yield a boost in speed since casting becomes largely unnecessary. For instance, if a `LinkedList<T>` is instantiated with `String` its methods would actually return `String` objects. Unfortunately, in the only direct speed comparison between heterogeneous and homogeneous implementations, the Pizza team[1] found that this efficiency boost, which was minimal to begin with, was often outweighed by the increased load time necessary in finding and loading each instantiation of the parameterized class [ORW98].

[1]The Pizza project evolved into GJ and was developed by many of the same people.

In addition to extra loading time, which can become considerable when multiple instantiations of a parameterized class are sent over a network, it is impossible to instantiate a new parameterized type on the fly because instantiated versions of a polymorphic class are generated by the compiler. This is inconsistent with the full reflective capabilities offered by Java.

In summary, the major advantage to using a standard heterogeneous approach is that it allows objects of a parameterized type to be treated as first class at run time without taking a large performance hit or without modifying the JVM. The major disadvantages include greater run-time memory requirements, extra load-time needed to load multiple instantiations of a polymorphic type, and the inability to create new instantiations of a polymorphic type on the fly.

In addition to these disadvantages, it turns out that purely heterogeneous implementations allow violations of Java's encapsulation rules regarding packages, as we will now illustrate.

2.3.1 The Package Dilemma

The way Java's package protection mechanism is designed actually makes a purely heterogeneous approach incorrect[2]. Suppose we have two packages, `PackageA` and `PackageB`. `PackageA` contains the public parameterized class `A<T>` and `PackageB` contains a public class `B` and a package protected class `HiddenB`. Consider the following situation:

```
package PackageA:
    class A<T> {
        T t;
        T getT() {
            return t;
        }
    }

package PackageB:
    class B {
        public void crackPackageB() {
            A<HiddenB> a = new A<HiddenB>();
        }
    }
```

When we compile this code, it produces a specialized class that represents the instantiation of `A` with `HiddenB`.

```
PackageA:
    class A_HiddenB {
```

[2]This was first documented in [ORW98].

```
        HiddenB t;
        HiddenB getT() {
            return t;
        }
    }
```

Thus class `A_HiddenB` in `PackageA` explicitly uses a protected class from package `B`! This clearly violates the idea of package protection in Java. To solve this, one might add a restriction that disallows instantiation of a parameterized class from outside the current package with a package protected class from within the package, but this decreases the flexibility of parametric polymorphism by restricting it to fewer situations.

The package problem was not solved in this next heterogeneous approach.

2.4 Fixing Some Code Bloat

Despite the general run-time overhead associated with heterogeneous approaches, there have been proposals that minimize its cost. Agesen et al. call for a small modification to the JVM's class loader and class file specification that allows for load-time heterogeneity as opposed to compile-time heterogeneity [AFM97]. This solves the problem of file bloat, which we have said is a potentially serious problem when transferring a program through a network. Furthermore, by having the generic version of the class file around until load time, client classes can create instantiations on the fly, which, as mentioned earlier, is not possible in a heterogeneously compiled approach.

Thus heterogeneous instantiations of a class `A<T>` are created by the modified class loader. Aside from excess time spent finding and loading instantiated versions of a polymorphic class from memory or from the network, the run-time costs associated with this method are the same as for a purely heterogeneous approach.

One benefit to their approach is that it allows **mixins**. Traditionally mixins are difficult to implement because they make separate compilation of code difficult. Since their loader produces the heterogeneous instantiations, it can type check mixins at load time just before execution, allowing for proper static type checking in the verifier.

Mixins are not supported by LOOJ or by our proposed VM modification, so the problems they raise with current type checking models are not explored here. See [AFM97] for a discussion of mixins.

2.5 NextGen

Cartwright and Steele's NextGen is one of the more ambitious proposals [CS98]. Their goal is to go beyond GJ and support first class parametric

polymorphism without modifying the JVM. However, the implementation of NextGen is so complicated that it is unclear whether any correct implementation exists.

NextGen is basically a heterogeneous translation, meaning that each instantiation of a parameterized class is represented as a separate class. It is not purely heterogeneous, however, as most of the code of a parameterized class is shared between its various instantiations.

It accomplishes this by building a type hierarchy consisting of lightweight wrapper classes and interfaces. When a polymorphic class is compiled, a lightweight wrapper interface and an abstract class representing the uninstantiated parameterized class are created. When a new parameterized type is instantiated at the source code level, a new class is created in which specific code "snippets" are generated to handle the instantiated type parameter. That is, every mention of a type parameter in a parameterized class is replaced by an abstract method call to a snippet function that is implemented (i.e. generated by the NextGen compiler) in the lightweight wrapper class representing the instantiation of the parameterized type.

Thus most of a class's code is in the abstract parent class while the lightweight child classes, which represent instantiated versions of the parent, contain only a small amount of specialized code. In this way NextGen's design, though effectively heterogeneous as multiple classes exist for each instantiated type, does not suffer as much code or run-time memory bloat as other purely heterogeneous approaches.

However, the class loader still has to load an interface, an abstract class, and, potentially, multiple lightweight wrapper classes. Furthermore, just as with purely heterogeneous implementations, new instantiations cannot be created on the fly because the compiler generates all the heterogeneous instantiations,

A more detailed description of the translation process can be found in [CS98]. For our purposes, it is sufficient to note that the primary benefits of NextGen are that it treats instantiations of a polymorphic class as first class without any modification to the JVM and with less overhead than a standard heterogeneous approach. However, should the JVM support parametric polymorphism, these benefits would become unimportant. In other words, the extremely complicated translation and generated inheritance hierarchy become unnecessary with a smarter JVM.

2.6 LM

Like NextGen, Load-Time Management (LM) treats parameterized types as first class types without any modification to the virtual machine. Like GJ, it is a homogeneous approach. It accomplishes this by adding instance variables to a polymorphic class that remember information about type vari-

ables for specific instantiations at run time. Also like GJ, LM uses erasure when translating its source code into Java bytecode.

To see how this homogeneous approach successfully treats polymorphic types as first class, consider the following polymorphic class:

```
class A<T> { ... }
```

When LM compiles this class, it would effectively output the following Java code:

```
class A {
    public Object T;
    ...
}
```

where `T` is an object containing information about the type parameter at run time. It would then translate expressions such as type casts and `instanceof` involving instantiations of polymorphic types into multi-line statements that use the instance variable `T`. Continuing with our previous example, this single LM statement:

```
(o instanceof A<String>)
```

would be translated (roughly) to the following code:

```
(if ((o instanceof A) && (o.T instanceof String)))
```

The exact details of this translation are not relevant to us and can be found in [VN00].

Unlike the heterogeneous implementations encountered, it does not suffer from much run-time space overhead, though there is some space overhead due to objects keeping LM type information in instance variables. However, simple operations such as `instanceof` and checked type casts become more complex, multi-line statements that take longer to execute.

This approach is very similar to the one taken by LOOJ and was developed around the same time. An example of the LOOJ translation of type casts and `instanceof` expressions can be found in Chapter 3.

2.7 PolyJ

Myers et al.'s PolyJ [MBL97] is another proposal that adds F-bounded parametric polymorphism to Java. This approach is based on **where** clauses, which bound type parameters by specifying structural requirements rather than by using by-name extension requirements. The concept of `where` clauses was first introduced in the CLU language [LSAS77].

For example, suppose we write the following GJ code:

```
interface Comparable<T> {
    int compareTo(T other);
}

class A<T implements Comparable<T>> {
    ...
}
```

In PolyJ this code would be represented as:

```
class A[T]
    where T { int compareTo(T other); }
{
    ...
}
```

Thus rather than specifying that the type parameter `T` implements a specific interface or extends a specific class, a `where` clause is inserted that merely requires the type parameter to have the given public methods.

There are some advantages to having structural requirements as opposed to by-name extension requirements. The standard argument given in support of `where` clauses is the following. Suppose there exist a number of classes, all with similar functionality, but that do not implement a specific interface for whatever reason. Rather than creating a new interface and modifying all those classes to implement the interface, it would be easier to retrofit the parameterized class to allow it to be instantiated with the older, existing code.

We believe that the problems associated with structural type bounding outweight this advantage. There are three major problems with bounding a type parameter with structural requirements rather than with by-name extension. The first is that it adds a new concept to Java. The rest of the proposals use `extends` or `implements` so as to make it easier for Java programmers to catch on to the new concept of parametric polymorphism.

The second disadvantage, though rare in practice, is accidental conformance. The example most often given is that of the artist and the cowboy.

```
class Artist() {
    public void draw() { ... }
}

class Cowboy {
    public void draw() { ... }
}

class ArtSchool[T]
```

```
        where T { void draw(); }
    { ... }
```

In this situation, an `ArtSchool` class can be instantiated with a `Cowboy` as well as with an `Artist`, which is not desirable behavior.

This leads to our third problem. Constructs like `where` clauses can lead to messy compatibility problems as they allow for a rather cheap way out of a bad project design. A complex set of components with similar functionality really should have a single name and explicit design specification, such as an interface, associated with it. If no such interface is created, then any one of the individual classes in this set of components could change the name of a specific method, potentially disallowing it to instantiate type parameters using `where` clauses that still use the old name of the method.

For example, suppose we create the following class:

```
class University {
    ...
    ArtSchool[Artist] artSchool;
    ...
}
```

Now, suppose that a month later the person responsible for implementing the `Artist` class decides that it is more accurate to name the `draw()` method `sketch()`, and does so. This change renders `University` non-executable. By having classes that share a common functionality implement a common interface, we are guaranteeing that this problem does not occur.

Aside from using `where` clauses instead of by-name extension, PolyJ does not introduce additional expressiveness beyond what the other proposals offer. For the reasons given above, we feel that `where` clauses are not the best way to add parametric polymorphism to Java and therefore will restrict ourselves to a source language that is more consistent with Java's use of by-name extension.

2.7.1 Enhancing the JVM for Speed

There are two implementations available for PolyJ. The first does not require any modification to the existing JVM specification and suffers from many of the same complications as GJ. The second is the only design that calls for a fairly large change to the JVM specification. It modifies the bytecode language, class loader, bytecode verifier, and runtime representation of classes to support F-bounded parametric polymorphism in the JVM. They found up to a 17% speedup in some cases, with only a 2% slowdown in some situations not using parametric polymorphism, but contend that the minor performance hit taken in those situations was due more to a lack of optimization than to an inherent performance overhead associated with their

design [MBL97]. Thus PolyJ can be extremely fast with support in the VM, which lends strong support to our argument that support for parametric polymorphism should be included in the JVM's type system.

The PolyJ VM fits much of our initial criteria, although it does not support exact types, `ThisType`, or `ThisClass`. The details of their JVM modification are beyond the scope of this thesis, but it is sufficient to note that one of our explicit goals, in contrast to their design, is to limit our modification of the JVM to loading and linking processes while avoiding changes to other areas, such as the bytecode language or run-time class representation, that affect complex systems in the JVM, such as garbage collection and JIT compilation.

2.8 LOOJ

LOOJ, an extension to Java developed at Williams College by Kim Bruce and his students, has F-bounded polymorphism, `ThisType`, `ThisClass`, and exact types. It therefore boasts the most expressive type system of any of the proposed extensions thus far.

At the moment, LOOJ runs without JVM modification using a very similar approach to that taken by LM (it uses erasure for translation and instance variables to keep track of polymorphic instantiation specifics) and its performance suffers accordingly. During translation, the instance variable inserted into polymorphic classes is a `PolyClass` object. The `PolyClass` class contains much of the code used by LOOJ at run time to perform `instanceof` and `checkcast` operations with polymorphic types.

LOOJ's language extensions, type rules, and implementation are included in future chapters, so are not presented here. It is the purpose of this thesis to create a simple virtual machine modification that allows for LOOJ code to be run more efficiently that it is now.

2.9 Summary: More Specific Goals

After considering all the major proposals that add parametric polymorphism to Java, we see that the only extensions bringing full first class parametric polymorphism to Java without modifying the JVM are NextGen, PolyJ, LM, and LOOJ, all of which suffer both space and speed hits at run time. Furthermore, the PolyJ JVM modification has shown us that a JVM with an enhanced type system can run parameterized code very efficiently, experiencing a speedup of up to 17% in some cases. Unfortunately, the PolyJ JVM modification uses structural requirements for parametric types and does include support for exact types, `ThisType`, or `ThisClass`. We therefore do not consider its type system a sufficient enough improvement over the current JVM's to warrant a new JVM specification.

We propose a JVM modification that contains support for the type system in LOOJ, which includes exact types, `ThisType`, and `ThisClass`. This differs from PolyJ's JVM extension because it supports by-name and not structural subtyping, as well as additional LOOJ language features.

The remainder of this thesis discusses the design decisions and implementation details of our proposed modification to the JVM specification.

Chapter 3

The Java Virtual Machine

The average man does not want to be free.
He simply wants to be safe.

– H. L. Mencken

Java's security is one of the major reasons behind its success. Its strict type system allows a number of errors to be caught at compile time. The fact that memory is managed at run time by a garbage collector means that developers do not have to spend precious hours tracking down elusive memory leaks. The so-called "sandbox" execution model lets untrusted Applets run locally without the fear of them doing evil things such as modifying the contents of a hard drive. Java 2 even has a flexible run-time `SecurityManager` class that allows programs to create very specific security profiles.

The Java bytecode verifier is the first line of defense a JVM uses to detect potentially unsafe, corrupted, or malicious code. Any bytecode that passes verification is guaranteed to be "safe" to execute. What exactly the term "safe" implies will be described in more detail in section 3.2. Safety is guaranteed, in part, by a link-time static type check performed by the verifier, so it is necessarily impacted by any modification to the JVM's type system. In fact, our goal is to limit our modification to the class loader and verifier so that the run-time system, including any variety of JIT compilers and garbage collectors, remains untouched.

This chapter first gives a quick overview of the design of the JVM, including a description of the purpose and design of the bytecode verifier. A crash course in how the verifier performs type checking follows the specification for the existing verification algorithm. We then explain the design aspects of GJ and LOOJ, including excess type casts and bridge methods, that hinder performance but are necessary in allowing their compiled bytecode to pass

verification.

3.1 An Overview of the JVM

This section offers a brief overview of the design of the JVM, highlighting only those aspects relevant to our project. For a complete description of the JVM specification, see [LY99].

3.1.1 The Class File

The JVM is an abstract specification for a machine that runs Java bytecode found in Java class files. A Java class file includes one or more Java class definitions.

Each class definition contains a list of method definitions. Each method definition includes flags for the method (such as private, synchronized, etc.), a set of **attributes** (annotations containing extra information for the virtual machine), and the method's executable bytecode.

There is an analogous list of field (i.e. static or non-static variable) definitions, containing both flags and attributes for each field.

Each class definition also includes a **constant pool** containing much of the class's data. During execution, the bytecode frequently references the data in the constant pool. For example, a method invocation such as[1]:

```
invokevirtual A/foo()V
```

is represented in the bytecode by a single byte for the `invokevirtual` instruction and a second byte that is an index into the constant pool containing a string representing the method's signature. This string is then used to find the actual method in memory for execution. Thus the constant pool contains all strings for method signatures, referenced classes, etc., which are used at run time.

3.1.2 Class Loading: Seniors Policy

A familiarity with the basic process of class loading is necessary for the discussion on building the LOOJ type hierarchy presented in Chapter 5. Class loading is most easily explained with a quick example.

```
class A { ... }
interface I { ... }
class B extends A implements I { ... }
```

[1]The JVML syntax used in this thesis is that accepted by the Jasmin Java Assembler, which is the most popular Java bytecode assembler.

When `B` is loaded, it causes both `A` and `I` to be loaded. If the VM cannot find either of those class files, a `ClassNotFoundException` is thrown and the loading/processing of class `B` is terminated.

Furthermore, `A` must be **linked** before `B`, as linking a class (i.e. preparing it for actual use) requires verification. Thus even though the class `B` might be the class that is immediately needed, loading it requires the entire type hierarchy below it to be loaded and processed as well.

The important consequence of this is that `A` is verified before `B`.

3.1.3 The Execution Model and Java Bytecode

The term **bytecode** refers to the actual array of bytes in memory (retrieved from the class file) that is executed when a specific method is called. The bytecode itself is written in a machine language called the Java Virtual Machine Language, or **JVML**. Each method has its own bytecode, so any analysis performed on a Java program is done on a by-method basis. Because of this, the verifier performs data-flow analysis on each method individually, as described in section 3.2. This subsection describes the major aspects of the Java execution model that are relevant to the verifier's design. A detailed description of the JVML or run-time environment is beyond the scope of this thesis.

A JVM thread executes one method at a time. The thread maintains a stack of activation records for method calls. This same stack is used by the method at run time for computations and is referred to as the **operand stack** in this context.

There are no registers in the JVM. Instead, a method can request any number (less than 65536) of spaces for local variables, which are stored in the **local variable array**. To remain consistent with its stack-based design, the value stored in a local variable cannot be manipulated directly. Instead, its value must be pushed onto the operand stack, manipulated, and popped off the stack and copied into the variable space again.

Local variables at the JVM level do not necessarily coincide with local variables at the source code level. Consider the following example.

```
void m() {
    int x = 5;
    x + 1;
    float y = 6.0;
    y + 0.1;
}
```

In this case, it might make sense to have the method ask for space for only one local variable. In the first half of the method it could be used to hold `int` values, while in the second half it could be used to hold `float` values.

Alternatively, this method could be compiled so that the operand stack is used exclusively and data is never stored into any local variables.

Every non-static method has at least one local variable which is a reference to the object that is the method receiver. One can think of this object reference as being akin to Java's `this` keyword. Parameters to a method are also stored in local variable spaces. Thus each method's declared local variable size must be at least large enough to hold all values for its parameters.

An object's instance variables are stored with the rest of its data in the heap. Like local variables, their data can only be manipulated through the operand stack.

Thus the operand stack is the only memory that is used directly by the JVM during execution for any type of calculation, method invocation, or object manipulation, while the local variable spaces are used only for storing intermediate values.

A method's header specifies the maximum number of bytes on the stack that the method will ever use and the maximum number of bytes needed for the local variables. The former value clearly does not include space that might be needed for further activation records on the stack in the case of method invocations. Thus a method executes in the finite amount of space it requests. In fact, a method that passes verification is guaranteed not to exceed its requested amount of space. There is no requirement that the stack be empty when the method returns, as is the case for some other stack-based languages.

3.1.4 Some Examples and Deconstructing Signatures

To make the reader more comfortable with the Java execution model before continuing, this subsection includes some Java code, the corresponding translation to bytecode, and an example of what we will see later to be code that fails verification.

The following discussion also includes a deconstruction of how type signatures are represented in the class file, a cursory understanding of which is necessary to understand the LOOJ type signatures presented in subsection 4.1.

```
public class Foo {
        public String getFoo() { return "Foo"; }

        public static void main(String args[]) {
                Foo foo = new Foo();
        }
}
```

Translates to:

```
.source Foo.java
.class public Foo
.super java/lang/Object

.method public <init>()V
.limit stack 1
.limit locals 1
        aload_0
        invokespecial java/lang/Object/<init>()V
        return
.end method

.method public getFoo()Ljava/lang/String;
.limit stack 1
.limit locals 1
        ldc "Foo"
        areturn
.end method

.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
        new Foo
        dup
        invokespecial Foo/<init>()V
        astore_1
        return
.end method
```

As indicated by the `.limit` statements in the translation from Java to JVML, the compiler has inserted space limits into every method.

Suppose we were to execute the main method. The execution would proceed as follows:

1. `new Foo` pushes a reference to an object of type `Foo` onto the top of the stack. Memory is allocated for our new Foo, but the object is uninitialized, meaning that any attempt to use it results in an error.

2. `dup` duplicates the reference on the top of the stack.

3. `invokespecial Foo/<init>()V` looks up a method called "`<init>`" with type signature "`()V`" in a class called "`Foo`". The invocation pops the reference off the top of the stack in the process and initializes the memory it points to by calling the constructor it found in class "`Foo`".

4. `astore_1` pops the value off the top of the stack and stores it into local variable 1.

5. `return` returns from the method call.

This execution never exceeds its declared operand stack limit of 2, nor does it exceed its declared number of local variables. If instead `main` were as follows:

```
.method public static main([Ljava/lang/String;)V
.limit stack 1
.limit locals 2
        new Foo
        dup
        invokespecial Foo/<init>()V
        return
.end method
```

then it would clearly attempt to use more operand stack space than it has requested. Pass 3 of verification would catch this problem, as we shall see in the following section.

Method Type Signatures

The following is a brief explanation of method type signatures. Consider the method `main`. Its type signature is `main([Ljava/lang/String;)V`. Between the '(' and ')' is the list of types of method parameters. After the ')' is the return type of the method. `[Ljava/lang/String;` refers to an array (hence the '[') of String objects, where String can be found in the package java.lang[2]. A class name starts with 'L' and ends with a semicolon.

As a further example of the type signature syntax, the type signature for the following method:

```
int add(int x, int y) {...}
```

is `add(II)I`, where `I` refers to integer.

The signature of a method is used at runtime to call it. In the example above, when the method `getFoo()` is called from another class the JVM uses its full type signature to look it up in memory for execution.

3.2 The Bytecode Verifier

Code passed by the bytecode verifier is basically "safe" to execute by a virtual machine. This section elaborates what level of safety is guaranteed

[2]For historical reasons the '.' used in the Java language is translated to the '/' of the JVML.

of a class that passes verification. It also points out where type checking takes place in the current bytecode verification algorithm.

According to the JVM specification, there are four passes of verification. Although only pass 3 involves type checking, brief descriptions of all four are included here so that readers unfamiliar with Java Class verification understand fully what conditions are satisfied by bytecode that passes verification.

3.2.1 Pass 1: Classfile Integrity

The first pass occurs during the loading of a class's data from a class file into memory. It simply ensures that the file in question is a properly formatted class file. This check is typically performed by the class loader, although its function is included in what the JVM specification calls "verification".

3.2.2 Pass 2: Static Constraint Checks

The second pass ensures that the constant pool is properly organized, which is more thorough than simply checking that it is formatted correctly. There are some constant pool entries that refer to other entries in the pool and this pass checks that they refer to entries of the appropriate type. For example, a `NameAndType` entry consists of two bytes that refer to the indices for two Utf8 formatted strings found elsewhere in the pool. If one of the indices refers to the middle of another entry or to an entry that is not a Utf8 string constant, this pass will catch that error.

It also checks that final classes are not extended and final methods are not overridden, that every class except `Object` has a superclass, that all method references have a valid type descriptor, that abstract methods have no code, and other static constraints that can be checked without looking at a method's actual bytecode.

The first two passes are not affected by our modification to the JVM specification.

3.2.3 Pass 3: Bytecode Verification

Pass 3 is commonly referred to as Java **bytecode verification**. This subsection includes a list of the checks performed during pass 3 as well as the (almost) complete specification for the bytecode verification algorithm. It is included to illustrate where in the algorithm type checking occurs, which is the aspect of verification of interest in our project.

This pass uses a simple data flow analysis algorithm on the bytecode of an individual method (that is, pass 3 is performed on a by-method basis) to check, at each instruction, regardless of what code path is taken to reach the instruction, that (*the following is excerpted from [LY99, 142]*):

- The operand stack is always the same size and contains the same type of values.
- No local variable is accessed unless it is known to contain a value of an appropriate type.
- Methods are invoked with the appropriate arguments.
- Fields are assigned only using values of appropriate types.
- All opcodes have appropriate type arguments on the operand stack and in the local variable array.

During the data flow analysis, the following conditions are also checked (*excerpted from [LY99, 143-144].*):

- Branches must be within the bounds of the code array for the method.
- The targets of all control-flow instructions are each the start of an instruction. In the case of a wide instruction, the wide opcode is considered the start of the instruction, and the opcode giving the operation modified by that wide instruction is not considered to start an instruction. Branches into the middle of an instruction are disallowed.
- No instruction can access or modify a local variable at an index greater than or equal to the number of local variables that its method indicates it allows.
- All references to the constant pool must be to an entry of the appropriate type.
- The code does not end in the middle of an instruction.
- Execution cannot fall off the end of the code.
- For each exception handler, the starting and ending point of the code protected by the handler must be at the beginning of an instruction or, in the case of ending point, immediately past the end of the code. The starting point must be before the ending point. The exception handler code must start at a valid instruction, and it may not start at an opcode being modified by the wide instruction.

Upon reviewing this list, we see that the only (non-logical) errors not caught statically by the verifier include events such as the following:

- addressing a null pointer,
- indexing an array beyond its bounds,
- trying to cast an object to an inappropriate type,

- division by zero,
- etc.

These are managed by dynamic checks. For example, any time an index is made into an array, a bound check is made. Due to the Halting Problem we cannot eliminate many of these run-time errors with static bytecode analysis. However, it is the purpose of this thesis to decrease the number of necessary checked type casts at run time by eliminating the need for casts that are guaranteed to succeed.

The Data Flow Analysis Algorithm

Pass 3 performs these checks through a simple data flow analysis. To familiarize the reader with the algorithm, its specification is included here. A familiarity with the algorithm is necessary in order to understand where and when type checking takes place in the bytecode verifier.

At each instruction, a record is kept of the size and contents (i.e. types) of the operand stack and of the type stored in each local variable. When the data analyzer is initialized, all of the types are set to `unstable` and the "changed" bit for the first instruction of the method is set to true. Then the analyzer is run as follows (*excerpted from [LY99, 144-146].*):

1. Select a VM instruction whose "changed" bit is set. If no instruction remains whose "changed" bit is set, the method has successfully been verified. Otherwise, turn off the "changed" bit of the selected instruction.
2. Model the effect of the instruction on the operand stack and local variable array by doing the following:
   - If the instruction uses values from the operand stack, ensure that there are a sufficient number of values on the stack and that the top values on the stack are of an appropriate type. Otherwise, verification fails.
   - If the instruction uses a local variable, ensure that the specified local variable contains a value of the appropriate type. Otherwise, verification fails.
   - If the instruction pushes values onto the operand stack, ensure that there is sufficient room on the operand stack for the new values. Add the indicated types to the top of the modeled operand stack.
   - If the instruction modifies a local variable, record that the local variable now contains the new type.

3. Determine the instructions that can follow the current instruction. Successor instructions can be one of the following:

   - The next instruction, if the current instruction is not an unconditional control transfer instruction (for instance `goto`, `return`, or `athrow`). Verification fails if it is possible to "fall off" the last instruction of the method.
   - The target(s) of a conditional or unconditional branch or switch.
   - Any exception handlers for this instruction.

4. Merge the state of the operand stack and local variable array at the end of the execution of the current instruction into each of the successor instructions. In the special case of control transfer to an exception handler, the operand stack is set to contain a single object of the exception type indicated by the exception handler information.

   - If this is the first time the successor instruction has been visited, record that the operand stack and local variable values calculated in steps 2 and 3 are the state of the operand stack and local variable array prior to executing the successor instruction. Set the "changed" bit for the successor instruction.
   - If the successor instruction has been seen before, then merge the operand stack and local variable values calculated in steps 2 and 3 into the values already there. Set the "changed" bit if there is any modification to the values.

     To merge two operand stacks, the number of values on each stack must be identical. The types of values on the stacks must also be identical, except that differently typed reference values may appear at corresponding places on the two stacks. In this case, the merged operand stack contains a reference to an instance of the first common superclass of the two types. Such a reference type always exists because `Object` is a superclass of all class, array, and interface types. If the operand stacks cannot be merged, verification of the method fails.

     (Merging two sets of local variables is similar, except that incompatible types are replaced with the type `unstable` *and thus do not immediately cause verification to fail).*

5. Continue at step 1.

Different implementations of the verifier do not necessarily follow this specification exactly. For instance, instead of doing analysis on a per-instruction basis, most do the data flow analysis on the level of basic blocks. Furthermore, when merging two different reference types some verifiers create

a set of possible merged types, including the superclass and all interfaces implemented by both types. Sun's verifier does not use this perhaps more accurate approach.

An Example

To illustrate the process of verification, the following is a step-by-step example of a simple method that fails bytecode verification in a merging step. Recall that, for verification purposes, values of data are not saved (indeed, they are not even available) so only the *types* of values are stored in the simulated operand stack and local variable array.

```
.class A
.super java/lang/Object

.method public add(II)I
.limit stack 2
.limit locals 3
        iload_1
        iload_2
        iadd
JumpLabel:
        pop
        fconst_0
        goto JumpLabel
        ireturn
.end method
```

The reader may become nervous when viewing the infinite loop above. Because of the Halting Problem the verifier cannot check for infinite loops, and as they make it easy to create verification examples one is used here. Recall that verification passes once a stable state has been reached (meaning that no instruction has its changed bit set). According to the algorithm specification, the merger of two operand stacks can only revise the type of a reference by finding the first common superclass of two disagreeing reference types. Therefore, because class hierarchies are finite, at some point a stable state will be reached, so the verifier itself will not check this code infinitely.

At the beginning of the method, the operand stack and local variable array contain the following types.

```
    Operand Stack             Locals
-------------------------------------------  (0)
    unstable                  A
    unstable                  int
                              int
```

Notice that the first local variable contains a reference to the type of object that receives the method call. This reference is akin to Java's `this` keyword. Also, for the purposes of demonstration both the local variable array and the simulated operand stack are indexed from the top down. Thus the top-most values have the lowest indices.

The algorithm then simulates execution. `iload_N` pushes an `int` value stored in variable number `N` onto the operand stack. Obviously, if variable `N` does not contain a value of type `int`, then verification of the method fails. After the two `iload` statements we get:

```
    Operand Stack             Locals
-----------------------------------------  (1)
    int                       A
    int                       int
                              int
```

`add`, which pops off two integer values, adds them together, and pushes the integer result back onto the operand stack, yields:

```
    Operand Stack             Locals
-----------------------------------------  (2)
    int                       A
    unstable                  int
                              int
```

It is important to remember that, during verification, each instruction has an operand stack and local variable array associated with it that is used for merging operations. Thus this last operand stack and local array is associated with the `pop` instruction. That is, when any other code path is taken to reach pop, the operand stack and local variables from that path must be compatible with the above operand stack and local array as described in step 4 of the data flow analysis specification.

To continue our simulation, `pop` yields:

```
    Operand Stack             Locals
-----------------------------------------  (3)
    unstable                  A
    unstable                  int
                              int
```

`fconst_0` loads the floating point constant 0 onto the stack:

```
    Operand Stack             Locals
-----------------------------------------  (4)
    float                     A
    unstable                  int
                              int
```

The `goto` instruction is then processed, which leads to a merge operation with the already existing operand stack and local variable array associated with the `pop` instruction, shown in (2) above. Unfortunately, the top of the stack in (4) is not compatible with that of (2), as one control path yields an `int` value in stack position 0 while the other yields a `float`. Thus verification would fail when attempting to merge these two states.

This example was provided simply to give a feel for the workings of the algorithm that is to be modified. However, it is easy to see how type checking of merges would extend to reference types, which is what we are primarily concerned with. Later in this chapter there is an example of the algorithm being applied to reference types.

3.2.4 Pass 4: Lazy Loading

Pass 4 is only separate from pass 3 for efficiency reasons and does not really affect type checking. It essentially allows the VM to defer loading classes referred to by the class being verified (unless loading them is absolutely necessary for type checking during bytecode verification) until those classes are needed at execution-time. Thus it is not strictly necessary for a VM to separate pass 4 from pass 3, but the specification allows for a lazy loading of referenced classes to delay errors associated with attempts to load missing classes.

One example[3] of when lazy loading is allowed is if, during bytecode verification, a method invocation is processed where the method returns something of type `A` and the value returned is assigned immediately afterwards into an instance variable of type `A`. The bytecode being verified may even access instance variables in an object of type `A` without causing the class to be loaded. Class `A` is not loaded until the method call instruction is executed at run time, at which point pass 4 checks that the currently executing object has permission to access `A` and that any fields it references in `A` exist and are of the correct type.

Pass 4 is not affected by our modification to the JVM as it does not, in principle, have to be separated from pass 3.

3.3 Type Checking and the Verifier

Since we are modifying the JVM's type system, we potentially need to modify the above verification algorithm anywhere type checking occurs. In this section we first discuss where type checking takes place in the verifier. We then describe the current type checking algorithm and show an example of pass 3 verification with type checking.

[3] This example is taken from [LY99, 142].

3.3.1 Type Checking Circumstances

Type checking occurs in two kinds of circumstances: merge operations and non-merge operations. In the former case we ask the question: do types α and β have a common supertype? In the latter we ask: can an β be used in place of an α or not? This subsection shows some examples of non-merge operations that require type checking.

Recall that a merge operation is performed any time more than one control path exists to get between two instructions. Aside from merge operations, type checking occurs with the use of a number of JVML instructions. One example of this is the `aastore` instruction, which stores a reference value in an array. According to JVML type rules, the value to be stored must be an extension of the type the array stores.

Another place type checking occurs is at method invocations. The types on top of the stack are checked against the types of the method parameters. If the types on the operand stack are incompatible with those expected by the method, verification fails.

Type checking is also needed any time an instance variable is written to. Recall that to manipulate an instance variable, its value has to be pushed onto the stack from its location in memory (which would be with its containing object in the heap), manipulated, and then popped back into its location in memory. The verifier guarantees that any value stored in instance variables will be of the correct type.

These are just some examples of where type checking takes place in the verifier. There are others as well. In most implementations they all consult the same methods to answer the type checking questions so this set of methods becomes the focus of our implementation.

We now describe the rules for type checking in the virtual machine more concretely.

3.3.2 Type Checking 101

Java types include arrays, primitives, classes, and interfaces. Aside from primitive types, these are all reference types. At run time, a class or interface is represented in memory by an instance of class `Class`. A `Class` object has information about a class such as its superclass, the list of interfaces it implements, a method invocation table, its static variables, etc.

For merge operations, determining the first common superclass of two classes is therefore trivial. One simply has to follow the superclass links from both classes until a common one is reached. Finding the first common superinterface of two interfaces is analogous. However, multiple interface inheritance may cause the meaning of of "first common superinterface" to be ambiguous. To determine the set of interfaces implemented by both of two classes, we simply find common superinterfaces of the interfaces implemented

by one class and those implemented by another class.

For non-merge operations, we are asking a slightly easier question: can type β be used in place of type α? In this case, we simply search for α in β's type hierarchy. If found, the answer is yes.

The basic type checking rule is therefore:

> To check if type β can be used where the type α is expected, one simply has to follow β's supertype hierarchy, including the hierarchies of those interfaces implemented by β, until either α is found or the entire hierarchy is exhausted. In the latter case, β cannot be used where an α was needed.

A slightly different rule exists for arrays.

> To check if an array type β can be used where the type α is expected, check that the type of object stored in β is an extension of the type stored in α. This process may repeat several times in the case of multidimensional arrays, but will eventually come to two situations. The first is that the two arrays are of different dimensions, in which case they are not compatible. The second is that the algorithm returns the result of the base case operation, which is the standard type check described above.

Thus type checking of reference types relies heavily on the type hierarchy.

3.3.3 Representing the Type Hierarchy at Run Time

All the classes and interfaces currently in the JVM can be accessed through a hashtable, where an entry's key is usually its full type signature, though it could also be the class's package and name. The choice is up to implementors, and there are likely other effective key systems as well, though any system must depend on the class's package and name.

For example, the full type signature of class `String` is:

```
Ljava/lang/String;
```

while its package and name is just `java/lang/String`.

Array types are usually accessed through a second hashtable, where the key is also related to its type signature. For example, the type signature of an array of arrays of `String` is `[[Ljava/lang/String;`.

We will use the term **type hierarchy** to refer to the tree-like structure linking classes to superclasses, superinterfaces, and implemented interfaces. The **class pool** or **class hashtable** refers to the hashtable storing all classes at run time, where the key/value pair is the type signature and a reference to the class object containing the class's run-time data.

Consider the following set of classes and interfaces:

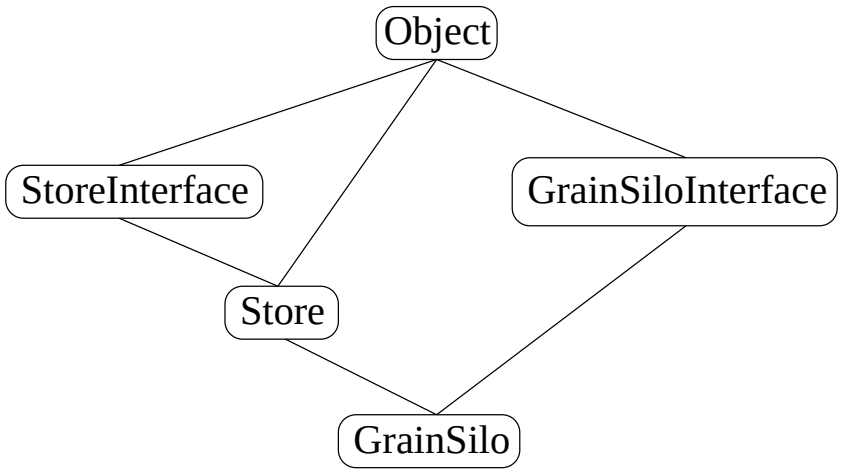

Figure 3.1: A graphical representation of a simple Java type hierarchy.

```
public interface StoreInterface {
    public void setValue(Object value);
    public Object setValue();
}

public interface GrainSiloInterface {
    public void setValue(Grain value);
    public Grain getValue();
}

public class Store implements StoreInterface { ... }

public class GrainSilo
    extends Store
    implements GrainSiloInterface
{ ... }
```

Its type hierarchy is given in figure 3.1. When searching the hierarchy for type checking, the links are followed from the bottom up.

3.3.4 Building the Hierarchy

A class is inserted into the class hashtable as it is loaded from its class file into memory. If the class's superclass, superinterface, or implemented interface is not already in the hierarchy, the the missing supertype must be loaded from memory for the original class to become usable.

As an illustration of this system, consider our `Store` example.

```
class Execute {
    public static void main(String args[]) {
        GrainSilo g = new GrainSilo();
        g.setValue(new Grain());
    }
}
```

Suppose we were to execute the `Execute` class. When the JVM is first loaded, a number of supporting classes are automatically loaded from class libraries into the type hierarchy, including various error and exception classes, class loaders, I/O classes for class loading, the class `Class`, `Object`, etc.

After loading these classes, the `Execute` class file is read and its class data is inserted into the class pool. Since it implements no interfaces and its superclass is `Object`, which has already been loaded, no other class loading occurs.

During verification the `main()` method's bytecode is analyzed.

```
.method public static main([Ljava/lang/String;)V
.limit stack 3
.limit locals 2
    new GrainSilo
    dup
    invokespecial GrainSilo/<init>()V
    astore_1

    aload_1
    new Grain
    dup
    invokespecial Grain/<init>()V
    invokevirtual GrainSilo/setValue(LGrain;)V

    return
.end method
```

In order to push the type of `GrainSilo` onto the simulated operand stack, we lookup its `Class` object in the class pool.

This brings us to an important point. A reference to the `Class` object for a class (or a data structure containing a reference to the `Class` object) is used to represent a type on the simulated operand stack. This gives us easy access to the class's hierarchy, instance variable type signatures, and method type signatures for type checking.

To continue with our example, we cannot yet push a pointer to the `Class` object representing `GrainSilo` during verification onto the top of

the simulated operand stack because `GrainSilo` has not been loaded. So `GrainSilo` is loaded, which causes its superclass `Store` to be loaded and added to the hierarchy, and so on.

Thus we see that, currently, the entire type hierarchy of Java is built on the fly.

3.3.5 Simple Example: Verification with Reference Types

To illustrate the process of bytecode verification with reference types, the following is a step-by-step example of a simple method that passes verification. The method does not actually come from any actual Java source and was written specifically as a simple example of how reference types are merged during bytecode verification. In this example, `A` is just some random class, so `[LA;` is the type signature for an array of `A`.

```
.class Foo
.super java/lang/Object

.method public foo(LA;[LA;)V
.limit stack 1
.limit locals 3
        aload_2
JumpLabel:
        pop
        aload_1
        goto JumpLabel
        return
.end method
```

At the beginning of the method, the type values on the simulated operand stack are initialized with `unstable` and the types in the local variable array are initialized with the `this` reference and the types of the method's parameters.

```
    Operand Stack             Locals
-----------------------------------------  (0)
    unstable                  Foo
                              A
                              [LA;
```

The algorithm then simulates execution. The `aload_n` instructions load an object reference from local variable number `n`.

```
    Operand Stack             Locals
-----------------------------------------  (1)
    [LA;                      Foo
```

```
                                    A
                                    [LA;
```

Thus the state in (1) is associated with the `pop` instruction. After `pop` and `aload_1` we have:

```
    Operand Stack              Locals
-------------------------------------------  (2)
    A                          Foo
                               A
                               [LA;
```

Now, when we jump to the `pop` instruction again, we must merge the operand stacks in (1) and (2). To do this, we find the first common superclass of `[LA;` and `LA;`. As one is an array and the other is not, the first common superclass of both types is `Object`, so after merging we get:

```
    Operand Stack              Locals
-------------------------------------------  (3)
    Object                     Foo
                               A
                               [LA;
```

This is the stable state for the `pop` instruction during verification, as can be seen if the algorithm is applied until `pop` is reached again.

3.4 Catering to the Verifier

Keeping the framework described in the last sections in mind, and remembering that the JVM specification does not allow for the run-time representation of polymorphic types, we can now analyze how GJ and LOOJ take pains to cater to the verifier. Both insert "stupid" type casts that allow the verifier to properly type check bytecode. We call the these inserted type casts "stupid" because they are guaranteed to succeed, given the semantics of the source language. They also use bridge methods to ensure that dynamic method binding works as expected.

Flat Types

For the purposes of our analysis, the term **GJ type** refers to a type that exists in GJ but not in standard Java. Similarly, the term **LOOJ type** refers to a type that exists in LOOJ but not in standard Java. Most GJ types are LOOJ types, and many GJ types are LOOJ types.

One GJ type that is not a LOOJ type is the flat type of a polymorphic type. For a polymorphic type, such as `Store<T>`, the name `Store` is called

the **flat name** or **flat type** so that two distinct instantiations of `Store<T>` have the same flat type.

GJ allows flat types to be used in the source language because all instantiations of a polymorphic type are erased to the flat type during translation anyway. LOOJ does not allow flat types to be used in the source language because it treats polymorphic types as first class at run time through the use of its `PolyClass` instance variables. The reader should keep this in mind for the description of the LOOJVM type checking system presented in Chapter 5.

3.4.1 GJ: Excess Run-time Costs

This subsection describes the two major run-time costs suffered by GJ. These costs are stupid type casts, which exist solely to cater to the verifier's type checking demands, and bridge methods, which exist to allow dynamic method binding to function as one would expect. Both incur run-time costs that our modification does not.

Stupid Type Casts

The first run-time cost is the insertion of "stupid" type casts that are known to always succeed. These casts exist exclusively to satisfy the verifier and are inserted during the compilation step called **erasure**. In this step, GJ-specific type information is erased and replaced by bounds and standard Java types during translation to bytecode. For example, consider the following GJ classes:

```
class Store<T> {
    protected T value;
    public T getValue() { return value; }
    public void setValue(T value) {
        this.value = value;
    }
}

class Foo {
    public static void main(String args[]) {
        Store<String> s = new Store<String>();
        s.setValue(new String(``Foo''));
        String t = s.getValue();
    }
}
```

This is clearly type safe. However, because `T` cannot be stored at run time in the current VM architecture, GJ erases GJ-specific type information and inserts type casts that are guaranteed to succeed. During the translation from

GJ source to Java bytecode, the GJ compiler translates (via erasure) the GJ syntax tree to correspond to the following, legal Java source. When we refer to "translated" or "erased" source, we are referring to this intermediate code that is eventually translated into JVML.

```
class Store {
    protected Object value;
    public Object getValue() { return value; }
    public void setValue(Object value) {
        this.value = value;
    }
}
class Foo {
    public static void main(String args[]) {
        Store s = new Store();
        s.setValue(new String(``Foo''));

        // notice the extra checked type cast
        String t = (String)s.getValue();
    }
}
```

The reason the type cast in that last line is necessary is because at run time there is no representation for the type `Store<String>`. The JVM only contains the class `Store`, which has a field of type `Object` due to the bound of `T` being `Object` (implicitly as no bound was declared).

Although this is a simple example, there are many programs that make heavy usage of "generic" data structures that, in Java, necessarily store values of type `Object`, which undergo type casts whenever they are retrieved from the structure. Individually a checked type cast is not that expensive but they can add up.

Thus one thing we would like our modified verifier to do is pass code like the above without the unnecessary type cast to `String` on the last line, yet still catch type errors.

Bridge Methods

Problems arise in GJ when the bound of a type parameter is constrained in a subclass of a polymorphic class. Consider the following example:

```
class Store<T> {
    protected T value;
    public T getValue() { return value; }
    public void setValue(T value) {
        this.value = value;
```

```
    }
}
class GrainSilo<T extends Grain> extends Store<T> {
    protected int bushelsStored;
    public GrainSilo() { bushelsStored = 0; }
    public void setValue(T value) {
        bushelsStored++;
        this.value = value;
    }
}
```

`GrainSilo` inherits the instance variable `value` as well as the methods `getValue()` and `setValue()`. However, because the bound of type parameter `T` in `GrainSilo` is `Grain` and not `Object`, as it was in the superclass `Store`, the erasures of `Store` and `GrainSilo` become:

```
class Store {
    protected Object value;
    public Object getValue() { return value; }
    public void setValue(Object value) {
        this.value = value;
    }
}
class GrainSilo extends Store {
    public void setValue(Grain value) {
        bushelsStored++;
        this.value = value;
    }
}
```

Thus `setValue()` is *overloaded* in `GrainSilo` and not *overridden*, as would be expected for inherited methods. This can lead to run-time type errors because of Java's dynamic method invocation. For example:

```
Store<Grain> store = new GrainSilo<Grain>();
store.setValue(new Grain());
```

At compile time, the invocation of `setValue()` would be associated with the `setValue()` method in the `Store` class, which has the type signature `setValue(LGrain;)V`. However, at run time we would like to execute the `setValue()` method in the `GrainSilo` class, which would occur if the method were overridden, as is usually the case. However, because `GrainSilo.setValue()` has a different type signature than that which was statically bound to the method call, when the method of signature `setValue(Ljava/lang/Object;)V` is called at run time the only matching

method in the object `store` will unfortunately be the `setValue()` method of the superclass.

To solve this problem, GJ adds a **bridge method**. A bridge method overrides the definition in the superclass and basically forwards the call to the appropriate version in the subclass. Thus the translation of `GrainSilo` becomes:

```
class GrainSilo {
    public void setValue(Grain value) {
        bushelsStored++;
        this.value = value;
    }

    /* BRIDGE METHOD */
    public void setValue(Object value) {
        this.setValue((Grain)value);
    }
}
```

The second method clearly overrides the `setValue()` method from `Store` and ensures that the correct method is called at run time.

Unfortunately, this incurs an additional method call and potentially a few type casts. Parameters that are polymorphic types must all be cast in the bridge method body, just as `value` is cast above. Our modification to the loader and verifier overcomes this cost, as will be explained in Chapter 5.

3.4.2 LOOJ: Excess Run-time Costs

LOOJ suffers from the same run-time overhead as GJ since it uses type erasure during translation to JVML with `ThisType` and `ThisClass` in addition to parametric types and type parameters. In fact, `ThisType` and `ThisClass` are essentially type parameters whose bound is constrained in every subclass. Casts and bridge methods must be inserted as before.

LOOJ has further run-time overhead because the JVM does not support its language extensions at run time. This extra overhead is due to its treatment of LOOJ types as first class. LOOJ allows this by inserting an instance variable of type `PolyClass`, which was described briefly in section 2.8, into every class that is a LOOJ type (recall that a **LOOJ type** is any type in LOOJ that uses its language extensions). During translation, when an action such as `instanceof` is performed on one of these special types, LOOJ translates the simple line into a more complicated set of method calls that effectively performs the action at run time.

The full details of these translations are beyond the scope of this thesis and can be found in [Fos01]. It is sufficient to note that, because we are

avoiding any modification to the Java run-time, we cannot provide more efficient solutions to the problems associated with the treatment of LOOJ types as first class. For example, checking a cast to a polymorphic type is impossible without the `PolyClass` instance variable and the more complicated set of instructions substituted for the simple type cast.

The reader should understand the translation of a type cast to a LOOJ type as it affects type checking in the verifier.

```
Store<String> s = (Store<String>)object;
```

becomes:

```
Object $synth_1$;
Store s =
    ((($synth_1$ = object) == NULL)
     ||
     ((Store)$synth_1$.$instanceOf$Store(
             new PolyClass(String.class)))
    ? (Store)$synth_1$
    : throw new ClassCastException();
```

Thus the one line cast to a parameterized type gets translated to a more complicated expression. The following discussion illustrates how the verifier is affected by this translation.

Verifying LOOJ's Casts

The LOOJ translation of a cast to a polymorphic instantiation illustrated above presents an interesting problem for the bytecode verifier. Consider verifying the LOOJ statement above with the cast to `Store<String>`. In pseudo-bytecode, we can write the statement as:

```
checkcast LStore<Ljava/lang/String;>;
```

Before the statement, the current simulated operand stack and local variable array in the verifier would be:

```
    Operand Stack             Locals
-------------------------------------------  (0)
    ...                       @ThisClass
    SomeRefType               ...
```

After the statement, we get:

```
    Operand Stack             Locals
-------------------------------------------  (1)
    ...                       @ThisClass
    Store<String>             ...
```

Thus, when simulating verification, the correct LOOJ type would be pushed onto the operand stack. However, after LOOJ translation the cast is not represented by a single JVML line, but rather by the complex expression shown above. Unfortunately, it is not immediately clear when the LOOJ type should be pushed onto the stack during verification, especially because erasure has removed all of the LOOJ-specific type information. The solution to this dilemma will be presented in Chapter 4.

As we will see later, our new type checker will only need some special cases for LOOJ types that are not simply type parameters or instantiations of a parameterized type.

3.5 Summary

This chapter explained the aspects of the class file and the JVM run-time class representation that are relevant to our project. It described the current verification algorithm in depth to illustrate where type checking takes place, as the type checker is the primary focus of our JVM modification. It then explained how the verifier utilizes the class pool and class hierarchy to perform type checking. Finally, it explained the current run-time costs incurred by GJ and by LOOJ for their type system enhancements. These costs include extra checked run-time type casts and extra method calls that go through bridge methods.

We almost have enough knowledge to go ahead with our verifier modification. However, before a modified verifier can use LOOJ type information to type check method bytecode during pass 3 of bytecode verification, it must have available to it the source-level, static LOOJ type information. Because LOOJ, like GJ, translates to JVML via erasure, its type information is lost after compilation to bytecode. The next chapter explains how class file attributes are used to maintain LOOJ type information from compilation through verification.

Chapter 4

The Signature Attribute

Knowledge is power.

– Sir Francis Bacon

As we have shown, LOOJ has a type system containing information not present in standard Java and that is lost in the translation process. One must therefore ask the question: if LOOJ type information is thrown out in the translation to JVML, how is separate compilation with LOOJ types possible? More importantly, how can we get the original LOOJ type information to the LOOJ bytecode verifier? When verification occurs at link time, having knowledge of the original LOOJ types isn't just powerful, it is necessary.

This chapter explains how the **Signature attribute** stores LOOJ type information in class files to allow for separate compilation. It then illustrates the failure of the current attribute specification to provide adequate information for bytecode verification. Finally, it presents a new set of attributes that allow the LOOJ bytecode verifier to type check bytecode without the stupid casts and bridge methods shown in the last chapter.

4.1 LOOJ Type Signatures

For the rest of this thesis we need a basic understanding of LOOJ type signatures. The best way to figure these out is to see some examples. A formal specification of the type signatures is in [Fos01].

After giving a brief introduction to LOOJ type signatures, the next section discusses how they are maintained in the Java class file after compilation of LOOJ source code.

4.1.1 Polymorphic Types

As you would expect, polymorphic type signatures start with an 'L', just like other reference types, which is followed by the class name, then '<', then the list of type parameter types, then '>;'. For example, the type signature of Store<String> is:

```
LStore<Ljava/lang/String;>;
```

4.1.2 Type Parameters

The type signature for a type parameter is a 'T' followed by the name of the type parameter and ending with ';'. So a type parameter T is TT;. Thus the type signature of Store<T> is:

```
LStore<TT;>;
```

As a fancier example, consider the type Foo<T,W,String>. Its type signature is:

```
LFoo<TT;TW;LString;>;
```

Suppose Store<T> has a method called m() that takes an int, a T, and another int as parameters and returns an int. Its signature is:

```
Store/m(ITT;I)I
```

Notice that ';' does not act as a "end of type" delimiter for primitive types.

4.1.3 ThisType and ThisClass

One would expect ThisType and ThisClass representations to be similar to those of type parameters, as they are essentially specialized type parameters. However, the type signatures of ThisType and ThisClass provide a little more information.

ThisType starts with 'M' and is followed by the complete type signature of the class that it is in ('M' comes from MyType in LOOM, a language developed by Kim Bruce and his students at Williams). ThisClass is similar, but starts with a 'K'.

Thus a method m() in Store<T> that takes no parameters and returns something of ThisType has a type signature:

```
m()MLStore<TT;>;
```

4.1.4 Exact Types

Just as in the source language, the type signature of an exact type starts with '@' and is followed by a complete type signature. So an instance variable of exactly ThisType in Store<T> has the signature:

```
@MLStore<TT;>;
```

4.2 The Power of Comments

As any programmer who has spent many wasted hours staring at a colleague's source code can attest, comments often contain information essential to understanding the real meaning of pieces of code. Although a class file is not human readable, it contains many "comments" that allow compilers and virtual machine implementations to better understand the code within. These comments are called **attributes**.

The class file specification allows class definitions, method definitions, and field definitions to have any number of attributes. Although attributes can be thought of as comments or annotations in the class file, they are most often functional. They were initially included in the design to allow vendor-specific enhancements to the Java language that would not otherwise affect its overall specification. For example, one could imagine a Java compiler that includes bytecode analysis in a method attribute that is used at run-time by an optimizing JIT compiler, saving the JIT compiler the work of performing analysis for optimization at run-time. There is actually a small set of attributes that all JVM's are required to recognize, and any attribute not in that list should be able to be safely ignored by any JVM implementation in which it is not recognized.

The only type information of a LOOJ class needed by other classes for separate compilation are the LOOJ type signatures of the LOOJ class's public interface. Implementation details, such as the use of LOOJ language features in method code, are completely forgotten in the translation to a Java class file. This loss of LOOJ type information describing the bytecode itself has required us to create an enhanced attribute specification for use in method bytecode verification.

GJ and LOOJ use an attribute called the **Signature** attribute[1] to store non-Java type information in the class file post-translation. Before modification, the LOOJ Signature attribute therefore only contains type information for publicly accessible methods and instance variables that use LOOJ features. Thus the erased type of a public method or field is stored in the class file's method or field definition, while the LOOJ type, which is not usable by the JVM, is essentially stored in a comment within the public method or field definition. The LOOJ compiler then reads the comment in the definition of the public method or field and treats it as its original (i.e. source code) statically declared LOOJ type, and not the "actual" Java type used by the JVM at run-time.

For example, at compile time, a class referencing `Store`'s `setValue()` method would be able see its type signature as:

```
setValue(TT;)V
```

[1] For a full specification of the information stored in the Signature attribute, see [Fos01].

and not the following version, which is the one actually stored in the class file and used by the JVML during execution:

```
setValue(Ljava/lang/Object;)V
```

4.2.1 Inheritance

The type signature in the Signature attribute for a class is not exactly what has been presented above as LOOJ type signatures. It starts with type parameter information, then has the LOOJ signature of the superclass, then the LOOJ signatures of any implemented interfaces.

For example, the strings stored in `Store`'s Signature attribute and in `GrainSilo`'s Signature attribute would respectively be:

```
<T:Ljava/lang/Object;>Ljava/lang/Object;
<T:LGrain;>LStore<TT;>;
```

Thus the bound of each type parameter, the specific LOOJ supertype, and the specific LOOJ superinterfaces (although there are none in this simple example) are are present in the current Signature attribute.

Now that we have described the information stored in the Signature attribute, figure 4.1 is a graphical representation of how information is stored in the class file. For simplicity, all the information in the constant pool should be considered to be strings. As described in this subsection, the inheritance hierarchy of the `Store` class is encoded in its LOOJ Signature attribute.

4.3 Limitations of the Signature Attribute

As hinted above, the current Signature attribute does not contain enough information to allow the bytecode verifier to effectively verify methods using LOOJ language features. Since the current Signature attribute only contains information for the type signatures of methods, fields, and classes, type information erased in the actual bytecode (i.e. that which is actually type checked by the verifier) during the translation process is actually lost. For example, the following Java code:

```
new Store<String>();
```

would translate to the JVML code:

```
new Store
dup
invokespecial Store/<init>()V
```

and, unfortunately, not to the following:

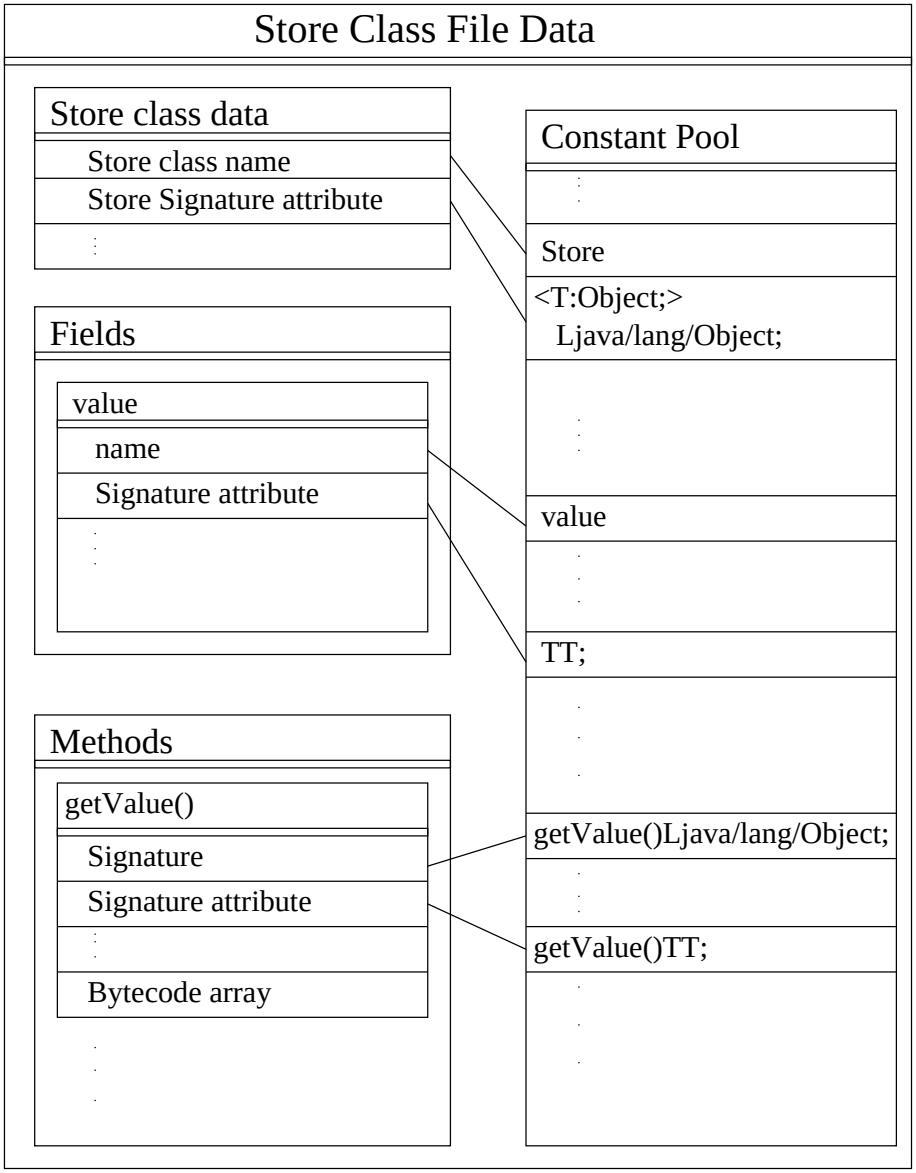

Figure 4.1: A graphical representation of information stored in a class file.

```
new Store<String>
dup
invokespecial Store<String>/<init>()V
```

which is what we would like during verification. That is, after simulating this set of instructions in the bytecode verification algorithm, our simulated operand stack should have a `Store<String>` on top of it and not simply a `Store`. This deficiency clearly prevents us from type checking a great deal of polymorphic code.

4.4 More Comments!

Without a very generous amount of footnotes, a James Joyce novel, though written in English, would be incomprehensible to contemporary American readers (and even with the footnotes can be utterly baffling). The footnotes explain nuances in language and historical figures that would have been recognized by Irish readers at the time of writing 100 years ago but whose significance has long since been erased from popular knowledge. In other words, they help explain what Joyce actually *meant*. The annotations inserted by the LOOJ compiler act in a very similar manner, but in this case the nuances in language are erased LOOJ types and not clever turns of phrase, and the reader is the LOOJ virtual machine. Our "footnotes", the attributes, explain to the VM what certain bits of bytecode actually *mean* and not what they simply appear to mean.

Thus our solution to the dilemma presented in the last section is to store the necessary information in more attributes within the class file. All that is needed is a set of annotations that would allow the verifier to recognize what type is actually *meant* to be represented in the bytecode where an erased, flat type actually appears. In the previous example, the `new` instruction actually references the flat type `Store` in the bytecode, but what it *meant* to reference was the LOOJ type `Store<String>`.

Unfortunately, one cannot insert comments into the middle of the bytecode array. Instead, the attributes are added to the method definition containing the bytecode. There is a set of new attributes that together give the verifier enough information to verify the bytecode using the LOOJ types that were present at compile time and not the erased, Java types that are present at run-time.

4.4.1 First Impressions are Lasting

Before plunging into the specification for the new set of attributes, we must figure out when erased information should be kept around. The answer is simply that any instruction that *introduces* a LOOJ type onto the operand stack requires an attribute specifying which LOOJ type should be used.

Once the LOOJ type is on the operand stack or in the local variable array, it is available for type checking for the duration of its existence. In other words, no further attribute information is needed to keep the LOOJ type in the verifier during simulated execution. The LOOJ type has, in effect, created a lasting first impression.

Any instruction that does not introduce a new type (one that manipulates information already on the operand stack) does not require annotation. For example, even though `instanceof` statements involving LOOJ types lose type information during translation, they do not push a LOOJ type onto the operand stack, and therefore the erased type information is not maintained in our attribute set. Clearly some instructions may involve type checking with LOOJ types, such as storing a LOOJ type on the top of an operand stack into an instance variable, but since the LOOJ type information is already present in the verifier no further attribute is needed to aid the type checking process.

In our previous example, the `new` instruction was shown to be an instruction that could introduce a LOOJ type onto the operand stack. The complete list of instructions is included below, and each instruction receives its own attribute containing enough information for the LOOJ bytecode verifier to type check it properly.

Although there are many new attributes, they all contain the same three pieces of information.

1. The name of the JVML instruction that should be, but is not currently, introducing a LOOJ type onto the operand stack.

2. The instruction's index into the method's bytecode array.

3. The index into the constant pool containing the Utf8 formatted string representing the LOOJ type to be used by the verifier in place of the type referred to by the bytecode itself.

When the compiler erases LOOJ type information, it creates the appropriate attribute and adds it to the appropriate method's definition.

4.4.2 The Forgetful Commands

We now list the situations and JVML instructions that introduce new types onto the operand stack or local variable array, and discover which of those requires more information than is provided by the current Signature attribute.

1. A LOOJ type can enter as a parameter to the method being verified. In this case, the current Signature attribute, which contains the method's LOOJ signature, can be parsed to discover the LOOJ type of the method parameter, so no further information is needed.

2. A LOOJ type can be introduced by accessing a field (static or instance variable) with a LOOJ type[2]. The `getfield` command grabs the value out of the specified instance variable and pushes it onto the operand stack. Analogously, `getstatic` is used to access static variable data. Just as with method parameters, Signature attribute information is already being produced by the LOOJ compiler for instance variable definitions, so whenever the type of an instance variable is to be pushed onto the operand stack the LOOJ type information is already available to the verifier. Again, no further information is needed.

3. As shown before, the `new` instruction may introduce a LOOJ type. Our first new attribute, the **NewLooj** attribute, specifies where in the bytecode the `new` instruction appears and what LOOJ type should be pushed onto the stack in place of the Java type the bytecode actually references.

   Unfortunately, the `new` instruction does not in itself create a new object. The `<init>` method must be called on the new object in order to instantiate it (before `<init>` is called, the memory pointed to by the new object reference contains random data).

   According to the JVM specification, the class containing the specific constructor called by `invokespecial` must be identical to the reference type on the top of the stack. For instance, if `A` is a superclass of `B`, then:

   ```
   new A
   dup
   invokespecial B/<init>()V
   ```

   is illegal and should fail verification. Unfortunately, this definition is not consistent if we annotate the `new` instruction without annotating its companion call to `<init>`. To see this, consider the following code.

   ```
   new Store<String>();
   ```

   This would be translated to the JVML:

   ```
   new Store
   dup
   ```

[2]Recall that local variables at the source level do not directly correspond to local variables in the JVM. At the start of verification, the local variable array contains only the `this` reference and the method's parameters, so any LOOJ types entering the local variable array must come from one of these other situations.

```
invokespecial Store/<init>()V
```

Using the **NewLooj** attribute information, our verifier would see this JVML as:

```
new Store<String>
dup
invokespecial Store/<init>()V
```

So the type on the top of the stack when reaching the call to `<init>` is different than `Store`. That is, according to the current JVM specification, the above JVML is illegal and should instead be:

```
new Store<String>
dup
invokespecial Store<String>/<init>()V
```

However, since in either case the same actual method code would be called at run-time to initialize the newly allocated object space (LOOJ is, after all, a homogeneous implementation of parametric polymorphism) there is no need to annotate the `invokespecial` instruction in the same way as the `new` instruction.

Thus we simply revise the JVM specification so that a constructor method called on a flat type can initialize any polymorphic instantiation of that flat type. This saves us the trouble of adding an unnecessary attribute.

4. `anewarray` creates a new array of the specified reference type (there are specific commands for creating numerical arrays). Thus an array of `String` would be created via the JVML code:

   ```
   anewarray [Ljava/lang/String;
   ```

   The **NewLoojArray** attribute contains the index into the bytecode array where the command can be found and the LOOJ array type to be pushed onto the operand stack.

5. `multinewarray`, which creates a multidimensional array, has the analogous **MultiLoojArray** attribute containing the usual information associated with it.

6. The four `invoke*` instructions reference methods that may return a LOOJ type which would be pushed onto the operand stack. In most cases, the return type can be found in the LOOJ type signature stored in the called method's Signature attribute and no further attribute information is necessary.

   One must also consider the translation of a call to a parameterized method. Consider the following situation:

```
class Foo {
    T foo<T>() { ... }
    public static void main(String args[]) {
        Foo     f = new Foo();
        String  s = f.foo<String>();
        Integer i = f.foo<Integer>();
    }
}
```

   Because of type erasure, both method calls to the polymorphic method `foo<T>()` would appear in the JVML as:

```
invokevirtual Foo/()Ljava/lang/Object;
```

   However, the Signature attribute would allow us to conclude that the method *really* meant to return a `T`, which in the first call is a `String` and in the second an `Integer`. As explained above, the Signature attribute does not contain enough information for the verifier to deduce this.

   The **PolyMethod** attribute specifies which type should actually be returned by the method (and, by extension, pushed onto the operand stack).

7. The `checkcast` instruction replaces the type on the top of the operand stack with the type that it is to be cast to and so could certainly introduce a LOOJ type. However, as shown in the last chapter, a type cast involving a LOOJ type is translated by by the LOOJ compiler to a more complicated expression. For convenience, that example is repeated again here.

```
(Store<String>)object;
```

   becomes:

```
Object $synth_1$;
((($synth_1$ = object) == NULL)
 ||
 ((Store)$synth_1$.$instanceOf$Store(
         new PolyClass(Integer.class)))
? (Store)$synth_1$
: throw new ClassCastException();
```

In this situation it is clear that `Store<String>` should be the type on the top of the operand stack after simulating the execution of that mess of an expression, but when it is translated to JVML that single line of Java becomes a whole set of JVML instructions.

Where we actually want the LOOJ type to be pushed onto the operand stack is certainly not after the call to `$instanceOf$Store()` (a method synthesized by the LOOJ compiler), but rather after the actual type cast of the synthesized variable `$synth_1$`. That one part of the large expression above fortunately corresponds to a single JVML statement:

```
checkcast Store
```

As usual, our next attribute, **CheckLoojCast**, contains the index into the bytecode array for the instruction and the actual (i.e. LOOJ) type that should be pushed onto the operand stack.

A quick examination of the remaining JVML commands will convince the reader that these are the only instructions that could possibly enter a new LOOJ type onto the operand stack or into the local variables. Thus the above set of attributes contains enough information for the verifier to type check bytecode using LOOJ types.

Now that the appropriate type information can be passed to the LOOJ bytecode verifier, we need to describe the modified type checking algorithm that takes advantage of this information to pass code lacking the stupid casts. This is done in Chapter 5.

4.5 Bridge Methods

Removing stupid casts is only one of the two optimizations we hope that our enhanced JVM will provide. The other is to eliminate the overhead created by bridge methods and there is currently no way for the VM to recognize a method as a bridge method.

As before, we turn to attributes to convey that information. When a bridge method is created by the compiler it adds a **BridgeMethod** attribute

that includes an index into the constant pool containing a `MethodRef` that references the method to which our bridge method bridges.

For example, consider the following LOOJ source.

```
class Store<T> {
    T value;
    public void setValue(T newVal) {
        value = newVal;
    }
}

class GrainSilo<T extends Grain> extends Store<T> { ... }
```

The translation of `GrainSilo`, with an added bridge method, is:

```
class GrainSilo extends Store {
    void setValue(Grain newVal) {
        value = newVal;
    }

    /* Bridge Method */
    void setValue(Object newVal) {
        this.setValue((Grain)newVal);
    }
}
```

Figure 4.2 contains a representation of how the BridgeMethod attribute stores information in the class file. Note that the bridge method is *not* the new version of the method in `GrainSilo`, which takes a `Grain` as a parameter, but rather the inherited version with the original signature, which takes an `Object` as a parameter. Recall that the latter version is synthesized to ensure that method calls statically bound to it are forwarded to the new, correct version of the method.

What exactly the LOOJ virtual machine does with this information to minimize the run-time cost of bridge methods is explained in Chapter 5.

4.6 Summary

This chapter explained how the Signature attribute is used by the LOOJ compiler to allow for separate compilation of source files. It also explained how that same information can be used by the verifier for some of the type checking that it performs. Unfortunately, we saw that the Signature attribute was not sufficient by itself. To complement the Signature attribute we created a small set of attributes that together form a complete set of

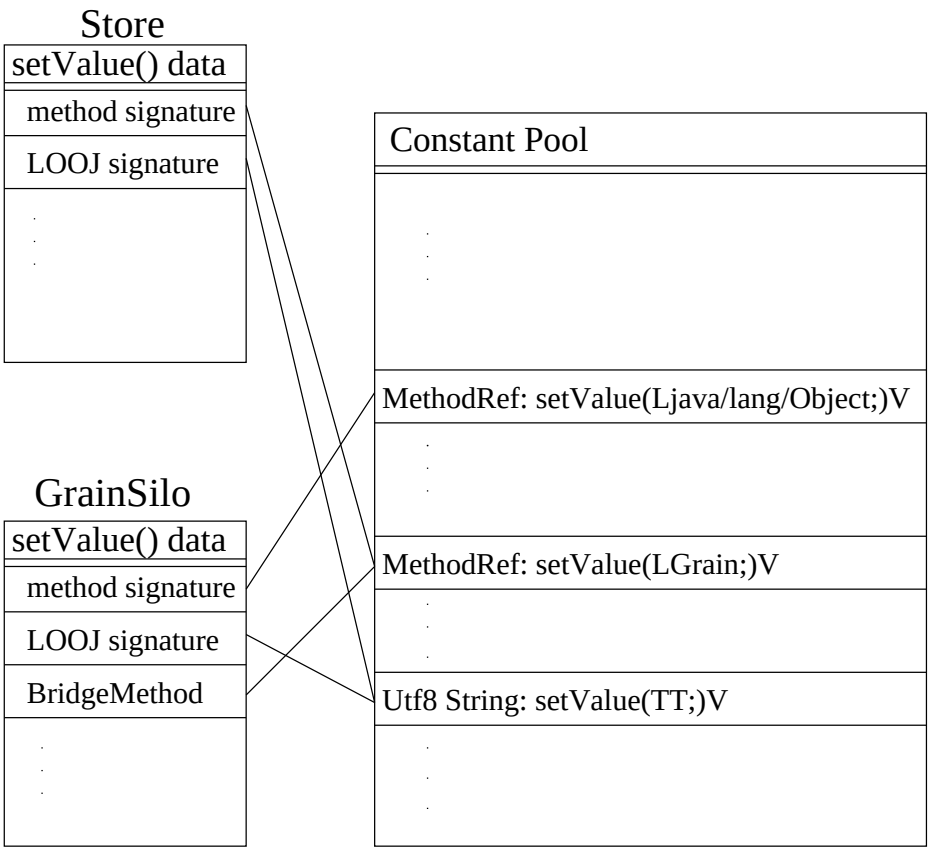

Figure 4.2: A graphical representation of the information stored in a BridgeMethod attribute.

type information needed by the LOOJ bytecode verifier. We also used attributes to label bridge methods for method call optimization in the VM. The optimization will be describe in Chapter 5.

After modifying the LOOJ compiler, the new set of LOOJ attributes annotating class files is the following:

- Signature
- BridgeMethod
- NewLooj
- NewLoojArray
- MultiLoojArray
- CheckLoojCast
- PolyMethod

Now that we can provide the verifier with the knowledge of the erased LOOJ type information, we must describe our modified type checking algorithm.

Chapter 5

Introducing LOOJVM

The public is more familiar with bad design than good design. It is, in effect, conditioned to prefer bad design, because that is what it lives with. The new becomes threatening, the old reassuring.

– Paul Rand

This chapter presents the design and implementation of our JVM modification. Before discussing the details of our design, we will restate our goals. They are:

- Implement parametric polymorphism in the JVM. This implementation should allow for code not containing stupid casts to be run safely and efficiently.
- Bridge method invocations should not contain the usual overhead.
- Implement LOOJ language features, including `ThisType`, `ThisClass`, and exact types, in the JVM.
- Include the above features in the JVM without affecting the run-time speed of classes not using LOOJ type features.
- Include the above features in the JVM in such a way as to run faster than GJ or LOOJ code currently runs.
- Do all of the above without modifying the JVM's run-time environment, as that would require the potential modification of complicated algorithms such as garbage collection and JIT optimizations.

We will start with an overview of the algorithm's design. We then proceed to examine the type-checking rules in the LOOJ source language and

describe algorithms that implement these rules in the modified VM. Next, we describe the optimization used to alleviate the cost of bridge methods. We then restate the preprocessing algorithm and summarize the design details. We conclude with an informal argument that our modifications to the Java bytecode verifier's type system do not create loopholes that may be taken advantage of by malicious code.

5.1 Overview of Algorithm Design

The philosophy behind our design is to use as much of the existing structure of the JVM as possible so that modifying existing implementations is relatively easy. Recall that type checking essentially involves utilizing the class pool and hierarchy so that when a type (or string representation of the signature of a type retrieved from the constant pool) is encountered, we can simply get the class from the hashtable and traverse the class hierarchy. Therefore, our design involves adding LOOJ types, such as instantiations of a polymorphic class, to the class hierarchy so that they can be used to type check just as normal types are.

Therefore, our algorithm must add all LOOJ types to the class pool and class hierarchy. It does this in a preprocessing step directly before pass 3 of bytecode verification. Since pass 3 is performed on all classes before they are used regardless of how they are loaded, this preprocessing algorithm is applied to all classes with LOOJ signature attributes.

The basic specification for preprocessing a class is as follows:

1. Process the class's LOOJ signature, if one exists. As described in Chapter 4, the LOOJ type signature for a class contains a list of its type parameters and their bounds, its LOOJ superclass signature, and the LOOJ signatures of any implemented interfaces.

   If an interface is said to be implemented exactly, check that it is implemented exactly (i.e. that the class has exactly the set of methods the interface promises, constructors, and no other methods) and record that the implementation is exact.

   Add new LOOJ types encountered to the hierarchy. The algorithm for adding a new LOOJ type to the type hierarchy is described in subsection 5.3.1.

2. For each field definition, process LOOJ attributes for the field and save Signature information with its representations in memory. During type checking of field access, the LOOJ type will be used in lieu of the normal type. Add new LOOJ types encountered to the hierarchy.

3. For each method definition, process LOOJ attribute method information as follows.

(a) Process the Signature attribute, if present. Add new LOOJ types encountered to the hierarchy.

(b) Process bytecode annotations (these include many of the new attributes, such as NewLooj, described in Chapter 4). Save information from these attributes so that it is available during bytecode verification of the method. After verification, information from bytecode annotations may be forgotten.

As usual, add new LOOJ types encountered to the hierarchy.

(c) Process the BridgeMethod attribute, if present. Ensure that the LOOJ signature for this method is identical to the LOOJ signature of the bridge method (this may require preprocessing the method to be bridged to). Then cause both methods to share bytecode in the VM. Thus instead of forwarding the method call, a bridge method becomes an alias to the same bytecode.

The details of this algorithm will be explained throughout this chapter.

5.2 Why Reinvent the Wheel?

The basic idea behind the new type checking algorithm is to utilize the existing type checking framework as much as possible. We build the LOOJ type hierarchy within the standard Java type hierarchy, creating and inserting LOOJ types into the class pool as they appear. Therefore, whenever the verifier asks the question "can β be used as an α", it uses the same techniques, including searching the supertype hierarchy and implemented interfaces hierarchy.

For example, the superclass of type `GrainSilo<Grain>` would be type `Store<Grain>`, so a search for their first common superclass would yield `Store<Grain>`. Conversely, `Store<String>` and `Store<Integer>` have only the common superclass `Object`.

Just as before, to get a handle on the class `Store<String>`, one would simply have to look in the class pool, using "`LStore<String>;`" as the key. The question arises: when and how do we build the LOOJ hierarchy? Before answering this, we first discuss the type-checking rules LOOJ uses on its language extensions.

5.3 LOOJ Type Checking Rules

This section reviews LOOJ type checking rules and explains how they are implemented in the virtual machine. The next section explains how the LOOJ type hierarchy is constructed.

The next four subsections detail the type rules for polymorphic types, type parameters, exact types, `ThisType`, and `ThisClass`. Each of those

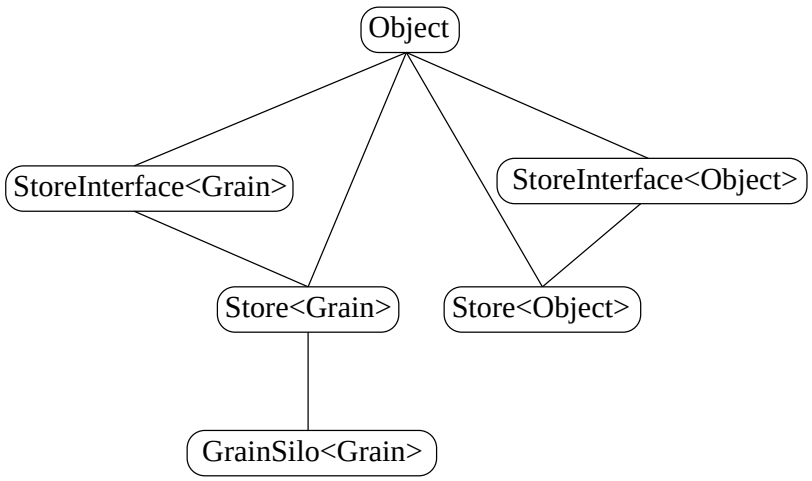

Figure 5.1: An illustration of polymorphic extension rules.

subsections includes both the type rules that apply in the source language and how those rules are implemented in the modified virtual machine.

5.3.1 Polymorphic Types

In order for one instantiation of a polymorphic class to be an extension of another instantiation of a (possibly different) polymorphic class, the corresponding type parameters must be *identical*. For example:

```
/* Correct Extensions */
GrainSilo<T>      extends    Store<T>
GrainSilo<Object> extends    Store<Object>
GrainSilo<Grain>  extends    Store<Grain>
Store<Grain>      implements StoreInterface<Grain>

/* NOT Correct Extensions */
GrainSilo<Grain>  extends    Store<Object>
Store<Grain>      extends    Store<Object>
Store<Grain>      implements StoreInterface<Object>
```

Thus even though `Grain` extends `Object`, `GrainSilo<Grain>` does not extend `Store<Object>`, and `Store<Grain>` does not extend `Store<Object>`. These relationships are illustrated in figure 5.1.

To illustrate what exactly is meant by by "corresponding type parameters", consider the following example of what they are not.

```
class A<T> { ... }
class B<T> extends A<String> { ... }
```

In this case, `B<Grain>` is a subtype of `A<String>` because the type parameter `T` in `A` has nothing to do with the type parameter `T` in `B`.

So in order for a type parameter in some polymorphic subclass `B` of a polymorphic class `A` to correspond to a type parameter in `A`, it must be used to instantiate the value of a type parameter in `A`.

Now consider the following trivial example.

```
class A<W> { ... }
class B<T> extends A<T> { ... }
```

In this case, `T` in `B` corresponds to `W` in `A` since `T` is used to instantiate `W`.

In the VM

As stated earlier, we plan on using the existing type checking framework as much as possible in order to avoid any substantial redesign. Therefore we would like to be able to lookup the type `Store<Grain>` in the class pool by its string signature just as we would for a normal class. Fortunately, each instantiation of a polymorphic type has a unique type signature, so string lookup works as usual.

Now we must explain how we type check polymorphic classes in the bytecode verifier. As explained in the type checking rules above, one polymorphic instantiation may extend another (assuming one extends the other at the source level) only if their corresponding type parameters are identical. Their type parameters are identical only if the string signatures of the corresponding type parameters are identical.

Since we are using the existing type checking framework, all we have to do then is correctly add polymorphic instantiations to the class hierarchy. Given that:

```
SomeClass<ABCD,...> extends SuperClass<BD,...>
```

where `A`, `B`, etc. refer to the string signatures of the type parameter instantiations (for instance, `A` = "LGrain;"), we use the following algorithm:

1. Ensure that the classes and interfaces used to instantiate the polymorphic class do not violate the declared bounds of the class's type parameters. If they do, raise a `TypeParameterBoundException`.

2. Use the type signature of the polymorphic instantiation to insert the new class into the class pool and class hierarchy. This class is not a full, heterogeneous class, but rather a lightweight wrapper specifically used for type checking in pass 3 of verification, containing information about its type parameters, pointers to classes it extends and interfaces it implements (so that it can be used for type checking), and a pointer to its flat class (recall that we are using a homogeneous implementation

strategy, so all method code and static variables are in the flat class and are shared by all instantiations of the polymorphic class).

The data contained in this lightweight class used for type checking is described in more detail in section 5.4.

3. Construct the correct superclass signature. The signature starts with an "`L`" and is followed by the superclass's name. It then contains a "`<`" and a list of the type signatures of the superclass.

   Fill in the type signatures of `SuperClass`. We can find out which type parameters in `SomeClass` correspond to others in `SuperClass` through the LOOJ signature of `SomeClass` described in subsection 4.2.1. End the type signature with "`>;`".

4. Lookup the superclass in the class pool and class hierarchy. If the superclass does not exist, add it to the hierarchy starting at step 1 above. Fill in the class's pointer to its superclass.

5. For each superinterface, construct its correct type signature in the same way the superclass signature was constructed. Add the interface to the type hierarchy from step 1 if not already present and fill in the class's pointer to the interface.

Once the type hierarchy for a particular polymorphic instantiation is setup in the class hierarchy, type checking works identically to the current model. That is, if one wants to see whether some polymorphic type or instantiation β can be used as an α, we simply follow the hierarchy up as usual.

5.3.2 Type Parameters

A type parameter is treated as an extension of its bound in the source language. For example, in our `GrainSilo` example any variable that is declared to be of type `T` is actually treated as if it were an extension of `Grain`. This behavior is reflected in the class's erasure. This aspect of type checking type parameters is pretty straightforward.

Things are a little more complicated when we are type checking a method call to an object that is statically bound to a particular instantiation of a polymorphic type. Consider the following example:

```
GrainSilo<Wheat> g = new GrainSilo<Wheat>();
...
Wheat w = g.getValue();
```

The method call to `getValue()` is statically bound to a method with signature `getValue()TT;`. However, since `g` is statically a `GrainSilo<Wheat>` it

clearly should not return a `Grain` or a `T`, which may not have any meaning in the current scope.

Thus if a type parameter is statically instantiated, we must use its instantiated static value in place of its bound for type checking.

In the VM

Unlike polymorphic instantiations, type parameters are types that do not correspond directly to any type in the type hierarchy. That is, if we encounter the type signature `TT;`, which represents a type parameter, then we cannot simply lookup `T` in the class pool and use what we find to type check variables of type `T`. Instead, we use its bound or, if available, its instantiated type, just as in the source language.

For example, in our `Store<T>` class, `getValue()` returns a `T`. Consider the following code:

```
Store<String> store = new Store<String>();
store.setValue(``foo'');
String s = store.getValue();
```

Through the `getValue()`'s Signature attribute, we know that its LOOJ type signature is `()TT;`. However, we cannot simply look up the bound of `T` in `Store` since, in our current situation, the meaning of `T` has been refined through its instantiation as `String`.

We will now formalize this rule. During type checking, when a type parameter is encountered:

1. If the type parameter is encountered by accessing a field or method in an instantiation of a polymorphic type, substitute the instantiated type of the type parameter. We note that polymorphic instantiations must therefore store their type parameters' instantiated values and not simply superclass and superinterface pointers.

   If the instantiated type found is another type parameter, repeat this step until an actual type is found.

2. If the type parameter is a parameter from the class containing the method currently being verified, or from the method itself in the case of a polymorphic method (in other words, if it is an uninstantiated type parameter), then treat it as if it could be any extension of its bound.

The second rule deserves some minor clarification. If a type parameter `T`'s bound is α, we cannot assign an α into a variable of type `T` because `T` may be an extension of α. The value in a variable of type `T` can, however, be assigned to a variable of type α.

To continue with our previous example, using this algorithm the method call to `getStatic()` is received by an instantiation of a polymorphic class. We consult the data in this instantiation and find that `T` is really `String` and therefore substitute a `String` for the return value of the method.

Thus when making a method call or accessing a field from an instantiation of a polymorphic type, we have to be careful to use the instantiated type and not simply the declared bound of the type parameter.

5.3.3 No Stupid Type Casts: LOOJ at Work

The following is our first example of the bytecode verification using LOOJ type information. It is also an example illustrating that stupid type casts are unnecessary with a smart type checker. As such, we will review all the steps involved in type checking.

Consider our previous example. It is repeated here for convenience.

```
Store<String> store = new Store<String>();
store.setValue(‘‘foo’’);
String s = store.getValue();
```

When we translate this to bytecode, we would normally get

```
new Store
dup
invokespecial Store/<init>()V
dup
ldc "foo"
invokevirtual Store/setValue(Ljava/lang/Object;)V
invokevirtual Store/getValue()Ljava/lang/Object;
checkcast Ljava/Lang/String;
```

The `dup` instructions above exist because non-static method invocation instructions pop the receiving object reference off the stack.

Although nothing is done with the `String` retrieved from `getValue()` at the end of the method, we see that there is a type cast inserted to ensure that the object on the top of the stack is recognized to be of type `String` and not `Object`, the type supposedly return by the erased version of the `getValue()` instruction.

Now suppose we remove the stupid cast.

```
new Store
dup
invokespecial Store/<init>()V
dup
ldc "foo"
invokevirtual Store/setValue(Ljava/lang/Object;)V
invokevirtual Store/getValue()Ljava/lang/Object;
```

We will now proceed to verify this bytecode and will end up with a `String` on top of the stack at the end, and this without the extra type cast.

Through the NewLooj attribute, we know that the first line should be:

```
new Store<String>
```

When we reach it, we can therefore push the type `Store<String>` onto the operand stack instead of the incorrect flat type `Store`. The `ldc` instruction simply pushes an object reference to the string "foo" onto the operand stack. For verification purposes, we do not care what the value of that string is, only that the type of reference being pushed onto the stack is `Ljava/lang/String;`. So now we have an object of type `String` on top of the stack, followed by an object of type `Store<String>`.

The next instruction looks in the `Store` class for the method `setValue()` with the given type signature. When it finds this method, our verifier notices that it is annotated with a Signature attribute stating that its actual type signature is:

```
setValue(TT;)V
```

When we type check the method call, we therefore just check that the reference type on the top of the stack, which is currently a `String`, can be used as a `TT;`.

To discover what `T` means in the current context we need to check what it has been instantiated with in the receiving object. In this case, the receiving object (the next thing on the stack after the parameters to the method being called) is `Store<String>`, in which a `T` is really a `String`. So this method call is OK.

A similar line of reasoning allows us to push another `String` onto the stack for `getValue()`'s return type. Therefore, at the end of verification, these lines produce a `String` on the top of the operand stack without the stupid type cast, as desired.

Unfortunately, a formal proof that it is safe to remove all type casts inserted by GJ or LOOJ is beyond the scope of this thesis, and would probably make a pretty good doctoral thesis.

5.3.4 Exact Types

Recall that if an object is exactly an α, then it is not an extension of α. So to check if a β can be used as an $@\alpha$, then we simply have to check if $\beta = @\alpha$. Some might be confused with the above rule. For example, why is it that we cannot simply check $\beta = \alpha$? The answer is that whenever the type α is encountered, it represents any extension of α, whereas the type $@\alpha$ represents exactly the type α, so the two signatures are in fact distinct.

A class implements an interface exactly if it declares that it implements the interface exactly, contains all the methods from that interface, any number of constructors, and no other methods. For example:

```
interface SelfStoreInterface {
    void setValue(@ThisType newValue);
    @ThisType getValue();
}

class StoreA implements @SelfStoreInterface {
    private @ThisType value;
    ...
    public StoreA() { ... }
    public StoreA(@ThisType value) { ... }
    public StoreA(int foo) { ... }
}
class StoreB implements @SelfStoreInterface {
    private @ThisType value;
    ...
}
```

Both `StoreA` and `StoreB` implement exactly `SelfStoreInterface`, so anywhere an exact `SelfStoreInterface` is expected we can safely use either `StoreA` or `StoreB`.

In the VM

Type checking exact types during pass 3 of bytecode verification requires us to keep track of which types are known to be exact. The first thing we must do to accomplish this is to keep track of which interfaces are implemented exactly by a given class.

During verification, we keep a set of flags for the types stored on the simulated operand stack and a set of flags for the types stored in the local variables. The flags denote which types are known to be exact. It is important to keep track of every type that is known to be exact. Aside from those entering the data flow analysis as method parameters, as the return type from a method invocation, or as the type of an instance variable that is pushed onto the stack, any object creation instructions create exact types, such as `new` and `newarray`.

Type checking using exact types can be formalized as follows:

1. When performing a merge operation, the exact flags are propagated unless two reference types do not agree and need to be merged into a superclass. In that case the flag is set to false in the resulting operand stack.

2. When checking if a β can be used in place of an α, we check if α is an exact type. If so, and $\beta = @\alpha$ (i.e. $\beta = \alpha$ and β is known to be an exact type) then type checking succeeds. Otherwise it fails.

If α is not an exact type, the value of β's exact flag is ignored.

Thus type checking exact types relies on our keeping track of which types are known to be exact.

5.3.5 `ThisType` and `ThisClass`

Even though `ThisType` and `ThisClass` are basically specialized type parameters, they introduce complications not found when type checking normal type parameters. This subsection first describes the basic type rules[1] involving `ThisType` and `ThisClass` and explains how those type rules are implemented in the VM.

The Basic Rules

The type rules for `ThisType` and `ThisClass` are pretty straightforward. If those keywords appear in a class `C` with exact interface `IC`, then the rules are:

```
@ThisClass this
ThisClass  extends    C
ThisType   extends    IC
ThisClass  implements @ThisType
```

The first rule states that `this` is a `@ThisClass`. However, we should elaborate on the rules regarding access to private instance variables. `this` has access to private variables, even though the meaning of `this` changes when methods are inherited. To remain consistent with Java's current approach, `ThisClass` also has access to a class's private variables.

The next two rules are more straightforward. `ThisClass` refers to `C` or any class that inherits from `C`, so it clearly extends `C`. Similarly, in `C` it is pretty clear that `ThisType` extends `IC`.

The final rule trivially states that `ThisClass` implements `@ThisType`.

In the VM

Fortunately, we do not need to make special cases to type check `ThisClass` and `ThisType` in the virtual machine, as both can be viewed as type parameters whose bound is constrained in every subclass. We treat the bound of `ThisClass` to be the class that it appears in, or the particular instantiation

[1] Some of the details described here differ from type rules presented in [Fos01]. This is due to undocumented refinements to the type system since Foster's thesis. For instance, a variable of type `ThisType` can no longer access public instance variables. The rules presented in this thesis should therefore be considered more accurate where discrepancies exist.

of a parametric class that it is used by (in the same way that we know what type a type parameter really represents in an instantiation).

Similarly, the bound of `ThisType` is the public interface of the class that it appears in. The LOOJ compiler generates an exact interface implemented by a class if an exact interface is not supplied. This interface is used within the type checker as a bounding interface for `ThisType`.

In cases where the exact interface implemented by a class in unavailable at link time, the virtual machine should generate the interface itself if that class (or any superclass or superinterface of that class) uses `ThisType`.

The formal type checking rules for `ThisType` and `ThisClass` are the same as for bounded type parameters as described in subsection 5.3.2.

5.3.6 An Example: Verifying More Complex LOOJ Types

The more features we add to the type system of a language, the more difficult it is to create examples that illustrate all the features in a small amount of space. Here we will verify the `produce()` method of a self-replicating factory class to illustrate how we type check exact types and `ThisType`.

Consider the following code:

```
interface Factory<T> {
    public @T produce();
}

class Replicator<T> implements Factory<ThisType> {
    public @ThisType produce() {
        @Replicator<T> r = new Replicator<T>();
        return r;
    }
}
```

This simple example does *not* pass static type checking, and will fail bytecode verification. In doing so, it illustrates the concepts we wish to show. The `produce()` method translates to the following JVML:

```
new Replicator
dup
invokespecial Replicator/<init>()V
areturn
```

For the sake of repetition, when the analyzer looks at the bytecode array it substitutes the LOOJ types it receives via the LOOJ attributes for the types in the bytecode. Thanks to the NewLooj attribute, the verifier sees the above JVML as the following:

```
new Replicator<T>
dup
invokespecial Replicator/<init>()V
areturn
```

When we initialize the bytecode analyzer, it starts off in the state below.

```
    Operand Stack             Locals
-------------------------------------------  (0)
    unstable                  @ThisClass
    unstable
```

Notice how we are using `ThisClass` instead of `Replicator` for the local variable reference analogous to `this`, and also notice that it is an exact reference.

Now we simulate execution.

- The `new` instruction pushes a reference of type `@Replicator<T>` onto the operand stack and the `dup` instruction duplicates that reference type.

  ```
      Operand Stack             Locals
  -------------------------------------------  (1)
      @Replicator<T>             @ThisClass
      @Replicator<T>
  ```

  As mentioned earlier, we allow an `<init>` method to initialize any instantiation of the type on which the method is called. Thus the method invocation pops off the reference on the top of the stack and initializes it.

  ```
      Operand Stack             Locals
  -------------------------------------------  (2)
      @Replicator<T>             @ThisClass
      unstable
  ```

- The `areturn` instruction returns the object reference on the top of the operand stack (if the stack were empty or if the type on the top were a primitive type, verification would fail). Here we must check if `@Replicator<T>` can be used as a `@ThisType`, which is the return type of `produce()`. Recall that `ThisType` refers to any *extension* of the exact interface implemented by the currently executed object. Therefore, in a subclass of `Replicator` with additional methods it would be illegal to use a `@Replicator<T>` as a `@ThisType`! This failure was explained in more detail in subsection 5.3.2.

  Thus our type checker has correctly proved that `produce()` fails verification because `@Replicator<T>` cannot be used as a `@ThisType`.

Finally, recall that, in the implementation, the type `@Replicator<T>` is really a `Replicator<T>` on the simulated operand stack, but that the set of exact type flags tells us that it is an exact type during type checking.

5.4 Modifying the Class Loader

In the following sections we describe the modifications made to the class loader and the new preprocessing step added for bytecode verification that prepares a class's LOOJ type hierarchy for bytecode verification.

Chapter 4 described how LOOJ type information is maintained from the source language to the verifier. The class loader must be modified to recognize all LOOJ attributes and the `Class` run-time class representation must be modified to be able to hold this information after class loading.

This section describes what extra data is put in each `Class` object during load-time. As we are creating a modification specification that should be applicable to any current JVM, this discussion is intentionally vague. Specific code snippets are not provided, nor is specific data structure information. All that is shown is what information should be kept in order to complete type checking in pass 3 of verification.

Essentially, our modified class loader has to recognize all LOOJ attributes and save that information somewhere for verification. As Chapter 4 explains what LOOJ information is maintained through attributes in the class file, we will not repeat that information here. Suffice it to say that all attribute data needs to be stored and accessible for type checking during verification.

Some of the attribute information should be around after verification completes. For instance, if a method with a LOOJ type signature is called from another class that is to be verified, we need to use that LOOJ type information to type check the parameter types passed to the method as well as to push the correct return type onto the simulated operand stack.

The attribute data that is not needed after verification completes is the set of attributes describing method bytecode and the BridgeMethod attribute. Since the bytecode has passed the one-time verification process, all details regarding LOOJ type information in the bytecode and BridgeMethod attribute information may be thrown out to save space. The reason that the BridgeMethod attribute is unnecessary post verification will become clear when class preprocessing is described.

In summary, prior to passing verification a `Class` object must contain the following information:

- Pointers to superclass and superinterfaces.
- The LOOJ type signatures of the class itself, its fields, and its methods.
- A list of the class's type parameters and their corresponding bounds.

In the case of the current `Class` representing an instantiation of a polymorphic class or interface and not merely a flat class, pointers to the instantiation types of its type parameters are needed. Furthermore, a pointer to the flat version of the class is needed.

- Aside from its LOOJ type signature, each method must maintain the following information.
  - A list of bytecode indices modified by LOOJ attributes and the correct LOOJ types associated with those indices.

    In other words, information in an attribute such as NewLooj should be maintained with the method's data in memory until the method passes verification.
  - A list of its type parameters and their bounds if the method is polymorphic.
  - A pointer to the method it bridges to if the method is modified by a BridgeMethod attribute.

To reiterate, our approach is a homogeneous design, but, not unlike the lightweight wrapper classes in NextGen, the lightweight LOOJ classes representing instantiations of a parameterized type which are inserted into the type hierarchy contain only the instantiation-specific information described above that is needed for type checking. The majority of the class's information is contained in the flat class in memory.

5.5 Preparing for Verification

The type hierarchy is necessary for bytecode verification and, as mentioned above, we take advantage of the simplicity of the current model of type checking in our design. However, we have not yet described how we build that type hierarchy. Its construction occurs just before pass 3 of verification so it is effectively a preprocessor for the pass 3 of verification.

The class loader received the LOOJ type information from the class file but did nothing with it other than store it in some reasonable way with the class's data in memory. Therefore, there still are no LOOJ types within the VM's type hierarchy after class loading. This is different from the traditional way the class hierarchy is built, where classes are added to the hierarchy during the class loading phase.

We should emphasize that any LOOJ types added to hierarchy during our preprocessing algorithm are *not* used a run-time. They only exist in the hierarchy to facilitate type checking of LOOJ code.

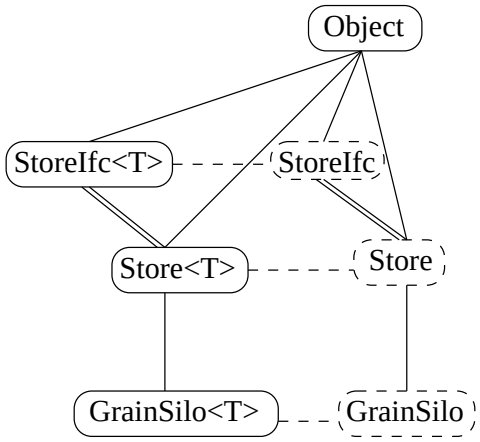

Figure 5.2: An example type hierarchy.

5.5.1 The LOOJ Type Hierarchy

The preprocessing algorithm that constructs the LOOJ hierarchy utilizes another algorithm to insert types into the existing type hierarchy. This algorithm was actually described in subsection 5.3.1 with the description of how polymorphic types are inserted into the LOOJ type hierarchy.

As mentioned above, type parameters, `ThisType`, `ThisClass`, and exact types do not have specific wrapper classes associated with them in the type hierarchy. Instead, as they are encountered they are treated in context by the type checker using the algorithms described in previous sections.

As an example of a LOOJ type hierarchy, consider the type hierarchy in figure 5.2. Note that the flat types are surrounded with dotted lines. This is to underscore the important point that flat types are not allowed in LOOJ, and are therefore not used during verification. In fact, if bytecode attempts to use a flat class as a type it fails verification.

The dotted line from `Store<T>` to `Store` is a pointer from the LOOJ type `Store<T>` to its flat type `Store`. The other lines out of `Store<T>` point to the class it extends and the interfaces it implements, including the exact interface it implements, `StoreIfc<T>`, which is generated by the LOOJ compiler. The exact nature of this implementation is underscored by a double line.

Now, suppose we encounter `GrainSilo<Grain>`, the instantiated polymorphic type of the flat class `GrainSilo`, during verification. As it does not already exist in our type hierarchy, we need to add it as well as any superclasses or superinterfaces not already present to the type hierarchy as described in subsection 5.3.1. After adding all the necessary classes by following the above algorithm, we end up with the type hierarchy in figure 5.3.

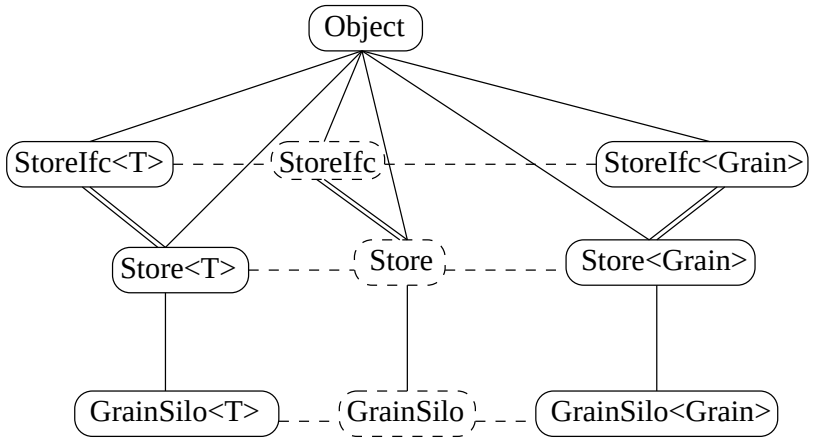

Figure 5.3: The type hierarchy after encountering `GrainSilo<Grain>`.

To summarize, when we see a new LOOJ type, we add it to the class pool and add all necessary but missing types to the hierarchy as well. Inserting a new LOOJ type into the type hierarchy is therefore a recursive process.

A Note on Implementation

If the reader has been thinking about implementation details while reading about type signatures and the type hierarchy, then he has noticed that the JVM and our attributes rely exclusively on strings to describe their type system.

As hinted at in the algorithm for adding a new polymorphic instantiation in subsection 5.3.1, when we first encounter a new LOOJ type we have to do some fairly intricate parsing of the type signature. Information such as the bounds for type parameters in an instantiation of a polymorphic class is obtained this way.

Again, we are not going to delve into specifics regarding the parsing of type data from LOOJ signatures. The full signature specification can be found in [Fos01]. Implementation specific parsing methods can be devised from that specification to extract type information.

5.5.2 Optimizing Bridge Method Calls

If encountered in a method descriptor, the BridgeMethod attribute is processed. Recall that a bridge method is simply a method that acts to forward a dynamically bound call to the correct method code (i.e. it ensures that methods in a superclass are overridden, not overloaded). It also inserts stupid type casts to make the forwarding method call possible.

Before making any modifications to the method body, we first check that the method to be bridged to is basically compatible with the current method. As illustrated in figure 4.2, the LOOJ signature of a bridge method and the method it bridges to should be equivalent, even though their actual, Java signatures are not.

Once the forwarding call is deemed legal, we replace the code for the method forwarding with the code for the intended method to be called. That is, both the bridge method and the method to be bridged to contain identical bytecode after this step.

In many implementations this is trivial as a method object's representation in memory contains a pointer to the array of bytecode to be executed. If this is not the case, then a method's body should be copied, which is a quick operation and a one-time cost.

Thus a bridge method becomes an alias for the actual method code that is intended to be executed, saving an extra method call every time the bridge method is called.

As a final point, we should note that there is no easy way to determine if a method is *actually* a bridge method or if a faulty attribute has been added by a malicious virus. In the latter case, since we are verifying all method calls to this method and the bytecode of the method to be bridged to, the method is still checked just as thoroughly by the verifier algorithm.

5.5.3 The Preprocessing Algorithm

We now repeat the specification for the preprocessing algorithm. At this point it should make more sense to the reader.

1. Process the class's LOOJ signature, if one exists. As described in Chapter 4, the LOOJ type signature for a class contains a list of its type parameters and their bounds, its LOOJ superclass signature, and the LOOJ signatures of any implemented interfaces.

   If an interface is said to be implemented exactly, check that it is implemented exactly (i.e. that the class has exactly the set of methods the interface promises, constructors, and no other methods) and record that the implementation is exact.

   Add new LOOJ types encountered to the hierarchy. The algorithm for adding a new LOOJ type to the type hierarchy is described in subsection 5.3.1.

2. For each field definition, process LOOJ attributes for the field and save Signature information with its representations in memory. During type checking of field access, the LOOJ type will be used in lieu of the normal type. Add new LOOJ types encountered to the hierarchy.

3. For each method definition, process method LOOJ attribute information as follows.

   (a) Process the Signature attribute, if present. Add new LOOJ types encountered to the hierarchy.

   (b) Process bytecode annotations (these include many of the new attributes, such as NewLooj, described in Chapter 4). Save information from these attributes so that it is available during pass 3 verification of the method. After pass 3 verification, information from bytecode annotations may be forgotten.

   As usual, add new LOOJ types encountered to the hierarchy.

   (c) Process the BridgeMethod attribute, if present. Ensure that the LOOJ signature for this method is identical to the LOOJ signature of the bridge method (this may require preprocessing the method to be bridged to). Then cause both methods to share bytecode in the VM. Thus instead of forwarding the method call, a bridge method becomes an alias to the same bytecode.

In summary, the LOOJ type hierarchy is built during the preprocessing stage of type checking and not during class loading. Once the class hierarchy is constructed, type checking during bytecode verification is performed based on the rules described in this chapter for LOOJ's non-polymorphic language extensions, including exact types, `ThisType`, and `ThisClass`.

5.6 The Verifier Works: An Informal Argument

Sun is nervous about modifying the bytecode verifier because, during the early days of Java, a number of flaws in the verifier's implementation were discovered that allowed unsafe code to execute on JVMs. A formal proof of correctness for our verifier's type system is unfortunately beyond the scope of this thesis. Indeed, Freund, one of the researchers who worked on Agesen et al.'s heterogeneous proposal described in Chapter 2, wrote his PhD thesis on a formal proof of correctness for the current bytecode verifier and developed a verifier based on this specification. That being said, we will try to argue briefly that the removal of the type casts normally inserted by GJ and by LOOJ through erasure is indeed a safe thing to do.

The basic argument is pretty simple: LOOJ knows the casts it inserts are guaranteed to succeed, so we are removing casts that are, in effect, doing no work in the first place (hence "stupid" type casts). Moreover, the new type checking done during bytecode verification is performed under the LOOJ type rules using the source-level, static LOOJ type information. We know that the LOOJ language is statically type safe, so if the verifier uses the same type system then it must also be statically type safe.

Therefore, we believe that our modified bytecode verifier, whose type system is the same as the statically type safe LOOJ programming language, does not introduce any loopholes that might be taken advantage of by malicious code, but due to time constraints we cannot prove this claim more concretely than we have done via this informal argument.

5.7 Summary

This chapter explained our new LOOJVM type checking algorithm, reviewing the type checking rules associated with LOOJ types and detailing how those rules are implemented in the LOOJVM. It also included the preprocessing algorithm that builds the LOOJ type hierarchy used by the type checker. An example illustrating how the enhanced type checker can verify LOOJ bytecode not containing stupid casts was presented. The algorithm optimizing bridge method calls was also given. Finally, it concluded with an informal argument that our modifications to the bytecode verifier do not introduce loopholes that could be taken advantage of by malicious code.

The next and final chapter evaluates the JVM modification presented in this thesis and includes some future work that needs to be done.

Chapter 6

Conclusions

I have not failed. I've just found 10,000 ways that won't work.

– Thomas Edison

We have examined other proposals that add parametric polymorphism to Java and have presented our proposed modification to the Java Virtual Machine that includes type checking support for parametric polymorphism, exact types, `ThisType`, and `ThisClass`. Polymorphic code without stupid type casts is passed by our enhanced verifier and the extra method call associated with bridge methods has been bypassed.

This chapter includes a summary of the design, a brief analysis of its success, and concludes with a list of future work that needs to be done on LOOJ and LOOJVM. The current state of implementation can be found in the appendices.

6.1 LOOJVM

The design of the LOOJVM affects several major parts of the LOOJ language. The first of these is the compiler. As described in Chapter 4, the LOOJ compiler was modified to provide further attribute information necessary for bytecode verification. Moreover, the LOOJ compiler was modified to not add the extra type casts usually inserted during erasure.

The second area affected is the class loader and the `Class` data structure in the VM. The class loader was modified to recognize the new LOOJ attribute information and the `Class` structure was modified to be able to hold this information, as well as information for specific instantiations of a polymorphic type. These modifications were described in section 5.4.

The next area affected was the run-time class hierarchy and class pool. These were built for type checking by a preprocessing algorithm described in subsection 5.5.3. The preprocessor also optimized bridge method calls as described in subsection 5.5.2.

The final area affected was the bytecode type checker. It was presented in Chapter 3 and the new rules need for LOOJ type checking and their algorithmic implementations in the verifier were described in section 5.3.

In summary, our design maintains static type information from the source code through link-time, instead of just through compilation, allowing our verifier to pass bytecode produced by the LOOJ compiler as if it were never erased to standard Java.

6.2 Success?

To evaluate the success of our project, let us restate our design goals.

- **Conceptually Simple**: It should be easy to make the specified modifications to existing JVMs. Companies and open source groups have invested too much time and money into JVMs to start from scratch.
- **Easy Just-In-Time Compilation**: Any JVM modification should fit into current JIT compilation techniques with little modification to existing JIT compilers. Even a great interpreter cannot outperform a mediocre JIT compiler, so any JVM enhancement that rules out easy JIT compilation is unreasonable.
- **Bytecode Compatibility**: All existing Java class files should run correctly on the enhanced JVM according to the current JVM specification.
- **Efficient Legacy Code**: All existing Java bytecode should run efficiently on the enhanced JVM. Efficiency means that it should run just as fast (or very close to as fast) as on an unmodified JVM.
- **Efficient Parameterized Code**: Code using new language features should also run efficiently. In this case, efficiency means that it should run *faster* than code containing the casts used in current "generic" data structures.

The first two goals have been realized. By restricting our modifications to the compiler, class loader, and bytecode verifier we have not touched the most complicated sections in a JVM. The current implementation of LOOJVM, which is based on the Kaffe[1] virtual machine, is only a couple

[1]Kaffe is a trademark of Transvirtual Technologies, but is published under the GPL.

thousand lines of code that is in source files separate from the rest of the virtual machine.

Bytecode compatibility has also been maintained. In fact, a class that does not have LOOJ attribute information is loaded and verified as if the language extensions were not even present. This also means that legacy code is run just as efficiently.

We are currently unsure about the success of the last goal. Although stupid type casts are no longer necessary due to our enhanced verifier and bridge method calls no longer require a separate method invocation, there does not currently exist a standard set of programs from which to benchmark code programmed with F-bounded parametric polymorphism, let alone code using other LOOJ type extensions. Moreover, due to the almost heterogeneous nature of the preprocessing algorithm and the LOOJ type hierarchy, there is certainly an increased loading time above that required by standard LOOJ (i.e. LOOJ run on a standard VM, complete with stupid type casts and bridge methods). Finally, LOOJ language features, such as the ability to cast to polymorphic types, still require the run-time `PolyClass` instance variables and complicated instruction expansion to work properly, as LOOJ type information is now forgotten after link time.

Therefore, in code that heavily utilizes generic data structures, our VM probably runs polymorphic code faster than LOOJ code is run on a standard JVM, but in degenerate cases with tons of different instantiations of polymorphic types that are each used only once, the excess preprocessing overhead associated with preparing each of those instantiations probably hurts performance.

6.3 Future Work

There is still a lot of work to be done in the LOOJ project. This work falls under two categories. The first is theoretical language design and the second is tools and implementation.

6.3.1 The LOOJ Specification

The LOOJ type system itself needs some refinement. In particular, it would be nice to be able to use the `implements` keyword to bound type parameters. Furthermore, it is debatable whether objects of type `ThisClass` should have access to private variables of a class because `ThisClass` really represents any extension of the current class.

The `PolyClass` model of run-time first class parametric polymorphism could be more efficient. The LM project proposed a `PolyClassManager` that would act as a class pool for polymorphic types. Such a manager could be added to the class library supported by the virtual machine to cut down on the number of copies of a particular polymorphic instantiation in

memory. Moreover, such an optimization could decrease the relative speed of an enhanced JVM and would therefore be useful in benchmark comparisons.

6.3.2 Tools and Implementation

There are a number of interesting tools and implementation problems that still need to be done. To begin with, we would like to produce a production level compiler. The current LOOJ compiler has been undergoing constant debugging since the beginning of the project and should be completed. Ideally, our compiler would be able to output either code targeting a standard JVM or code targeting the LOOJVM.

The LOOJVM itself needs an overhaul, which it will probably receive this summer. At the moment some of the type checking code surrounding the use of `ThisClass` and exact types is pretty buggy, and the organization of methods could be cleaner. Furthermore, the type checking of polymorphic methods is currently not supported at all. Like many large projects, the first time through can be really messy, but many lessons are learned and design flaws uncovered that allow for a much smoother second implementation.

As mentioned in the last section, we do not know how efficient or inefficient our current design is. The creation of a set of standard library of test classes for benchmarking would be invaluable.

Furthermore, there is currently no way to produce degenerate cases to test the LOOJVM's verifier. A modified Java Assembler that accepts LOOJ bytecode syntax and produces a class file with LOOJ attributes would be really useful, as it could be used to produce code that should fail verification. At the moment, the only code the LOOJVM tests is code produced by the LOOJ compiler, which is (hopefully) type safe code to begin with.

One potentially interesting extension to this project would be the development of a smarter preprocessor that could remove stupid type casts from a class file before it even enters the JVM. One drawback to the approach taken by this thesis is that polymorphic code produced by the LOOJ compiler can no longer be run on a standard JVM due to its lack of type casts. It should be possible to have the compiler output code that contains the dumb casts but remove them right before execution for smart virtual machines.

Finally, it would be nice to have a LOOJ Emacs mode. After all, GJ has one.

6.4 Conclusion

We have presented a rather simple modification to the Java Virtual Machine that allows the bytecode verifier to pass code using F-bounded parametric polymorphism, exact types, `ThisType`, and `ThisClass`, but lacking the stupid type casts usually inserted during erasure. Our modified VM also includes an optimization for bridge method calls. We believe that code that

makes considerable usage of generic data structures will certainly run faster without the excess type casts associated with standard erasure.

Designing and implementing the LOOJVM has been challenging and is an ongoing process. Much as the LOOJ compiler has evolved since the publication of Foster's thesis two years ago, the LOOJVM and its specification will likely be refined and improved in the future, but it is off to a very promising start.

Appendix A

Kaffe

The virtual machine implementation modified to create LOOJVM is the Kaffe[1] virtual machine. It is an open source virtual machine published under the GPL and is freely available at:

`http://www.kaffe.org/`

Because of the nature of the GPL, LOOJVM is also published under that license.

This appendix explains why we chose Kaffe and explains some implementation details provided to give a boost to anyone who desires a peek at the source code.

A.1 Why Kaffe?

Several other virtual machine implementations were considered for modification in this project. We decided that an open source route would be best, as it would make publishing the resulting source code hassle-free. The two major open source implementations considered were IBM's JikesRVM and Kaffe. The JikesRVM had the advantage of being implemented entirely in Java (with only a couple hundred lines of C code that is used to bootstrap the VM when it initially loads). Kaffe is written mostly in C. Furthermore, the JikesRVM is a more complete and much more efficient JVM implementation.

However, we found that the developers on the Kaffe developer's list serve[2] provided more consistent and friendly help when we were having trouble

[1] Kaffe is a trademark of Transvirtual Technologies

[2] kaffe@kaffe.org.

deciphering code buried within the virtual machine. The source itself is extremely well documented and organized, and the structure and organization of the project was relatively easy to pick up, despite its size.

A.2 Regarding the Implementation

This section is provided to help someone figure out how the implementation is organized. All modifications made to the virtual machine exist in one giant "loojvm" patch.

When the source tarball is downloaded, the directory ./kaffe/kaffevm contains the source code for much of the virtual machine.

The important files that were modified include:

- debug.h - LOOJ debugging comments added (when the loojvm is run with -the debug LOOJ option, it produces an amazingly large amount of useful output from the class loader and verifier).
- constants.h - describes the constant pool.
- readClass.c - the class file reader.
- classMethod.c and classMethod.h - files containing the class data structures.
- baseClass.c, baseClass.h, itypes.c, itypes.h - contain primitive types, array types, and other things needed for verification.

In addition to the above classes, the verifier subsystem, which was nonexistent at the start of the project, includes the following classes:

- verify.h - common header information for all the verifier's code files.
- verify2.c - pass 2 of verification.
- looj.h and looj.c - the preprocessing algorithm and LOOJ type checking methods.
- verify_blocks.c - methods that verify individual basic blocks.
- verify3.c - the pass 3 data flow analysis algorithm, which uses verify_blocks, complete with type checking code.

There is no guarantee that the organization of the files is going to remain in this form, so the best way to find out what is going on in the most recent implementation is to contact the current maintainer, who, at the time of writing, is Robert Gonzalez.

Bibliography

[AFM97] 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.

[BKR99] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 129–140, 1999.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ: Extending the Java programming language with type parameters. In *OOPSLA 98*, Vancouver, October 1998.

[BPF97] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good "Match" for object-oriented languages. In *ECOOP Proceedings, LNCS 1241*, pages 104–127. Springer-Verlag, 1997.

[Bur98] Jon Burstein. Rupiah: An extension to Java supporting match-bounded parametric polymorphism, ThisType, and exact typing. Williams College, 1998. Undergraduate Thesis in Computer Science.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[CS98] 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.

[Fos01] John N. Foster. Rupiah: Towards and expressive type system for Java. Williams College, May 2001. Undergraduate Thesis in Computer Science.

[KS01] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET common language runtime. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, 2001.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification: Second Edition*. Addison-Wesley, 2 edition, 1999.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, New York, NY, 1997.

[ORW98] Martin Odersky, Enno Runne, and Philip Wadler. Two ways to bake your pizza - translating parameterised types into Java. In *Generic Programming*, pages 114–132, 1998.

[VN00] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.