McGill University
School of Computer Science
Sable Research Group

Soot and Java 1.5

Sable Technical Note No. 4

Jennifer Lhoták (Shaw)

May 17, 2005

Contents

List of Figures

Abstract

Recently Sun released an updated version of the Java programming language, Java 1.5. This update includes many new features at the source code and bytecode levels. Soot is a bytecode analysis framework which has many intra- and inter-procedural analyses and optimizations. As Java is changing, so most Soot. We want to be able to take advantage of the new Java language features to further enhance the Soot toolkit. The purpose of this work is to update the Soot framework to handle the new features in Java 1.5 at the bytecode level. In this report we give an overview of the new features at the source and bytecode levels. We explain how we modify Soot to handle the bytecode level changes and the challenges we face while doing this. This type of project benefits from a great deal of testing and we present our testing strategy and testing frameworks used. Finally, we present some experimental data and conclude with the many new areas of future work.

1 Introduction and Motivation

Soot[20] is a bytecode analysis and optimization framework developed by the Sable Research Group[9] at McGill University. It currently handles class and source file inputs corresponding to code created up to and including Java 1.4. Recently, Sun[16] released a specification for extending the Java programming language with new features. These new features include generics, enhanced for loops, auto-boxing and unboxing, enumerations, variable length argument lists, static imports and annotations. Some of these new features, such as the enhanced for loop, introduce source level only modifications and some, such as annotations, also introduce changes at the bytecode level. There are also other minor, bytecode level changes such as a simplified class literal and extra debugging attributes. The main goal of this project is to update the Soot framework to support the changes introduced by Java 1.5 at the bytecode level.

2 Background and Related Work

Soot, as mentioned, is a bytecode analysis and optimization framework. One of its main contributions is that it is a common framework in which researchers can experiment with analyses and compare the results in a common system. Consequently, Soot is used by many researchers at many institutions [19] who are experimenting with Java analyses. As Java has been extended with several useful and interesting features people are quite anxious to start working with this new version. We already have, at least, three groups of researchers interested in being able to use the results of this project.

Many features in Java are encoded as attributes at the bytecode level. These attributes encode information about the class, extra information about fields and methods and even the code itself is encoded in an attribute. In Soot, while the code is represented at a higher level, most other information is encoded in attributes and propagated throughout Soot in this format. Originally, this attribute framework in Soot was developed for adding extra attributes to the class files for optimization purposes[15]. Since then its been extended to handle more information and this is the basis for handling most of the new Java 1.5 features within Soot at the bytecode input level.

As Sun has just released the new Java features, we refer to The class File Format proposed specification [13] which Sun has made available. This specification details all changes at the bytecode level and specifies the exact file format.

Sun has also made available a bit of documentation about the new features of the Java language from the source level perspective [11]. This documentation is useful when we are developing tests and benchmarks that exhibits all of the new features. Although, whenever possible we use existing test suites.

The biggest change at the bytecode level in Java 1.5 is the annotation facility. The main source of reference is the annotations specification [7]. There are also many small tutorial type articles [12][18][17] [6][10] published on the Internet and these are to gain an understanding when working with the test programs and the test suites as each one provides practical tips and examples about how to use this facility.

2.1 Overview of Java 1.5 Features at the Source Level

There are seven new languages features introduced at the source code level. Here we will discuss each one and give simple examples.

An enhanced for loop simplifies iterating over collections and arrays. In the following example we show the old for loop style in a small method which adds up the elements in an array of integers. In the for loop we need to list the initializers, get the length of the array and specify the condition and give the iterators. We then explicitly perform the array access to retrieve the needed element.

```
int sum( int [] a ){
    int result = 0;
    for ( int i = 0; i < a.length; i++ ){
        result += a[i];
    }
    return result;
}
```

In contrast, the next example shows the enhanced for loop where we no longer need to find the length of the array or give the iterators. Additionally, the array element is retrieved automatically.

```
int sum( int [] a ){
    int result = 0;
    for  ( int i : a ){
        result += i;
    }
    return result;
}
```

Autoboxing and unboxing automate the conversion between primitives and their corresponding wrapper classes. For example, if we want to add a variable of type int to a list we must first wrap it in an Integer object.

```
void addAge(List list, int age){
    list.add( new Integer(age) );
}
```

In Java 1.5, however, we can simply add the int variable to the list directly and the compiler will look after adding the necessary wrapping.

```
void addAge(List list, int age){
    list.add( age );
}
```

Static imports allow unqualified access to static members in a class. Consider the Math class with all the static fields and methods defined for mathematical calculations, at the use of each element, the access to the static member must be qualified with the Math class.

```
double computeArea( int radius ){
   return Math.PI * radius * radius;
}
```

With static imports we can import the Math class' static members and then use them directly without qualification.

```
import static java.lang.Math.PI;
...
double computeArea( int radius ){
   return PI * radius * radius;
}
```

VarArgs allow methods with variable length parameter lists. Prior to Java 1.5, the only way to have methods with variable length parameter lists was to specify an array or collection as the parameter and package up the argument elements as seen in this example.

```
void compute(){
    int sum = sum( new int [] {2, 3, 4, 5, 6} );
}
int sum( int [] a ){
    int result = 0;
    for ( int i : a ){
        result += i;
    }
    return result;
}
```

Now we can specify the parameter with ... to indicate a variable number of elements will be passed to the method and can send in the elements directly.

```
void compute(){
    int sum = sum( 2, 3, 4, 5, 6 );
}
int sum( int ... a ){
    int result = 0;
    for ( int i : a ){
        result += i;
    }
    return result;
}
```

Generics allow compile time type safety for collections. They are one of the most powerful new features at the source level as they not only provide type safety but eliminate the need for many checks and casts. Consider the following example where we iterate through our ArrayList getting the next element. All we know for sure is that the element is an Object. We must check that the element is a String and then cast it to a String before we can use it.

```
void output( ArrayList list ){
    Iterator it = list.iterator();
    while ( it.hasNext() ){
        Object next = it.next();
        if ( next instanceof String ){
            if ( ( ( String )next ).charAt( 0 ) == 'j' ){
                System.out.println( next );
            }
        }
    }
}
```

In Java 1.5 we can declare the ArrayList to be a list of Strings. All elements added to the list will be checked at compile time to ensure they are Strings so the extra checks and casts are no longer necessary.

```
void output( ArrayList<String> list ){
    for (String s : list){
        if ( s.charAt( 0 ) == 'j' ){
            System.out.println( s );
        }
    }
}
```

Enums allow type safe enumerations with namespaces. They are another very powerful new language feature. The basic problem before Java 1.5 was that there were no enumerations. The best one could do was to declare a set of final static ints, possibly in a constant interface and name each field with a prefix indicating the constant kind. We see in the example that we must use a prefix to distinguish between the seasons and terms. When we call the print method to print out a season we will only get an int and indeed there is nothing to prevent us from using this print method to print out any int that may not even be a season.

```
public final static int SEASON_WINTER = 1;
public final static int SEASON_FALL = 2;

public final static int TERM_WINTER = 3;
public final static int TERM_FALL = 4;

void print( int season ){
    System.out.println( season );
}
```

Now one can specify an enumeration directly. An enumeration can be as simple as a list of constants, or each constant can have secondary values associated with it. Constructors and methods which operate on the enumeration type can be specified and certain constants can override those methods for specific functionality if desired. Additionally, when we now call the print method, in out example, we must send an argument of type Season and it will print a textual representation of the constant.

```
enum Season {
   WINTER(-15), FALL(5);
   private final int avgTemp;
   Season(int avg){
       this.avgTemp = avg;
   }
}
enum Term {WINTER, FALL}

void print( Season season){
    System.out.println( season );
}
```

Finally, the last major change at the source level is the addition of an annotation framework. Annotations allow extra information to be associated with program elements but do not change the program semantics in any way. Annotations can be added to types, including enumerations and annotations, constructors, methods and fields, method parameters and local variables. Annotations may be one of three kinds: a marker annotation, a single-element annotation or a regular annotation. A marker annotation is one which takes no values. This is possible if the annotation is declared with no elements or if all of the declared elements have default values. In the example @Prototype is a an example of a marker element. A single-element annotation takes a single un-named value and must be declared with either a single element named value or a single element named value and other elements all with default values. An example is the @Copyright annotation. A regular annotation takes one or more named values as arguments and is declared with one or more elements with or without defaults. An example is the @Author annotation. There are three levels of annotations; source, runtime invisible and runtime visible. Annotations on local variables are source level only as there is no place to store them in the classfile. Annotations on all other program elements can be specified to be at any level.

```
@Prototype public class MyAspectJCompiler {
    @Copyright( "Jennifer" )
    public void generateCode(){
        @PotentialDeclType( List.class ) ArrayList list;
    }
    enum Season { WINTER, SUMMER };
    public void favoriteSport( @TimeOfYear( Season.WINTER )
        String sport){
    }
    @Author(name="Jennifer",age=27) Parser parser;
}
```

Annotation types are specified in the same fashion as interfaces. Each element is specified as a method with the method name corresponding to the element name. Each element may have a default value associated with it as seen in the PotentialDeclType example.

```
@interface Copyright {
    String value();
}

@interface PotentialDeclType {
    Class value() default Object.class;
}

@interface Author {
    String name();
    int age();
}
```

2.2 Overview of Java 1.5 Features at the Bytecode Level

In this section we discuss the bytecode level changes introduced by Java 1.5. First we will discuss the bytecode changes in relation to the seven new source level features, and then the other changes.

The enhanced for loop, autoboxing and unboxing, static imports and varargs are compiled into regular Java bytecode. For example, when we put an int directly in a collection, the compiler takes care of the wrapping for you. In other words, these are simply source level changes designed to make the source code more readable, more maintainable and more elegant. The standard class library has been optimized to handle these features well. The bytecode for the enhanced for loop leaves out any instanceof checks, as they are no longer necessary. The bytecode for autoboxing uses a common object for frequently boxed values. These changes do not need to be handled when processing the bytecode by Soot.

Generics are also compiled into regular bytecode with an attached **signature attribute**. A **signature attribute** contains a reference to a constant pool entry which contains a string of the source level signature of the element. For example, the following method

```
void output(ArrayList<String> list){...}
```

declared with a list of strings as the parameter, will have a signature attribute with the string

```
"(Ljava/util/ArrayList<Ljava/lang/String;>;)V"
```

Enums are considered to be types just like classes and interfaces and are compiled into regular classes. If the enumeration is declared within a class, it is compiled into a nested class of its enclosing class, as we should expect. Each enumeration has an enum modifier to indicate it is an enumeration. The constants are represented as fields, with each also having an enum modifier. Any methods or constructors declared are of course compiled into regular bytecode and if one of the constants has a body associated with it, for specialized functionality, the body is compiled into an anonymous nested class. Other than the enum modifiers on the type and fields, there are no bytecode level changes associated with enumerations that require any handling in Soot.

Annotations are represented in the bytecode as one of four visibility attributes: a **runtime visible annotations attribute**, a **runtime invisible annotations attribute**, a **runtime visible parameter annotations attribute**, or a **runtime invisible parameter annotations attribute**, depending on the level of the annotation and where it is declared. The **parameter annotations attributes** are for annotations declared on method parameters and the others are for annotations declared elsewhere. The visibility indicates whether the annotations should be available at runtime or not. Annotations of both levels of visibility are, of course, represented in the classfile and must

therefore be processed by Soot. Each program element has at most one **runtime visible annotation attribute** and at most one **runtime invisible annotation attribute**. A method may additionally have at most one **runtime visible parameter annotation attribute** and at most one **runtime invisible parameter annotation attribute**.
Each attribute may contain zero to many annotations. These annotations have a type and a number of named elements. For example,

```
@Prototype public class MyAspectJCompiler { ... }
```

is stored by default in a **runtime invisible annotations attribute** containing a single annotation with type

```
LPrototype;
```

and no elements. Each element has a name and a kind flag to indicate the kind and to indicate how many, and of what type of values it has.

```
@Sample(
    age=27,
    pi=3.14,
    salary=4.87,
    name="Jennifer",
    season=Season.SUMMER,
    myClass=String.class,
    people={"Ondrej","Lhotak"},
    note=@Copyright(level=@Prototype,author="Jennifer"))
```

The previous example indicates the kinds of elements. Each is stored in bytecode as follows:

```
"age" --> I --> 27
```

Int elements (and chars - C, booleans - Z, shorts - S and bytes - B) are stored as the element name, the kind indicator, and an integer.

```
"pi" --> F --> 3.14f
```

Float elements are stored as the element name, the kind indicator, and a float value (of course in actual bytecode the float is stored as raw int bits).
Long - J, elements are stored as the element name, the kind indicator, and a long value.

```
"salary" --> D --> 4.87
```

Double elements are stored as the element name, the kind indicator, and a double value (of course in actual bytecode the double is stored as raw int bits). All of the primitive types use the base type characters that should be familiar for representing the types of primitives. Other characters have been added to represent the other annotation element value kinds.

```
"name" --> s --> "Jennifer"
```

String elements are stored as the element name, the kind indicator s, and a string value.

```
"season" --> e --> "LSeason;", "SUMMER"
```

Enum elements are stored as the element name, the kind indicator e, and a string representing the class and a string representing the enum constant.

```
"myClass" --> c --> "Ljava/lang/String;"
```

Class elements are stored as the element name, the kind indicator c, and a string representing the class.

```
"people" --> [ --> s --> "Ondrej", s --> "Lhotak"
```

Array elements are stored as the element name, the kind indicator [, and the array values. These array values store the kind and value but no names. Array elements can be nested.

```
"note" --> @ --> "LCopyright;" --> "level" --> @ --> LPrototype;,
    "author" --> s --> "Jennifer"
```

Annotation elements are stored as the element name, the kind indicator @, and the annotation type and elements. These annotation elements store the kind and value **and the names**.

These elements are all quite straight-forward, except elements of kind array and annotation. The elements of kind array contain an array of un-named elements, which makes reusing the code to handle elements a bit tricky because essentially array values are just like elements. They can have up to approximately 65000 elements. An annotation element contains a complete annotation. These can be nested up to some large constant (about 65000) number of times. Fortunately, nested annotations have names. **Runtime visible and invisible parameter annotations** contain for each parameter in the method at most one set of runtime visible or invisible annotations. Each set contains zero to many annotations, which each contain zero to many elements as before.

```
@interface PotentialDeclType {
    Class value() default Object.class;
}
```

When annotations are declared they may have an associated default value. To store this value, there is an **annotation default attribute** which is attached to a method declaring the annotation element. It takes a value corresponding to the element kind associated with the method. The method `value` representing the annotation element will have a **annotation default attribute** with a string value

```
c --> "Ljava/lang/Object;"
```

These are represented as regular annotation values as described above.

Annotations are declared very much like interfaces and are considered to be regular types. An annotation modifier is used to indicate that the type is an annotation.

There are several other minor bytecode level changes which do not correspond to any source level changes. We will discuss these here.

An **enclosing method attribute** is attached to an anonymous or local nested class. This is an attribute for debugging purposes and indicates the class and method where this class was declared in the original source. For example the anonymous nested class

```
public class Outer {
    public void myMethod() {
        new Inner(){
            public int myInnerMethod() { ... }
```

```
            }
        }
    }
```

will have an **enclosing method attribute** with two strings

```
"LOuter;", "myMethod()V"
```

the class and method descriptors.

The **local variable type table attribute** stores signature information for local variables of generic reference types, in a code attribute table on each method. It is quite similar to the local variable table which stores for each local variable the name and a descriptor representing the type. For variables of a generic reference type there is also an entry in the local variable type table which stores the name and a signature representing the type. For example the following local variable

```
ArrayList<String> myList
```

will have an entry in the local variable table of

```
ArrayList<String> myList --> "Ljava/util/ArrayList;<L/java/lang/String;>"
```

The last change at the bytecode level involves class literals, for example in the source level we have:

```
String.class.getName();
```

The class Class has been generified and class literals are now represented as a class constant. Previously, several lines of bytecode were needed to represent class literals.

```
getstatic     ClassTestsSimple.class$java$lang$String Ljava/lang/Class; (8)
ifnonnull     #21
ldc           "java.lang.String" (9)
invokestatic  ClassTestsSimple.class$ (Ljava/lang/String;)Ljava/lang/Class; (10)
dup
putstatic     ClassTestsSimple.class$java$lang$String Ljava/lang/Class; (8)
goto          #24
getstatic     ClassTestsSimple.class$java$lang$String Ljava/lang/Class; (8)
invokevirtual java.lang.Class.getName ()Ljava/lang/String; (11)
```

The ldc_w bytecode, a pre-existing bytecode is now used for the representing the class literal as a constant.

```
ldc_w java.lang.String (9)
```

3 Specific Problem Statement

The purpose of this project is to update the Soot bytecode analysis framework, to accept as input class files with Java 1.5 features, to process these new features throughout Soot and to output class files with these features. This involves handling all new attributes, updating all the parts of Soot from Coffi to Jasmin, and extensive testing.

4 Solution Strategy and Implementation

In this section we discuss the implementation details in a step by step manner shown in Figure 1. Each box represents a phase of the implementation and we discuss each one in turn.

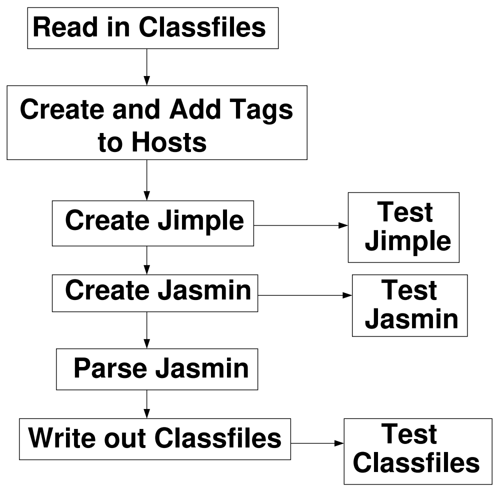

Figure 1: Implementation Overview

4.1 Read in Classfiles

The first step is to read in all the new attributes from the bytecode. The basic idea is to understand the specification and determine how many bytes should be read in for each element and then decode the constant pool entries. Usually, a reference to a constant pool entry is a short, a kind for annotation element is a single byte and we must remember to not try and read in a short that would refer to a name in the constant pool when reading in array element values. The confusion here arises due to the array element values being very similar to regular named annotation values and of course we want to reuse the code to read these in. We succeed at this code reuse with a simple flag to indicate an array value versus a regular element, allowing us to process them in the same way, here and throughout the rest of the project. Additionally at this stage we must accept a new kind of constant, the ClassConstant and handle it in Jimple. Similarly as the runtime visibility is the only difference between the pairs of annotations attributes we simply use a constant to indicate the visibility level and reuse the same code for everything else.

4.2 Create and Add Tags to Hosts

The second step is to create a tag for each attribute that can store the minimal amount of information needed to be able to send the attribute through Soot and write it out again. We attach these tags to their corresponding Soot hosts. The **signature attribute**, **runtime visible annotations attribute** and **runtime invisible annotations attribute** can be attached to SootClasses, SootFields and Soot Methods. The **runtime visible parameter annotations attribute**, the **runtime invisible parameter annotations attribute**, the **annotations default attribute** can be attached to SootMethods. The **enclosing method attribute** is attached to SootClasses.

4.3 Create Jimple

The next step is, of course, to update the creation of Jimple. The only change here is to create a new Soot constant type ClassConstant. We then need to update the type assigner to realize that a local that is a ClassConstant is of type Class.

4.4 Test Jimple

At this point, fortunately, Soot allows us to output a textual form of Jimple. In this textual representation we can also print out all of the tags that have been attached to all of the hosts. This is very useful as it allows us to verify that up until this point we have correctly read in all of the attributes. Usually, if we read in something incorrectly a class cast exception will be thrown, but occasionally things slip through and if we get a constant pool entry of the right type, by coincidence, we may not catch the error until this stage.

4.5 Create Jasmin

The next step is to produce Jasmin. By default, Soot produces Jasmin by converting Jimple to Baf and then converting Baf to Jasmin. There is also an option for converting Jimple to Grimp and then coverting Grimp to Jasmin. Hence, we must handle the new ClassConstant in both of these cases, even though we rarely produce Jasmin from Grimp. The conversion from Jimple to Baf and Jimple to Grimp for the ClassConstant was handled automatically, no special processing was required. To clarify this point, the creation of the ClassConstant class requires implementing the conversion methods needed, however they are too simple to need discussion. In both cases for producing Jasmin, it is very straight forward we simply emit a lcd_w instruction with the ClassConstant string which is a string representation of the required class.

The challenge in producing Jasmin lies in inventing the syntax for representing all of the new attributes in a simple, brief and non-ambiguous way. Most attributes are quite simple in this regard, for example, the signature and annotation default attributes have only a single value and the enclosing method attribute has only two values, which are easy to specify. These we specify as directives followed by the values. The annotation attributes are the difficult ones as they can be nested inside of each other and may have many elements of different kinds.

4.6 Test Jasmin

Once Jasmin is produced inside Soot, Soot is able to write it out in a textual format much like it writes out Jimple. Again, at this point, we stop and verify that the textual Jasmin is correct and specifies all of the attribute information needed.

4.7 Parse Jasmin

The most difficult part of this project is updating the Jasmin assembler to parse and accept the new Jasmin syntax created in the previous step. Jasmin was written several years ago when the Java language was much simpler. Since that time it has been updated in an ad-hoc way to support new attributes as required. Unfortunately, it is quite inconsistent. In particular, the grammar for methods is declared in an imperative style, the method is created and then all the parts such as the code attribute or the exception table are added later. The grammar for fields, on the other hand, is declared in a functional manner, where all the parts are created and then finally the field is created. This is quite a problem when we want to add the grammar rules that handle signatures attributes, for example, to both fields and methods as we must handle the attribute in two different ways. At this time we simply accept the limitations of Jasmin and specify everything necessary in order to have a functional implementation. As a future work project we plan to fix this situation.

4.8 Write out Classfiles

The backend of Jasmin, on the other hand, the Jas library is fairly well written and easy to use and we have no difficulties in building up the constant pool and writing out the classfiles. The most difficult part is determining the sizes of each attribute. At this stage we also handle the ClassConstant throughout Jasmin and Jas, which involves handling a new constant pool constant entry type.

4.9 Test Classfiles

Finally, we must test the Soot produced classfiles. First, we try and run the tests using the Java 1.5 VM. This serves two purposes: it ensures that the bytecode is verifiable and it ensures that the semantics of the program have been kept in tack. If the program will not verify in the VM, then we have to find some way to hunt down the problem. This can occur, for example, if the classfile is truncated or if there is an invalid type in the constant pool. To attempt to track the down the problem we use BCEL[1], which lists the classfiles in great detail. We can then compare the original bytecode with the Soot produced bytecode to attempt to find the problem. Sometimes even BCEL doesn't work and as a last resort we re-process the Soot produced classfile through Soot and usually Soot throws an exception at this point and indicates where the problem is. It isn't a fool proof method though an sometimes we need to reproduce Jimple or even reproduce Jasmin and use diff[4] on the files to find the problem.

5 Experimental TestSuite

A project like this requires a great deal of testing to ensure correctness. As there are a very large number of different combinations of test cases to consider it is unreasonable to attempt to write all the tests. Our approach here is to instead find large open source test suites from other projects which are also updating their tools. We are very fortunate to find a large test suite from the Eclipse[5] JDT project. The JDT project has a set of compiler regression tests in which they encode the tests in huge regression test files and run using a harness. For our purposes we want to be able to easily run the tests through javac and Soot so we write a little extractor which extracts each test into its own file. This test suite contains about 600 positive tests, ones that should compile and another 600 negative ones that should not compile. These tests cover all the major new language feature areas. In this part of the project we only look at the positive tests as we must have bytecode to feed into Soot. The negative tests will, however, be useful when we update Soot to handle Java 1.5 source input. Of these tests, the majority pass. There is one annotation test which gives incorrect results at runtime which needs to be sorted out. There are also several generics tests where there is some confusion between Eclipse and Sun over whether they should compile or not and the ones that do not compile with javac have not been tested with Soot. Overall 98% of the 600 positive tests are passing. The second smaller test suite that we use is from the Technische Fachhochschule Berlin[2]. This suite contains 50 positive tests in all areas except, unfortunately annotations. Extra tests are always useful though and all of these tests pass.

6 Results

The experiments for this project look at two areas: the running time and size of Soot produced classfiles. When comparing running times of javac produced classfiles and Soot produced classfiles we see that the times are quite similar in the majority of the cases, however there are some definite areas where Soot could be improved, for increased performance. When looking at the sizes we see that Soot always makes larger classfiles. We give a detailed discussion of these results below.

6.1 Experimental testbed or environment

All experiments are conducted on a Intel(R) Pentium(R) 4 CPU 2.00GHz. We use javac version 1.5.0_01 to produce all of the original classfiles that we wish to process through Soot. We modify Soot and Jasmin revision 2008 with all of our bytecode level changes. We then use java version 1.5.0_01-b08 JavaHotSpot(TM) Client VM (mixed mode, sharing), to run our Soot produced classfiles.

6.2 Benchmark programs

For our experiments, as we have not yet found any real Java application benchmarks that extensively use the new language features, we use a set of artificially lengthened test suite programs that cover the range of new language features. The AnnotEx1, AnnotEx2 and AnnotEx3 tests cover annotations with all the different element kinds, declared on method parameters, methods, fields and types including annotations declared on enumerations and annotations. The AutoBoxEx1 test looks at boxing and unboxing between primitives and their associated wrapper classes and AutoBoxEx2 tests the conversions between primitives and other wrapper classes or wrapper classes and other primitives. The EnumEx1 tests a complex enumeration with some specialized functionality for some constants and EnumEx2 tests a simple enumeration with common functionality for all constants. The ForEachEx1 test looks at iterating through a list of Strings. The GenericsEx1 test handles a very simple generified class and its use. The GenericsEx2 test is an exercise in creating, extending, implementing and using generified classes and handles the wild card specification system. The StaticEx1 test static imports and uses the static members from the Math class and finally the VarArgEx1 tests deals with calling a method, with a variable length parameter list, with several different length argument lists.

6.3 Detailed Results and their analysis

There are two experimental areas that we must consider in this project: running times and classfile size. We compute these experimental results in the following way. For the running time we run each classfile produced by javac ten times, each classfile produced by Soot is also run ten times, the lowest and highest running times are thrown out and the average is taken from the remaining eight. This average is the measurement used in the graphs for comparison purposes. For the classfile size we compare the size of the classfile produced by javac and the classfile produced by Soot. We only consider one size of each classfile as the results should be deterministic.

We must look at the running times of the classfiles produced by Soot and compare them to the running times of classfiles produced by javac. With only the standard Soot optimizations turned on we expect that the running times would be quite similar. We see in Figure 2, where the benchmarks are listed along the x-axis and the running time is listed on the y-axis in seconds, that sometimes Soot wins and sometimes javac wins and that often the running times are indeed quite similar. There are two benchmarks where we note that the running time of Soot produced classfiles is significantly slower than the running time of javac produced classfiles. To investigate the differences we compare the bytecode produced by javac with the bytecode produced by Soot. In the AutoBoxEx2 test we notice that the bytecode produced by Soot is very different than the bytecode produced by javac. In particular, we notice that the Soot produced bytecode introduces many extra conversion bytecodes (ie: i2b, i2d etc). This may be caused by the way types are inferred in Soot or perhaps javac optimizes in way that expects to handle the new language features. More investigation is needed in this area. In the VarArgsEx1 test we notice that the bytecode produced by Soot has many more stores and loads than the bytecode produced by javac. More investigation is needed to determine how to improve the Soot produced bytecode.

We must also consider the size of the classfiles produced by Soot and compare them to the size of the classfiles produced by javac. There are two goals: one is to determine if the Soot produced classfiles are too large which may indicate we are writing out something incorrectly, the other is to see if the Soot produced classfiles are too small, as this would indicate that we are forgetting to write something out. As we see in Figure 3, where the benchmarks are again listed along the x-axis and the size in bytes is listed on the y-axis, that all Soot produced classfiles are larger than javac produced classfiles. Upon further investigation, we note that Soot produced classfiles have many more line number table entries than javac produced classfiles.

7 Conclusions and Future Work

In this report we describe the new features in Java 1.5 at the source and bytecode level. We show the details about how we update the Soot bytecode analysis framework to process the new Java 1.5 features at the bytecode level. We describe the extensive testing framework and present some experimental results comparing the running time and size of classfiles produced by Soot versus original classfiles produced by javac. Overall, we are quite excited about the new language features in Java 1.5 and look forward to continuing work in this area.

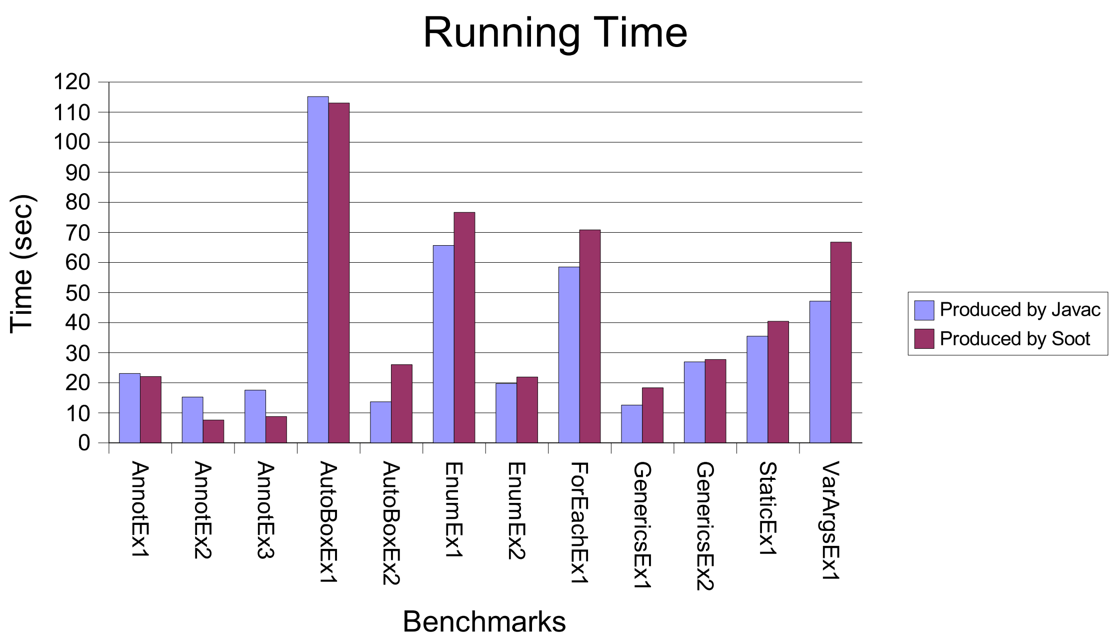

Figure 2: Comparison of Running Times

There are several minor implementation details which still need to be handled. Here I will describe each item in order of priority of importance.

The first item that needs to be addressed is properly calculating the version numbers produced by Jasmin in the new classfiles. When a classfile has a new ClassConstant bytecode, it must have a high enough version for the Java 1.5 VM to recognize the classfile as a new classfile. Otherwise the VM will reject the classfile giving an invalid type in constant pool exception. However, we cannot simply write out a high version number for all classfiles produced by Jasmin because Soot and Jasmin themselves still run with the older VM's which will give an invalid version number exception for classfiles with too high a version number and there is no reason to expect that all users will have switched to Java 1.5. Thus we must be still able to produce classfiles with old version numbers. The solution to this problem is too detect when we have a ClassConstant and output high version numbers only in that instance.

The second area of future work involves correctly updating the Jimple grammar and parser. While there are only minor changes to Jimple involving the ClassConstant, the changes are significant enough that we must update the Jimple grammar to handle this new kind of constant and also there are some new keywords (enum, annotation, varargs) that need to handled. Basically, we need to update the grammar and then run some tests to ensure that all newly modified Jimple can be read back in by Soot.

The third minor area of work involves resolving the overloading of the textual representation of conflicting modifiers. In Java 1.5 the value for the transient modifier, which applies only to fields, is also used for the varargs modifier, which applies only to methods. This is not a problem in the

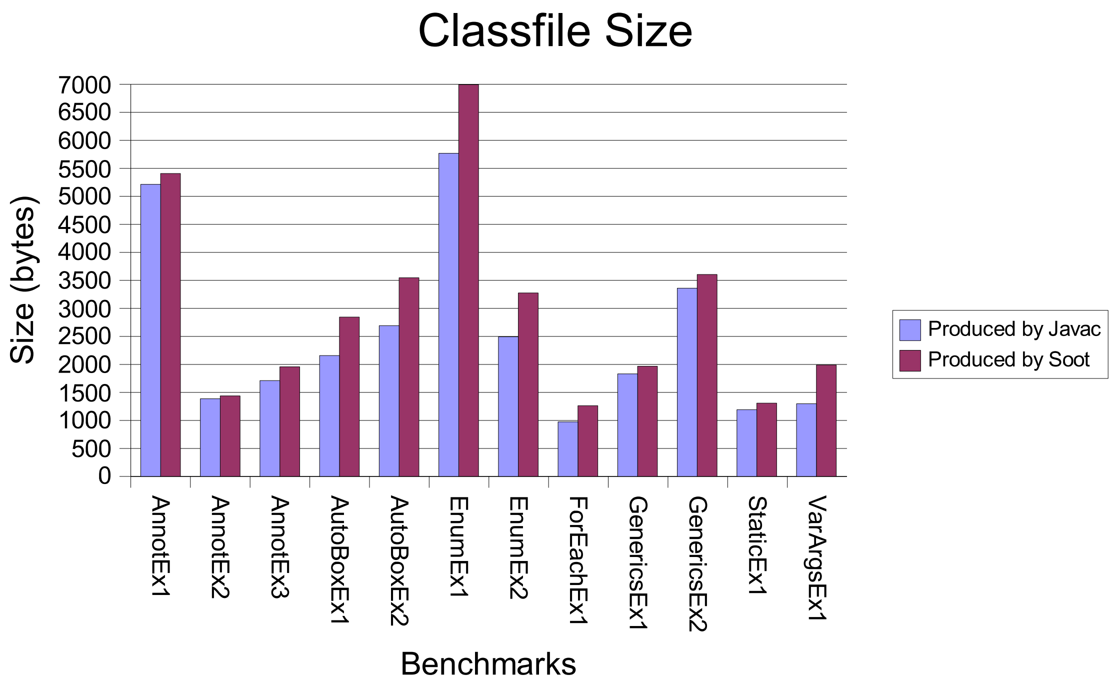

Figure 3: Comparison of File Sizes

semantics of the classfiles produced by Soot but in the textual representations such as Jimple and Jasmin we would like to see the modifier varargs and not transient beside our methods which in the source had variable length parameter lists.

The fourth problem to address is to write out the local variable type table. This table as mentioned earlier is useful for debugging purposes. At this point we have read this table into Soot, but not yet written it out.

There are also three areas of more interesting future work.

As we mention in the detailed implementation section, the Jasmin grammar consists of a mixture of imperative and functional styles. This makes it quite difficult to update and understand. It also includes many features that have associated comments claiming they are for historical purposes. In light of this, the plan is to rewrite the Jasmin grammar and parser to be completely functional. It uses the Java CUP parser generator [3] which lends itself to being functional.

The purpose of this project is to handle the new features of the Java 1.5 language at the bytecode level in Soot. Soot, however, also accepts as input Java source code. The changes to the Java language at the source code level are quite extensive. The plan for handling them involves establishing a CUP style grammar for the language features and building the AST using an extension to the original Polyglot Java 1.4 AST [14]. We will then generate code using an extension to Java2Jimple. A partial CUP style grammar which includes everything except annotations is available from [3]. As well, Sun has released a preview draft of the full new language specification which includes a full textual grammar including rules for annotations [8].

The final future, future work involves looking at the annotations already being generated by Soot analyses and see if they can be formulated in the format of Java 1.5 annotations and written out and potentially used at runtime.

A My Appendix

The source code for this project will be checked into the Soot repository, as soon as the version numbering for Soot produced classfiles is sorted out. At that time I will put the test suites and benchmarks in the repository as well.

References

[1] BCEL. http://jakarta.apache.org/bcel/.

[2] Technische Fachhochschule Berlin. http://www.tfh-berlin.de/ panitz/prog4/skript.html.

[3] Java CUP. http://www.cs.princeton.edu/ appel/modern/java/cup/.

[4] diff. http://unixhelp.ed.ac.uk/cgi/man-cgi?diff.

[5] Eclipse. http://www.eclipse.org.

[6] Program Annotation Facility. http://javaboutique.internet.com/tutorials/annotation/.

[7] A Program Annotation Facility for the Java Programming Language. http://www.jcp.org/aboutjava/communityprocess/review/jsr175/.

[8] James Gosling. *Java Language Specification Version 3*. Addison-Wesley, 2005.

[9] Sable Research Group. http://www.sable.mcgill.ca.

[10] Declarative Programming in Java. http://www.onjava.com/pub/a/onjava/2004/04/21/declarative.html.

[11] Enhancements in JDK 5. http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html.

[12] Annotations in Tiger. http://www-106.ibm.com/developerworks/library/j-annotate1/?ca=dnt-535.

[13] Lindholm and Yellin. The java virtual machine specification. Proposed changes : http://java.sun.com/docs/books/vmspec/2nd-edition/ClassFileFormat.pdf.

[14] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, 2003.

[15] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In *Compiler Construction, 10th International Conference (CC 2001)*, volume 2027 of *LNCS*, pages 334–554, 2001.

[16] Sun. http://java.sun.com.

[17] Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. http://www.onjava.com/pub/a/onjava/2004/06/30/insidebox1.html.

[18] Bridging the Gap: J2SE 5.0 Annotations. http://www.onjava.com/pub/a/onjava/2004/10/06/anno1.html.

[19] Soot Users. http://www.sable.mcgill.ca/ hendren/sootusers/.

[20] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.