JAVA BYTECODE TO PILAR TRANSLATOR

by

VIDIT OCHANI

B.E., Rajiv Gandhi Technical University, 2011

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department Of Computing and Information Sciences
College Of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2014

Approved by:

Major Professor
Robby

Copyright

Abstract

Software technology is the pivot around which all modern industries revolve. It is not surprising that industries of diverse nature such as finance, business, engineering, medicine, defense, etc. have assimilated sophisticated software in every step of functioning. Subsequently, with larger reach of application, software technology has evolved intricately; thereby thwarting the desirable testing of software. Companies are investing millions of dollars in manual and automated testing, however, software bugs continue to persist. It is well known that even a trivial bug can ultimately cost the company millions of dollars. Therefore, we need smarter tools to help eliminate bugs.

Sireum is a research project to develop a software analysis platform that incorporates various tools and techniques. Symbolic execution, model checking, deductive reasoning and control flow graph are few examples of the aforementioned techniques. The Sireum platform is based on previous projects like the Indus static analysis framework, the Bogor model checking framework and the Bandera Java model checker. It uses the Pilar language as intermediate representation. Any language which can be translated to Pilar can be analyzed by Sireum. There exists translator for Spark - a verifiable subset of Ada for building high-integrity systems.

In this report, we are presenting one such translator for Java Bytecode - A frontend which can generate Pilar from Java intermediate representation. The translator emulates the working of the Java Virtual Machine(JVM), by simulating a stack-based virtual machine. It will help us analyze JVM based softwares, such as, mobile applications for Android. We also evaluate and report statistics on the efficiency and speed of translation.

Table of Contents

List of Figures

Acknowledgments

I am grateful to the Department of Computing and Information Science for providing me the opportunity for Masters. I am lucky to have Dr. Robby as my Major Professor. He is one of the most passionate software engineer I have came across. This project would not have been possible without his continuous guidance. He fostered my interest in this area, and has always helped me by taking out time from his busy schedule to advise me. His technical thoroughness are second to none, and his cutting edge research in software analysis is fascinating. I have learned a lot from him, especially the art of compilers.

I would also like to thank my committee members, Dr. John Hatcliff and Dr. Simon Ou, for their insightful comments, support and guidance. Last, but not the least, I thank my family and fellow graduate students in the department for their help and emotional support.

Chapter 1

Introduction

1.1 Motivation

We live in a world which is increasingly dependent on software for most basic and advanced functioning. Software and computing technology has evolved to a point where most industries are dependent on amelioration in technology for materializing innovation. It is, therefore, not presumptuous to state that software technology is shaping the world we live in. Innovation in industries such as medical, education, finance, military, and air travel are few of the many testaments for practical usefulness of software technology. With the ambition to tackle increasingly difficult problems, software technology is getting subsequently intricate with respect to both size and the underlying logic. Increasing dependency of most industries has also increased the demand for more sophisticated and flexible software.

Organizations are struggling to manage the ever increasing complexity of software. Software testing is getting harder. It is becoming increasingly difficult to test software. Software companies are leveraging automation to systematize the tiring process of manual testing. However, the process largely remains inadequate and some bugs may still remain undetected. As a result, we see a surge in demand for tools to help software companies improve testing, thereby ensuring product safety to the end user.

Industries such as aeronautics, automotive, medicine, military and nuclear plants hinge on safety and therefore cannot afford software with lurking bugs. One of the recent examples

of a software crash is the Toyota's killer firmware[8], which caused uninteded acceleration causing death of the occupant. Often, these industries demand zero failure rate of their software under all circumstances. It can result in security vulnerabilities, ultimately causing billions of dollars in damage; so the software employed in the abovementioned industries demand exhaustive testing.

Another important requirement of software engineering is maintenance. In industry, maintenance of the codebase using old technology with ever changing staff is not trivial. The employees working on old systems do not comprehend it completely and often work in an adhoc manner: fixing bugs and adding features. This has resulted in an increase in demand of software analysis tools.

Sireum is a long term research project to develop a software analysis platform. It includes various static analysis techniques like data-flow, control-flow, symbolic-execution, model-checking, and deductive reasoning. The Sireum framework is based on previous extensive efforts in the Indus static analysis framework[17], the Bogor model checking framework[9] and the Bandera model checker[4]. It can be used to build different kinds of static analyzers. It is also very flexible, and can be easily extended to support different languages.

The framework uses an intermediate language representation named Pilar. The tools are generic and work on the intermediate representation(IR). Any language which can be translated into Pilar can be analyzed using the Sireum tools. There exists translator for Spark - a verifiable subset of Ada for building high-integrity systems. In this report, we will present a front-end designed for translating Java bytecode into Pilar. This will help us analyze Java based softwares and libraries using the static analysis tools available in the framework.

1.2 Contribution

The contributions of this report are as follows:

1. A translator for Java bytecode to Pilar.

2. Evaluation of existing Java classes using Sireum tools.

3. A high level discussion on how other languages can be translated.

1.3 Organization

The report is organized in six chapters. Chapter 1 is the introduction that details the problem and description of my project. Chapter 2 is Background that gives details about Java Virtual Machine, Sireum IR(Pilar) and Sireum JVM v1. Chapter 3 discusses about the implementation of my translator and goes into specifics for each bytecode instruction. Chapter 4 presents the testing strategy and some statistics about the speed. Chapter 5 discusses related work that has been done on this topic. The last chapter, Chapter 6, is the future work that can be done on this topic.

Chapter 2

Background

2.1 Java Virtual Machine

Java Virtual Machine(JVM) lies at the heart of Java technology. It is the abstract computer which executes Java programs. JVM can be implemented in software or on hardware directly. One of the main reasons of Java success is it's "write once, run anywhere" model which has been made possible because of the JVM. It is the component responsible for Java's hardware and operating system independence, garbage collection, security and portability.

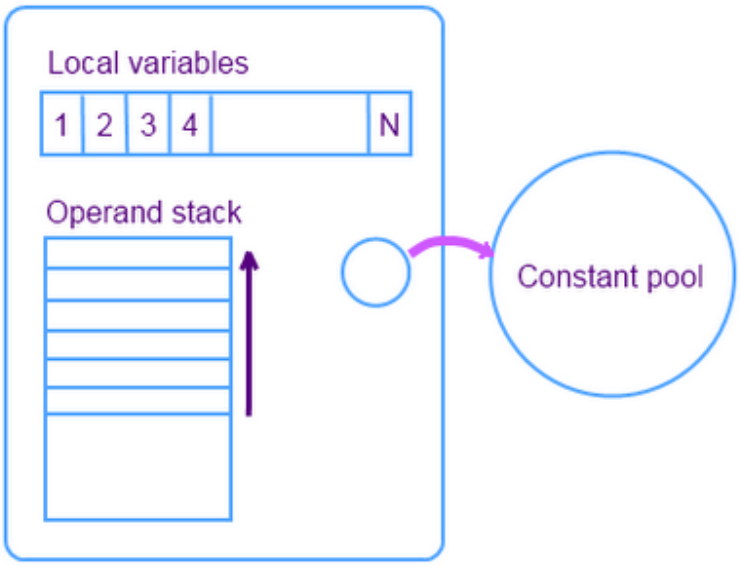

Figure 2.1: *Java Virtual Machine*

JVM is similar to other computers at many levels. It executes instructions and ma-

nipulates memory. Contrary to register-based x86 computers, JVM is a stack-based machine(Fig: 2.1). A stack based virtual machine makes very few assumptions about the target architecture, which makes it easy to implement on a wide variety of hardware. JVM also feature techniques like just-in-time compilation and adaptive optimization, designed to improve performance.

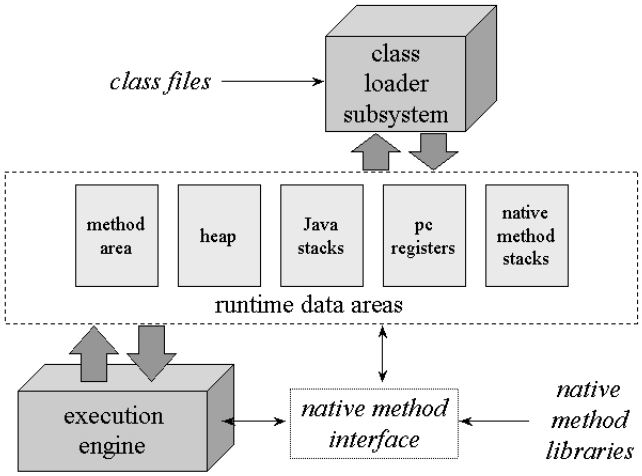

Figure 2.2: *JVM internal architecture*

The behavior of JVM is defined in terms of subsystems, memory areas, data types, and instructions. The abstract nature of the JVM's specification[16] makes it easy to have different implementations on a wide variety of platforms. It gives the individual implementors freedom to design the machine according to their requirement and that of the hardware

As shown in figure 2.2, all JVMs have an execution engine, heap, stack, method area and native methods area. The memory is used for storing class data, global variables, method locals and other intermediate values. Instructions are executed by the execution engine and is one of the main components of the machine. The JVM has no registers, instead it uses stack for all intermediate storage.

The language understood by JVM is called Java bytecode. The bytecode is independent of the source language, which opens a lot of possibilities as other language designers can

use the power of JVM by writing translators which can translate to bytecode. There are many translators capable of translating different languages into Java Bytecode such as the following:

- NestedVM[1] - A C to Java translator
- GCJ[11] - GCC Java compiler
- LLJVM[18] - Java compiler based on LLVM toolchain
- Cibyl[13] - C to Java, targets J2ME devices

2.1.1 Java bytecode Instructions

The Java bytecode is the instruction set of the JVM. The reason behind the term bytecode is that all instructions are of a single byte. A single byte has 256 possible values, which limits the total number of possible instructions to 256. Among them, 0x00 through 0xca, 0xfe, and 0xff are assigned values. Some are special purpose instructions. For example, 0xca for setting a breakpoint in Java debuggers, 0xfe and 0xff are for internal use by JVM. The different instructions fall into the following groups:

- Arithmetic Instructions - xADD, xSUB etc
- Load and Store - xLOAD, xSTORE etc
- Type Conversion Instructions - x2y etc
- Object Creation Manipulation - NEW, NEWARRAY etc
- Control Transfer Instruction - IFEQ, IFNE etc
- Method Invocation and Return Instructions - INVOKEVIRTUAL etc
- Throwing Exceptions - ATHROW
- Monitor Instructions - MONITORENTER, MONITOREXIT

where x and y can be any one of the following data types: B for byte, C for character, I for an Integer, D for double, F for float, L for long, Z for boolean, and A for an object or array data type.

2.1.2 JVM Data types

As shown in Figure 2.3, JVM has two type categories: value (primitive types) and reference (object types). The value types are scalar types and are further classified into three types, numeric types, boolean type and returnAddress type.

Numeric types are divided into integral type and floating point type. Integral types are byte, short, int, long and char. Floating point types are float and double. Boolean type is a true or false value.

Reference types are either: class types, array types, or interface types. They refer to dynamically created objects. An array consists of a component type and a dimension. The component type can again be an array, however, it ultimately resolves to a primary type, class type or interface type which is called the element type of the array.

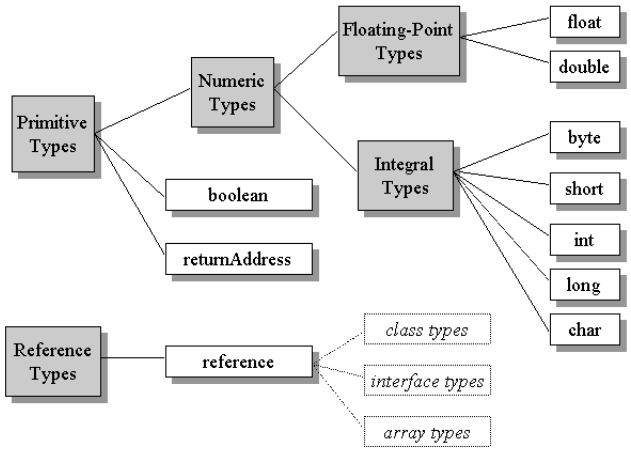

Figure 2.3: *Java Data Types*

2.1.3 Bytecode example

An example Java source file(2.4), which gets compiled to(2.5).

Figure 2.4: *Java Sample*

```
public class Foo {
  private String bar;

  public String getBar(){
    return bar;
  }

  public void setBar(String bar) {
    this.bar = bar;
  }
}
```

Figure 2.5: *Java bytecode for 2.4*

```
public class Foo extends java.lang.Object {
  public Foo();
    Code:
       0:   aload_0
       1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
       4:   return

  public java.lang.String getBar();
    Code:
       0:   aload_0
       1:   getfield        #2; //Field bar:Ljava/lang/String;
       4:   areturn

 public void setBar(java.lang.String);
    Code:
       0:   aload_0
       1:   aload_1
       2:   putfield        #2; //Field bar:Ljava/lang/String;
       5:   return
}
```

2.2 Sireum

Sireum is a collection of tools for software analysis. Sireum's different modules depends on an intermediate representation which is described in the following section:

2.2.1 Pilar

Pilar is the intermediate language used by Sireum framework. It is a Guarded Command language[7] inspired by Bogor Modeling Languague(BIR)[9]. There are many salient features of Pilar which makes it suitable for analysis. It is a simple, flexible and structured language.

The main quality of Pilar is the extensibility. One can add annotations of different kinds, which might or might not be used during the analysis. This also differentiates it from other intermediate representations which have strictly defined semantics. An example of Pilar is shown in Figure 2.6. All the identifiers in generated Pilar are fully qualified,

Figure 2.6: *Pilar Example*

```
procedure (|int|) {|org.sireum.test.jvm.samples.HelloWorld.sum(II)I|} ((|
    int|) [|a|], (|int|) [|b|])
  @MaxLocals 2
  @MaxStack 2
  @Owner (|org.sireum.test.jvm.samples.HelloWorld|)
  @Access (PUBLIC,STATIC)
  @Desc "org.sireum.test.jvm.samples.HelloWorld.sum(II)I"
{
  local jmp;

  #L00000Aa. return ([|a|]+[|b|]);
}
```

"package.Class.name". Another feature is visually separating different classes of identifiers by using different types of delimiters. Method names are shown as "{|package.class.sum(II)I |}", global fields are "+|package.class.field|+", local variables are "[|a|]", types are "(|int|)" and fields are surrounded by "<|package.class.field|>".

2.2.2 Sireum JVM1

Sireum JVM1 is the name of the previous version of current project. It was used to translate Java bytecode to Pilar, however, there were some caveats. Firstly, Sireum JVM1 was not a complete implementation of the Java Virtual Machine specification. It translated bytecode instructions with little modification, in a Pilar compatible format, which caused other problems. Here's an example of Pilar generated using Sireum JVM1:

```
#l1 .
  ALOAD 0; //bytecode instruction
```

This caused the tools used for analysis to be modified to support the generated Pilar. As an example, Kiasan[2.2.4], one of the Sireum tools, had to be modified to interpret bytecode instructions embedded within Pilar generated by Sireum JVM1 effectively creating a Symbolic JVM. However, it does not allowed Kiasan to be used for other languages, resulting in creation of a different Kiasan for other languages such as Spark.

Regardless, Sireum JVM1 gave pointers on how to translate Java bytecode. It used the same libraries, ObjectWeb ASM and StringTemplate, for Pilar generation. The new version described in this report is a completely independent version, and is done from the ground up.

2.2.3 Kiasan

Kiasan[5, 6, 12] is the tool for symbolic execution, which is an effective technique for automatically finding bugs in program. It can provide highly-automated and precise reasoning about programs. Kiasan can dynamically unroll loops and data structures can be explored up to a certain limit. This helps not only with test case generation, but can also aid in the verification of the program.

2.2.4 Alir

Alir[19] is Sireum's data flow analysis tool that can perform various kinds of inter and intra-procedural analyses. The common ones are Control Flow Graph, Reaching Definition

Analysis, Control Dependence Graphs, Data Dependence Graphs, Program Dependence Graphs etc.

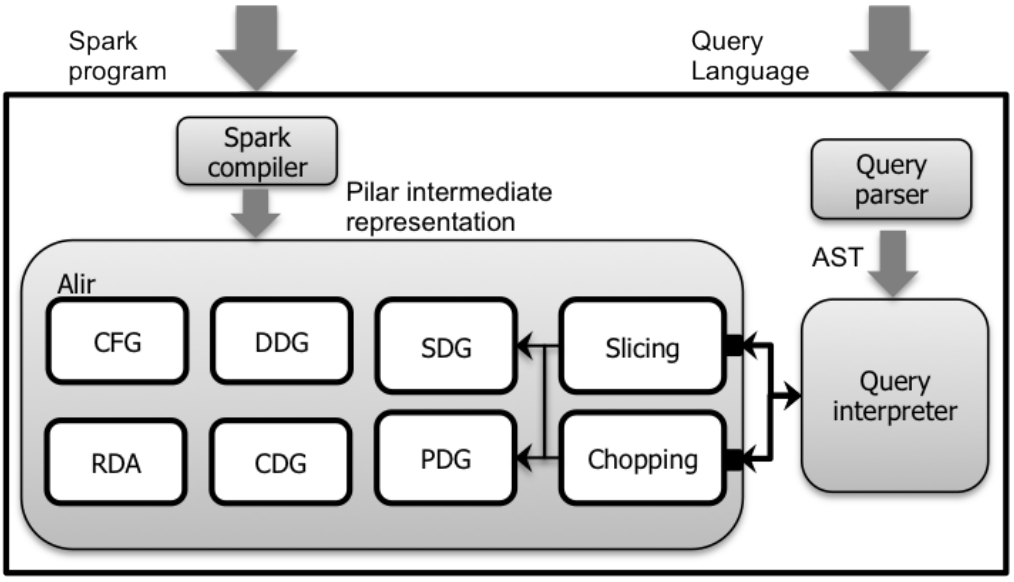

Figure 2.7: *Alir Components; figure taken from 19*

2.2.5 Amandroid

Amandroid is a new framework being developed for Android, which has a tool called Dex2Pilar. It can translate Dalvik[3] dex files into Pilar. The Pilar generated from Dex2Pilar tool was really useful as a reference for this project.

Chapter 3

Translation

The translation of Java bytecode to Pilar involves reading the class file and generating corresponding Pilar instructions. We use ObjectWeb ASM library to read and process Java bytecode.

Translation works by visiting different classes of bytecode instructions and then taking appropriate action. The translator uses a stack data structure to simulate JVM stack machine. As an example, whenever a "ILOAD X" instruction is encountered, the variable X is pushed on the local stack. Similarly, a IADD instruction pops two values from the local stack, creates a new value by adding them, and then pushes it back on the stack. For example, assume the stack has X and Y, if the next instruction is IADD, it will pop X and Y, add them and push the result "X + Y" on the stack.

StringTemplate library is used for defining the templates; which defines the format of Pilar output. It also provides a single place to define the view of generated Pilar. This follows good software practices. The specifics of how it is used will be defined in the subsequent sections. The different templates are hierarchically defined, starting from the Pilar record which represents a Java class.

The different structures in Pilar are defined as classes which are created while visiting the bytecode. It clearly separates responsibilities: visitor does the translation and creates models, the translator translates models using templates to Pilar. This closely resembles the well-known software pattern - Model View Controller(MVC).

The reading of class files or the bytecode is done using the ClassReader class from ASM. ClassReader accepts a ClassVisitor which uses visitor pattern to visit different members of a class. A class visitor translates all java classes, delegating to other visitors like field visitor, annotation visitor, method visitor as required. The exact details of each of these are covered in subsequent sections.

```
ClassVisitor cv = new ClassVisitor(Opcodes.ASM4, cw) { };
ClassReader cr = new ClassReader(b1);
cr.accept(cv, 0);
```

There are two modes of translation:

- Intermediate values in stack - This mode stores the intermediate values while translating the bytecode in Stack and produces Pilar code which resembles the Java source more closely.

```
    ILOAD 1  //Load variable [|v1|] onto stack
    ICONST_2 //Load constant 2 onto stack
    IMUL     //Multiply top two stack variables, ([|v1|] * 2)
    ILOAD 3  //Load variable 3 on stack [|v3|]
    IADD     //Add top two stack variables (([|v1|] * 2) + [|v3|])
    ISTORE 5 //Store the result into variable [|v5|]
```

  translates to:

```
    [|v5|] := (([|v1|] * 2) + [|v3|]) @type (|int|);
```

- Intermediate values in Variables - This mode stores all intermediate value in variables, and then use them for further calculation. It produces code which more closely resembles the bytecode, where each instruction is translated to Pilar code that access at most 3 address.

```
    ILOAD 1
    ICONST_2
    IMUL
    ILOAD 3
    IADD
    ISTORE 5
```

  translates to:

```
s0 := [|v1|]; //ILOAD 1
s1 := 2; //ICONST_2
s3 := s0 * s1; //IMUL
s4 := [|v3|]; //ILOAD 3
s5 := s3 + s4; //IADD
[|v5|] := s5; //ISTORE 5
```

3.1 Class Translation

ClassVisitor is the class responsible for translating Java classes. It visits all the different class bytecode instructions and creates the record model for the class. The different instructions a class bytecode have are as follows.

3.1.1 Fields

A field is a variable inside class and it stores the state of the class. It should have a name, a type, and might have some modifiers like public, private etc. ClassVisitor creates a field model and adds it to a record model. It then delegates the responsibility of visiting further to a FieldVisitor. This instruction doest not use the stack. A few examples :

```
// access flags 0x1
public I field2
```

```
(|type|) <|name|> @AccessFlag (XXX, YYY);
```

A field with public static and final modifiers is treated as a global field.

```
// access flags 0x19
AccessFlags TYPE NAME
```

```
global (|TYPE|) @@+|NAME|+ @AccessFlag (XXX, YYY);
```

3.1.2 Method

Method defines the behavior of a class. It has a method signature, access type and the method body. The method signature consists of the name, the parameter types and their order. The corresponding model in Pilar is called a procedure.

An example of what is read by the ClassVisitor..

```
public static main([Ljava/lang/String;)V throws java/io/IOException java/io/
    FileNotFoundException
```

which is used to create a Procedure model with the given name, signature and accesstypes. It is then passed to MethodVisitor which takes care of the method body.

3.1.3 Annotation

In Java, classes can have annotation and they are used to give meta information. An annotation can be runtime or compile time, it can be used by a compiler or the program using reflection. An annotation is translated by an AnnotationVisitor which is covered later in section 4.3(3.3).

3.2 Method Translation

The translation of a method starts at ClassVisitor level, where a Procedure model is created for the method and then passed to MethodVisitor. A MethodVisitor is responsible for translating the bulk of the method, including method body. The different parts of a method are translated as follows:

3.2.1 Locals

Local variables have a name and a type. In bytecode, they are addressed by a number starting at zero, which is an index into the local variable slots. They are available as debug information, and might not be always available. If present, it is used to get the names of variables.

```
LOCALVARIABLE NAME TYPE; L0 L3 0
```

translates to:

```
(|TYPE|) [|NAME|]
```

and is used within the scope L0 and L3.

JVM also uses local variables to pass method parameters, including the class instance. The length of the local variable array is determined at compile time.

3.2.2 Label

Labels represent a mark position, which is used by JVM to jump to certain locations.

A single label can be associated with more than one bytecode instructions. And so for translating it to Pilar "Aa-Zz" is appended at the end of Label to distinguish different instructions under the same Label. We tried using single letters "a-z" as suffix, but they were not sufficient in some cases and therefore we chose "Aa-Zz". In bytecode, label is just a number followed by a character, usually 'L'.

```
LN
```

translates to:

```
#L0000NAa
```

3.2.3 Load and Store

Load and store instruction transfer values between local variables and operand stack. Loading a local variable from the stack is xLoad, where x represents the data type. It can be i, d, f, z and a. They do not generate anything, except operating on the local stack. An example of load instruction is:

```
ILOAD  2
```

Storing a value from the stack to a local variable is of the form: xStore, where x is a data type.

```
ALOAD  2
ISTORE  3
```

The above statement pops a value from the stack and emits an assignment statement. If we have the debug information and local variable names are present, they are used; otherwise a pseudonym is used.

```
[|v3|] = [|v2|];
```

LDC instruction is similar to the Load instruction, but instead of loading a variable value it loads a constant.

```
LDC X
```

And similar to the LOAD instruction, it does not generate any output and loads a constant value on the stack.

3.2.4 Call Frame

A call frame(Fig: 3.1) is created and destroyed when a method is invoked and when the invocation completes respectively. It is created from the stack area of the thread calling the method. Only one frame is active at any time, called the current frame and the method who ownes it is known as the current method. Similarly, the class owning the method is called current class. There are no instruction for frames define in JVM specs, but ASM

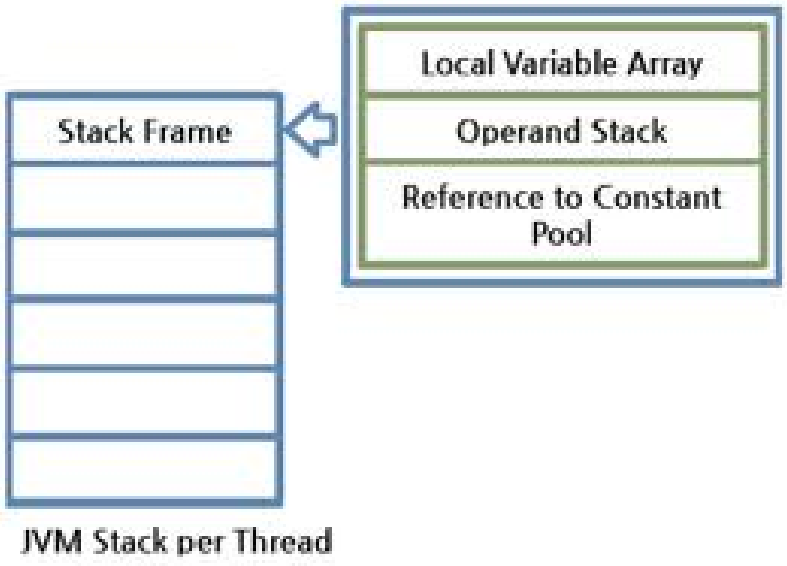

Figure 3.1: *Java Frames*

inserts pseudo instructions for it. They are inserted just before any instruction that follows

an unconditional branch instruction, that is the target of a jump instruction, or that starts an exception handler block.

ASM has the following frame types:

- SAME - Frame with exactly the same locals as the previous frame and with empty stack.
- SAME1 - Frame with exactly the same locals as the previous frame and with the single value on the stack.
- APPEND - Current locals are the same as the locals in previous frame, except additional locals are defined.
- CHOP - Frame with current locals are the same as locals in the previous frame, except the last 1-3 locals are absent and with empty stack.
- FULL - Represents complete frame data.

The template for translation of frame structure is

```
@Frame (@<type>, <nLocal>, '[<local; separator=", ">], <nStack>, '[<stack;
    separator=", ">])
```

where type is the type of the frame, nLocal is the number of locals in the frame, local is the array of local variables, nStack is the number of stack values and stack is the array of stack values.

3.2.5 Object creation and Manipulation

Classes and arrays, both are considered objects but JVM uses different instructions for them.

- NEW - This instruction is used to create a new instance of a class. A new stackvar is created in pilar and then stored in stack on the translator side.

```
    NEW classname
```

translates to:

```
s0 := new (|classname|);
```

- NEWARRAY, ANEWARRAY and MULTINEWARRAY are used to create different kinds of array, however, have a similar translation template.

```
ALOAD dimension
ANEWARRAY type
```

translates to:

```
new (|type|)[dimension]
```

- GETFIELD and PUTFIELD - These instructions are used to access the field members of a class using an object.

```
ALOAD Y
GETFIELD fieldname: type
```

translates to:

```
[|vY|] == null => throwNPE(); | sX = [|vY|].<|fieldname|>
```

where sX is a stack variable. Putfield is similar but pops two values from the stack, the objectref and the value to put in the field.

- GETSTATIC and PUTSTATIC - They are similar to Getfield and Putfield, except that they are used for static fields. Also, instead of an objectref classname is used to access them.

```
GETSTATIC StaticFieldName : Type;
```

translates to:

```
sX := +|StaticFieldName|+ @Type (|Type|);
```

where sX is a stack variable.

- xALOAD - Loads an array component on the stack. This does not emit any instructions, however, pushes value on the stack.

```
xLOAD y
ICONST_0
xALOAD
```

The above sequence of instruction will push "y[0]" on the stack; x represents some data type and y represents a variable.

- xASTORE - Stores a value from the stack to an array index.

```
ALOAD value
xSTORE m
```

translates to:

```
[|ref|]==null +> throwNPE();
| else [|m|]<0 || [|m|]>[|ref|].length +> throwAOB();
| else [|ref|][|m|] := value;
```

where x is the type, m is the index of the local variable array.

- ARRAYLENGTH - A special instruction to get the length of an array.

```
ALOAD m
ARRAYLENGTH
```

translates to:

```
sX := m.length;
```

where sX is the local stack variable.

- INSTANCEOF, CHECKCAST - These instructions are used to check properties of objects. They generate different instructions, the templates are..

```
ALOAD ref
INSTANCEOF Type
```

translates to:

```
sX := [|ref|] <: (|Type|);
```

where sX is a stack variable.

3.2.6 Operand Stack Management

The stack management instructions are pop, pop2, dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2, swap. They do not emit any Pilar instructions, rather operate directly on the stack datastructure used by translator.

3.2.7 Control Transfer Instruction

The instructions transfer the control of JVM from one place to another and are often used for conditional and loop constructs. They can be classified in three different categories.

- Condition Branch - Jumps based on some condition. They include ifeq, ifne, iflt, ifle, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmple, if_icmpgt if_icmpge, if_acmpeq, and if_acmpne.

```
xLOAD vY
IFEQ LX
```

  translates to:

```
if vY==0 then goto LX;
```

  where LX is the label to jump to.

- Compound conditional branch - It can test for multiple conditions, they are tableswitch, lookupswitch. Lookupswitch is similar to tableswitch, except the values need to be continuous in tableswitch.

```
ILOAD i
LOOKUPSWITCH
  x: LX
  y: LY
  z: LZ
  default: LD
```

  translates to:

```
switch [|i|]
  | x => goto LX
  | y => goto LY
```

```
        | z => goto LZ
        | => goto LD;
```

where x, y and z are the values, and LX, LY and LZ are the labels.

- Unconditional branch - GOTO and RET

```
    GOTO LX
```

translates to:

```
    goto LX
```

where LX is the label.

3.2.8 Method Invocation and Return Instructions

Methods are called using different instructions; based on their type. If they are instance methods, they are invoked using INVOKEVIRTUAL, INVOKEINTERFACE, INVOKESPECIAL and INVOKEDYNAMIC. If they are static methods, they are invoked using INVOKESTATIC.

```
  INVOKEx functionName(args1, ..., argsN)returntype
```

translates to:

```
  call temp := functionName(args1, ..., argsN);
```

The args are popped from stack and return type, if not void, is pushed on stack..

3.2.9 Arithmetic Instructions

All the instructions performing some kind of arithmetic, they include addition, subtraction, multiplication, division, remainder, negate, shift, binary operations and a few comparison operations.

```
    xLOAD sY
    xLOAD sZ
    xINS
```

translates to:

```
sT := sY OP sZ;
```

where xINS is an binary arithmetic instruction operating on x data type, and OP is the operator for corresponding instruction.

3.2.10 Type Conversion

Often one type is converted to another, and that's where this instructions are used. They are i2b, i2c, i2s, l2i, f2i, f2l, d2i, d2l, d2f, i2l, i2f, i2d, l2f, l2d, and f2d.

```
xLOAD v1
x2y
```

translates to

```
(|Y|) [| v1 |]
```

where Y represents some data type.

3.2.11 Monitor Instructions

JVM supports synchronization of methods and block of instructions, and for this it has two instructions, MONITORENTER and MONITOREXIT. It first pops an object reference(sX) from the stack, and then emit following Pilar code.

```
sX == null +> throwNPE(); | sX != null && lockAvailable(sX) +> lock(sX);
```

3.2.12 Exceptions

Java has extensive support for exception handling. The bytecode has only one instruction for the same.

```
ALOAD sm
ATHROW
```

translates to:

```
throw sm
```

3.3 Annotation Translation

Annotations adds metadata about the source code. They are used for different purpose, such as:

- Information for the compiler
- Compile-time and deployment-time processing
- Runtime processing.

Annotations can be nested inside each other, and the general template for translation is:

```
@Lpackage/class/Name(name1 = value1 , ... , nameN = valueN)
```

gets translated to:

```
@(|package.class.Name|)(name1 = value1 , ... , nameN = valueN)
```

Chapter 4

Evaluation

This section evaluates the translation of several Java libraries using the translator and the testing strategies used for evaluation.

4.1 Testing

The test project translates the classes and then uses ChunkingPilarParser to test if the generated files are parsing properly. This assures that the generated Pilar is syntactically correct. If there are expected output files, it also test if those matches the resulting output files.

There were several sample classes whose translation was inspected manually, and then those files were used as expected output.

4.2 Results

We have translated the complete Scala library using the translator, and it was tested using the above strategy. Since there were no expected output files, it only went through the first phase which verifies the syntax of generated files. Also, many Java and Scala classes were translated for those we had expected output and the generated output matched the expected ones.

As shown in table: 4.1, we also report some statistics on the speed of translation. It is

Project	Lines of Code	Translation time	Total time
Sireum JVM	23	3.259s	4.816s
Sireum Core	31814	30.15s	44.481s
Scala Collection	44016	36.083s	43.445s
Scala	210000	213.43s	302.588s

Table 4.1: *Results*

found to be linear in the lines of code.

Chapter 5

Related Work

5.1 LLVM IR

LLVM[14] is a collection of compiler technologies, based on an intermediate language representation, known as LLMV IR.

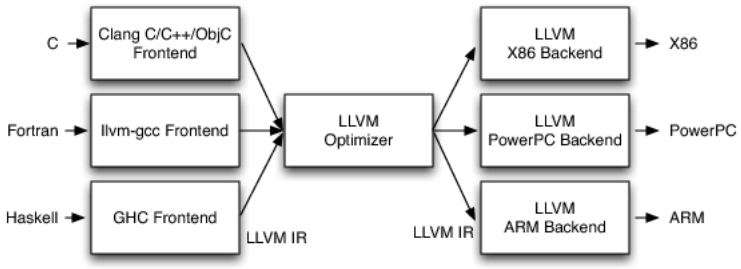

Figure 5.1: *LLVM Implementation of Three phase design; figure taken from 14*

The LLVM IR is the most important aspect of its design. It is the form used to represent code in the compiler. LLVM IR is mainly designed to be easy for mid-level analysis and transformation, which is generally done in the optimizer section of the compiler(Fig: 5.1). The specific goals the developer had when designing the IR were support for lightweight runtime optimizations, cross functional interprocedural optimizations, whole program analysis, and aggressive restructuring transformations. Another important aspect of the IR is itself a first-class language with well defined semantics. Here's an example of a C function and its corresponsing LLVM intermediate representation:

```
unsigned add1(unsigned a, unsigned b) {
```

```
  return a+b;
}
```

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}
```

LLVM IR is strongly typed compared to other low-level languages. Rather than using fixed set of registers in the representation, LLVM uses Infinite temporaries named with a %character.

Pilar is similar to LLVM in many aspects. It is designed for analysis, has a type system and is a first-class language. However, it differs in other aspects. It does not have fixed semantics, and is very flexible.

5.2 GCC RTL

GCC uses Register Transfer Language(RTL)[10] as a low-level intermediate representation, which is very close to assembly language. The source code is first translated into a tree(GIMPLE) representation, which is the central data structure for the GCC Front end. The tree is translated into RTL.

RTL instructions are described in an algebraic form that describes what the instruction does. It is inspired by Lisp lists. It has two forms; one used internally and other used for debugging(a textual description). The textual form looks very similar to lisp and uses parentheses to indicate pointers.

```
(set (reg:SI 140)
     (plus:SI (reg:SI 138)
              (reg:SI 139)))
```

5.3 Boogie

Boogie[2, 15] is a program verifier built for verifying Spec# programs in the .NET object oriented framework. It performs a series of transformations from the source program to

verification conditions to an error report. The Boogie pipeline is centered around BoogiePL, an intermediate representation tailored for expressing proofs and obligations. It separates the semantics of a program from generating proof obligations.

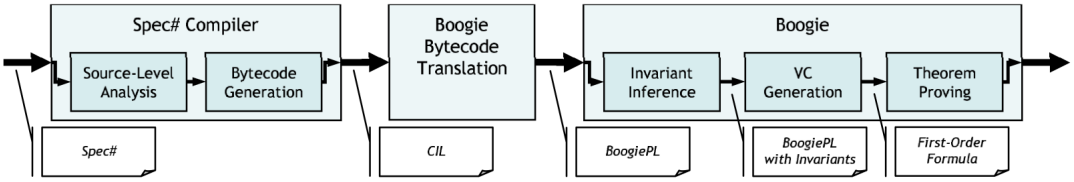

Figure 5.2: *Boogie Pipeline, figure taken from 2*

BoogiePL provides assert statements to encode proof obligations and assume statements for properties guaranteed by the source language. It also includes declarations for mathematical functions and axioms.

```
procedure F(n: int) returns (r: int)
  ensures 100 < n ==> r == n - 10;  // This postcondition is easy to check
  ensures n <= 100 ==> r == 91;     // Do you believe this one is true?
{
  if (100 < n) {
    r := n - 10;
  } else {
    call r := F(n + 11);
    call r := F(r);
  }
}
```

Listing 5.1: *Example taken from 2*

Like Pilar, BoogiePL can be represented in a textual form, and parsed from it. This makes it convenient for debugging and other similar purposes. In contrast to Pilar which is designed for different kinds of analysis, BoogiePL is meant for verification of correctness using VCGen approach with a fixed semantics.

Chapter 6

Future Work

The Sireum-JVM project has lot of scope. The next step can be to add support for different kinds of translation and many more languages. The most important one would be to make it faster. There are lot of optimizations that can be done. It takes quite some time to run for big libraries.

Another idea would be to translate Pilar back to Java Bytecode. It will present interesting problems and will open several possibilities. Another possible work would be to write a concrete interpreter for Pilar, so as to run the code directly.

Bibliography

[1] Brian Alliet and Adam Megacz. Complete translation of unsafe native code to safe bytecode. In *IVME '04: Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, pages 32–41, New York, NY, USA, 2004. ACM.

[2] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs 0002, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[3] Dan Bornstein. Dalvik virtual machine internals. Google I/O 2008, Juni 2008.

[4] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Bandera: a source-level interface for model checking java programs. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *ICSE*, pages 762–765. ACM, 2000.

[5] Xianghua Deng, Jooyong Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 157–166, 2006.

[6] Xianghua Deng, Robby, and John Hatcliff. Kiasan: A verification and test-case generation framework for java based on symbolic execution. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, ISOLA '06, pages 137–, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[8] Michael Dunn. Toyota's killer firmware. http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences, 2013.

[9] Matthew B. Dwyer, John Hatcliff, and Matthew Hoosier. Supporting model checking education using bogor/eclipse. In Michael G. Burke, editor, *ETX*, pages 88–92. ACM, 2004.

[10] GCC. Gcc rtl representation, 2013.

[11] GCC. Gcj : Gnu compile for java programming language, 2013.

[12] John Hatcliff, Robby, Patrice Chalin, and Jason Belt. Explicating symbolic execution (xsymexe): An evidence-based verification framework. In *Proc. ICSE*, pages 222–231. IEEE, 2013.

[13] Simon Kågström, Håkan Grahn, and Lars Lundberg. Cibyl: An environment for language diversity on mobile devices. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 75–82, New York, NY, USA, 2007. ACM.

[14] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[15] K. Rustan M. Leino. This is boogie 2, 2008.

[16] Oracle, 2013.

[17] Robby. Indus, 2006.

[18] David A. Roberts. Lljvm, 2013.

[19] Hariharan Thiagarajan, John Hatcliff, Jason Belt, and Robby. Bakar alir: Supporting developers in construction of information flow contracts in spark. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:132–137, 2012.

Appendix A

A complete example

```
// class version 51.0 (51)
// DEPRECATED
// access flags 0x20021
public class org/sireum/test/jvm/samples/HelloWorld2 {

  // compiled from: HelloWorld2.java

  @Ljava/lang/Deprecated;()
  // access flags 0x0
  INNERCLASS org/sireum/test/jvm/samples/HelloWorld2$Point org/sireum/test/
      jvm/samples/HelloWorld2 Point

  // access flags 0x19
  public final static I field = 9

  // access flags 0x1
  public I field2

  // access flags 0x1
  public Lorg/sireum/test/jvm/samples/HelloWorld2$Point; p

  // access flags 0x1
  public <init>()V
   L0
    LINENUMBER 13 L0
    ALOAD 0
    INVOKESPECIAL java/lang/Object.<init>()V
   L1
    LINENUMBER 16 L1
    ALOAD 0
    NEW org/sireum/test/jvm/samples/HelloWorld2$Point
    DUP
    ALOAD 0
    ICONST_1
    ICONST_2
```

```
    INVOKESPECIAL org/sireum/test/jvm/samples/HelloWorld2$Point.<init>(Lorg/
        sireum/test/jvm/samples/HelloWorld2;II)V
    PUTFIELD org/sireum/test/jvm/samples/HelloWorld2.p : Lorg/sireum/test/jvm
        /samples/HelloWorld2$Point;
   L2
    LINENUMBER 13 L2
    RETURN
   L3
    LOCALVARIABLE this Lorg/sireum/test/jvm/samples/HelloWorld2; L0 L3 0
    MAXSTACK = 6
    MAXLOCALS = 1

  // access flags 0x9
  public static main([Ljava/lang/String;)V throws java/io/IOException java/io
      /FileNotFoundException
  @Lcom/google/common/annotations/Beta;() // invisible
    TRYCATCHBLOCK L0 L1 L2 java/lang/ArithmeticException
    TRYCATCHBLOCK L3 L4 L5 java/lang/ArithmeticException
   L6
    LINENUMBER 21 L6
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    LDC "hello"
    INVOKEVIRTUAL java/io/PrintStream.println(Ljava/lang/String;)V
   L7
    LINENUMBER 22 L7
    ICONST_1
    ISTORE 1
   L8
    LINENUMBER 23 L8
    ICONST_2
    ISTORE 2
   L9
    LINENUMBER 24 L9
    ILOAD 1
    ILOAD 2
    IADD
    ISTORE 3
   L10
    LINENUMBER 26 L10
    ILOAD 1
    INEG
    ISTORE 4
   L11
    LINENUMBER 27 L11
    ILOAD 1
    I2L
    LSTORE 5
   L12
    LINENUMBER 29 L12
    NEW org/sireum/test/jvm/samples/HelloWorld2
    DUP
```

```
 INVOKESPECIAL org/sireum/test/jvm/samples/HelloWorld2.<init>()V
 ASTORE 7
L13
 LINENUMBER 30 L13
 ALOAD 7
 BIPUSH 9
 NEW org/sireum/test/jvm/samples/HelloWorld2
 DUP
 INVOKESPECIAL org/sireum/test/jvm/samples/HelloWorld2.<init>()V
 GETFIELD org/sireum/test/jvm/samples/HelloWorld2.field2 : I
 IADD
L14
 LINENUMBER 31 L14
 ILOAD 1
 ILOAD 2
 IF_ICMPLE L15
 ICONST_2
 GOTO L16
L15
FRAME FULL [[Ljava/lang/String; I I I I J org/sireum/test/jvm/samples/
    HelloWorld2] [org/sireum/test/jvm/samples/HelloWorld2 I]
 ICONST_3
L16
FRAME FULL [[Ljava/lang/String; I I I I J org/sireum/test/jvm/samples/
    HelloWorld2] [org/sireum/test/jvm/samples/HelloWorld2 I I]
 IADD
 NEW org/sireum/test/jvm/samples/HelloWorld2
 DUP
 INVOKESPECIAL org/sireum/test/jvm/samples/HelloWorld2.<init>()V
 GETFIELD org/sireum/test/jvm/samples/HelloWorld2.field2 : I
 IADD
L17
 LINENUMBER 30 L17
 PUTFIELD org/sireum/test/jvm/samples/HelloWorld2.field2 : I
L18
 LINENUMBER 32 L18
 ALOAD 7
 GETFIELD org/sireum/test/jvm/samples/HelloWorld2.p : Lorg/sireum/test/jvm
     /samples/HelloWorld2$Point;
 ICONST_3
 PUTFIELD org/sireum/test/jvm/samples/HelloWorld2$Point.x : I
L19
 LINENUMBER 34 L19
 BIPUSH 10
 NEWARRAY T_INT
 ASTORE 8
L20
 LINENUMBER 35 L20
 IINC 4 1
L21
 LINENUMBER 37 L21
```

```
    ILOAD 1
    ILOAD 2
    ICONST_3
    IMUL
    IADD
    ICONST_4
    ISUB
    ISTORE 9
   L22
    LINENUMBER 39 L22
    ILOAD 1
    ILOAD 2
    IF_ICMPGE L23
   L24
    LINENUMBER 40 L24
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    LDC "less than"
    INVOKEVIRTUAL java/io/PrintStream.println(Ljava/lang/String;)V
   L23
    LINENUMBER 42 L23
   FRAME APPEND [[I I]
    ILOAD 1
    IFNE L25
   L26
    LINENUMBER 43 L26
    ALOAD 7
    IFNULL L25
   L27
    LINENUMBER 44 L27
    ALOAD 7
    ILOAD 1
    ILOAD 2
    INVOKEVIRTUAL org/sireum/test/jvm/samples/HelloWorld2.sum(II)I
    POP
   L25
    LINENUMBER 48 L25
   FRAME SAME
    ICONST_0
    ISTORE 10
   L28
    GOTO L29
   L30
    LINENUMBER 49 L30
   FRAME APPEND [I]
    ALOAD 7
    ILOAD 10
    ILOAD 2
    INVOKEVIRTUAL org/sireum/test/jvm/samples/HelloWorld2.sum(II)I
    POP
   L31
    LINENUMBER 48 L31
```

```
  IINC 10 1
 L29
 FRAME SAME
  ILOAD 10
  BIPUSH 10
  IF_ICMPLT L30
 L32
  LINENUMBER 52 L32
  ALOAD 7
  INSTANCEOF org/sireum/test/jvm/samples/HelloWorld2
  IFEQ L33
 L34
  LINENUMBER 53 L34
  ALOAD 7
  ILOAD 1
  ILOAD 2
  INVOKEVIRTUAL org/sireum/test/jvm/samples/HelloWorld2.sum(II)I
  POP
 L33
  LINENUMBER 56 L33
 FRAME CHOP 1
  LDC "adfa"
  ASTORE 10
 L35
  LINENUMBER 57 L35
  ALOAD 10
  CHECKCAST java/lang/String
  ASTORE 11
 L36
  LINENUMBER 59 L36
  BIPUSH 10
  ANEWARRAY org/sireum/test/jvm/samples/HelloWorld2
  ASTORE 12
 L0
  LINENUMBER 61 L0
  ALOAD 12
  BIPUSH 9
  AALOAD
  ICONST_2
  PUTFIELD org/sireum/test/jvm/samples/HelloWorld2.field2 : I
 L1
  LINENUMBER 62 L1
  GOTO L3
 L2
 FRAME FULL [[Ljava/lang/String; I I I I J org/sireum/test/jvm/samples/
     HelloWorld2 [I I java/lang/Object java/lang/String [Lorg/sireum/test/
     jvm/samples/HelloWorld2;] [java/lang/ArithmeticException]
  ASTORE 13
 L3
  LINENUMBER 67 L3
 FRAME SAME
```

```
    ICONST_2
    ISTORE 1
   L4
    LINENUMBER 68 L4
    GOTO L37
   L5
   FRAME SAME1 java/lang/ArithmeticException
    ASTORE 13
   L37
    LINENUMBER 72 L37
   FRAME SAME
    ILOAD 1
    LOOKUPSWITCH
       1: L38
       200: L39
       3000: L40
       default: L41
   L38
    LINENUMBER 74 L38
   FRAME SAME
    ICONST_2
    ISTORE 1
   L39
    LINENUMBER 76 L39
   FRAME SAME
    ICONST_3
    ISTORE 1
   L40
    LINENUMBER 78 L40
   FRAME SAME
    ICONST_4
    ISTORE 1
   L41
    LINENUMBER 81 L41
   FRAME SAME
    ILOAD 1
    TABLESWITCH
       1: L42
       2: L43
       3: L44
       default: L45
   L42
    LINENUMBER 83 L42
   FRAME SAME
    RETURN
   L43
    LINENUMBER 85 L43
   FRAME SAME
    RETURN
   L44
    LINENUMBER 87 L44
```

```
    FRAME SAME
     RETURN
    L45
     LINENUMBER 89 L45
    FRAME SAME
     RETURN
    L46
     LOCALVARIABLE args [Ljava/lang/String; L6 L46 0
     LOCALVARIABLE i I L8 L46 1
     LOCALVARIABLE j I L9 L46 2
     LOCALVARIABLE k I L10 L46 3
     LOCALVARIABLE l I L11 L46 4
     LOCALVARIABLE l2 J L12 L46 5
     LOCALVARIABLE hw Lorg/sireum/test/jvm/samples/HelloWorld2; L13 L46 7
     LOCALVARIABLE arr [I L20 L46 8
     LOCALVARIABLE adf I L22 L46 9
     LOCALVARIABLE laf I L28 L32 10
     LOCALVARIABLE o Ljava/lang/Object; L35 L46 10
     LOCALVARIABLE lajfa Ljava/lang/String; L36 L46 11
     LOCALVARIABLE hw2 [Lorg/sireum/test/jvm/samples/HelloWorld2; L0 L46 12
     MAXSTACK = 4
     MAXLOCALS = 14

   // access flags 0x1
   public sum(II)I
    L0
     LINENUMBER 96 L0
     ILOAD 1
     ILOAD 2
     IADD
     IRETURN
    L1
     LOCALVARIABLE this Lorg/sireum/test/jvm/samples/HelloWorld2; L0 L1 0
     LOCALVARIABLE i I L0 L1 1
     LOCALVARIABLE j I L0 L1 2
     MAXSTACK = 2
     MAXLOCALS = 3
}
```

```
record (|org.sireum.test.jvm.samples.HelloWorld2|)
    @Source "HelloWorld2.java"
    @Type class
    @AccessFlag (PUBLIC)
    @Annotation (|java.lang.Deprecated;|)
    @InnerClass (
      @Name (|org.sireum.test.jvm.samples.HelloWorld2$Point|),
      @OuterName (|org.sireum.test.jvm.samples.HelloWorld2|),
      @InnerName Point,
      @AccessFlag ()
     )
  extends
```

```
    (|java.lang.Object|)
{
  (|int|) <|HelloWorld2.field2|> @AccessFlag (PUBLIC);
  (|org.sireum.test.jvm.samples.HelloWorld2$Point|) <|HelloWorld2.p|>
     @AccessFlag (PUBLIC);
}
  global (|int|) @@+|HelloWorld2.field|+ @AccessFlag (PUBLIC,STATIC,FINAL);

  procedure (|void|) {|org.sireum.test.jvm.samples.HelloWorld2.<init>()V|}
     ((|org.sireum.test.jvm.samples.HelloWorld2|) [|this|])
    @MaxLocals 1
    @MaxStack 6
    @Owner (|org.sireum.test.jvm.samples.HelloWorld2|)
    @Access (PUBLIC,CONSTRUCTOR)
    @Desc "org.sireum.test.jvm.samples.HelloWorld2.<init>()V"
  {
    local jmp;
    s0;
    s1;

    #L00000Aa. call s0 := {|java.lang.Object.<init>()V|}([|this|])
        @ClassDescriptor (|java.lang.Object|) @Type special;
    #L00001Aa. s0 := new (|org.sireum.test.jvm.samples.HelloWorld2$Point|);
    #L00001Ab. call s1 := {|org.sireum.test.jvm.samples.HelloWorld2$Point.<
        init>(Lorg.sireum.test.jvm.samples.HelloWorld2;II)V|}(s0,[|this|],1,2)
         @ClassDescriptor (|org.sireum.test.jvm.samples.HelloWorld2$Point|)
        @Type special;
    #L00001Ac. [|this|].<|org.sireum.test.jvm.samples.HelloWorld2.p|> := s0
        @Type (|org.sireum.test.jvm.samples.HelloWorld2$Point|);
    #L00002Aa. return @void;
  }
  procedure (|void|) {|org.sireum.test.jvm.samples.HelloWorld2.main([Ljava.
     lang.String;)V|} ((|java.lang.String[]|) [|args|])
    @Throws [|java.io.IOException|],[|java.io.FileNotFoundException|]
    @MaxLocals 14
    @MaxStack 4
    @Owner (|org.sireum.test.jvm.samples.HelloWorld2|)
    @Annotation (|Lcom.google.common.annotations.Beta;|)
    @Access (PUBLIC,STATIC)
    @Desc "org.sireum.test.jvm.samples.HelloWorld2.main([Ljava.lang.String;)V
        "
  {
    local jmp;
    s0;
    s1;
    s2;
    s3;
    s4;
    s5;
    (|int|) [|i|];
    (|int|) [|j|];
```

```
(|int|) [|k|];
(|int|) [|l|];
(|long|) [|l2|];
(|org.sireum.test.jvm.samples.HelloWorld2|) [|hw|];
(|int[]|) [|arr|];
(|int|) [|adf|];
(|int|) [|laf|];
(|java.lang.Object|) [|o|];
(|java.lang.String|) [|lajfa|];
(|org.sireum.test.jvm.samples.HelloWorld2[]|) [|hw2|];

#L00006Aa. s0 := +|java.lang.System.out|+ @Type (|java.io.PrintStream|);
#L00006Ab. call s1 := {|java.io.PrintStream.println(Ljava.lang.String;)V
    |}(s0,"hello") @ClassDescriptor (|java.io.PrintStream|) @Type virtual;
#L00007Aa. [|i|] := 1 @Type (|int|);
#L00008Aa. [|j|] := 2 @Type (|int|);
#L00009Aa. [|k|] := ([|i|]+[|j|]) @Type (|int|);
#L00010Aa. [|l|] := -[|i|] @Type (|int|);
#L00011Aa. [|l2|] := ((|long|))[|i|] @Type (|long|);
#L00012Aa. s0 := new (|org.sireum.test.jvm.samples.HelloWorld2|);
#L00012Ab. call s1 := {|org.sireum.test.jvm.samples.HelloWorld2.<init>()
    V|}(s0) @ClassDescriptor (|org.sireum.test.jvm.samples.HelloWorld2|)
    @Type special;
#L00012Ac. [|hw|] := s0 @Type (|java.lang.Object|);
#L00013Aa. s0 := new (|org.sireum.test.jvm.samples.HelloWorld2|);
#L00013Ab. call s1 := {|org.sireum.test.jvm.samples.HelloWorld2.<init>()
    V|}(s0) @ClassDescriptor (|org.sireum.test.jvm.samples.HelloWorld2|)
    @Type special;
#L00013Ac. s2 := s0.<|org.sireum.test.jvm.samples.HelloWorld2.field2|>
    @Type (|int|);
#L00014Aa. if [|i|] <= [|j|] then goto L00015Aa;
#L00014Ab. jmp := 2;
#L00014Ac. goto L00016Aa;
#L00015Aa. <@Frame (@Full, 7, `[(|java.lang.String[]|), (|int|), (|int|),
     (|int|), (|int|), (|long|), (|java.lang.Object|), (|top|), (|top|),
    (|top|), (|top|), (|top|), (|top|), (|top|)], 2, `[(|java.lang.Object
    |), (|int|), (|top|), (|top|)])>
#L00015Ab. jmp := 3;
#L00016Aa. <@Frame (@Full, 7, `[(|java.lang.String[]|), (|int|), (|int|),
     (|int|), (|int|), (|long|), (|java.lang.Object|), (|top|), (|top|),
    (|top|), (|top|), (|top|), (|top|), (|top|)], 3, `[(|java.lang.Object
    |), (|int|), (|int|), (|top|)])>
#L00016Ab. s3 := new (|org.sireum.test.jvm.samples.HelloWorld2|);
#L00016Ac. call s4 := {|org.sireum.test.jvm.samples.HelloWorld2.<init>()
    V|}(s3) @ClassDescriptor (|org.sireum.test.jvm.samples.HelloWorld2|)
    @Type special;
#L00016Ad. s5 := s3.<|org.sireum.test.jvm.samples.HelloWorld2.field2|>
    @Type (|int|);
#L00017Aa. [|hw|].<|org.sireum.test.jvm.samples.HelloWorld2.field2|> :=
    (((9+s2)+jmp)+s5) @Type (|int|);
#L00018Aa. s0 := [|hw|].<|org.sireum.test.jvm.samples.HelloWorld2.p|>
```

```
    @Type (|org.sireum.test.jvm.samples.HelloWorld2$Point|);
#L00018Ab. s0.<|org.sireum.test.jvm.samples.HelloWorld2$Point.x|> := 3
    @Type (|int|);
#L00019Aa. s0 := new (|int|)[10];
#L00019Ab. [|arr|] := s0 @Type (|int|);
#L00020Aa. [|l|] := [|l|] + 1 @Type (|int|);
#L00021Aa. [|adf|] := (([|i|]+([|j|]*3))-4) @Type (|int|);
#L00022Aa. if [|i|] >= [|j|] then goto L00023Aa;
#L00024Aa. s0 := +|java.lang.System.out|+ @Type (|java.io.PrintStream|);
#L00024Ab. call s1 := {|java.io.PrintStream.println(Ljava.lang.String;)V
    |}(s0,"less than") @ClassDescriptor (|java.io.PrintStream|) @Type
    virtual;
#L00023Aa. <@Frame (@Append, 2, `[(|int[]|), (|int|), (|int|), (|int|),
    (|int|), (|long|), (|java.lang.Object|), (|top|), (|top|), (|top|), (|
    top|), (|top|), (|top|), (|top|)], 0, `[(|java.lang.Object|), (|int|),
     (|int|), (|top|)])>
#L00023Ab. if [|i|] != 0 then goto L00025Aa;
#L00026Aa. if [|hw|] == null then goto L00025Aa;
#L00027Aa. call s0 := {|org.sireum.test.jvm.samples.HelloWorld2.sum(II)I
    |}([|hw|],[|i|],[|j|]) @ClassDescriptor (|org.sireum.test.jvm.samples.
    HelloWorld2|) @Type virtual;
#L00025Aa. <@Frame (@Same, 0, `[(|int[]|), (|int|), (|int|), (|int|), (|
    int|), (|long|), (|java.lang.Object|), (|top|), (|top|), (|top|), (|
    top|), (|top|), (|top|), (|top|)], 0, `[(|java.lang.Object|), (|int|),
     (|int|), (|top|)])>
#L00025Ab. [|laf|] := 0 @Type (|int|);
#L00028Aa. goto L00029Aa;
#L00030Aa. <@Frame (@Append, 1, `[(|int|), (|int|), (|int|), (|int|), (|
    int|), (|long|), (|java.lang.Object|), (|top|), (|top|), (|top|), (|
    top|), (|top|), (|top|), (|top|)], 0, `[(|java.lang.Object|), (|int|),
     (|int|), (|top|)])>
#L00030Ab. call s0 := {|org.sireum.test.jvm.samples.HelloWorld2.sum(II)I
    |}([|hw|],[|laf|],[|j|]) @ClassDescriptor (|org.sireum.test.jvm.
    samples.HelloWorld2|) @Type virtual;
#L00031Aa. [|laf|] := [|laf|] + 1 @Type (|int|);
#L00029Aa. <@Frame (@Same, 0, `[(|int|), (|int|), (|int|), (|int|), (|int
    |), (|long|), (|java.lang.Object|), (|top|), (|top|), (|top|), (|top|)
    , (|top|), (|top|), (|top|)], 0, `[(|java.lang.Object|), (|int|), (|
    int|), (|top|)])>
#L00029Ab. if [|laf|] < 10 then goto L00030Aa;
#L00032Aa. s0 := instanceof @varname [|hw|] @Type "(|org.sireum.test.jvm.
    samples.HelloWorld2|)";
#L00032Ab. if s0 == 0 then goto L00033Aa;
#L00034Aa. call s0 := {|org.sireum.test.jvm.samples.HelloWorld2.sum(II)I
    |}([|hw|],[|i|],[|j|]) @ClassDescriptor (|org.sireum.test.jvm.samples.
    HelloWorld2|) @Type virtual;
#L00033Aa. <@Frame (@Chop, 1, `[(|int|), (|int|), (|int|), (|int|), (|int
    |), (|long|), (|java.lang.Object|), (|top|), (|top|), (|top|), (|top|)
    , (|top|), (|top|), (|top|)], 0, `[(|java.lang.Object|), (|int|), (|
    int|), (|top|)])>
#L00033Ab. [|o|] := "adfa" @Type string;
```

```
#L00035Aa. [|lajfa|] := ((|java.lang.String|)) [|o|] @Type (|java.lang.
    String|);
#L00036Aa. s0 := new (|org.sireum.test.jvm.samples.HelloWorld2|)[10];
#L00036Ab. [|hw2|] := s0 @Type (|java.lang.Object|);
#L00000Aa. s0 := [|hw2|][9];
#L00000Ab. s0.<|org.sireum.test.jvm.samples.HelloWorld2.field2|> := 2
    @Type (|int|);
#L00001Aa. goto L00003Aa;
#L00002Aa. <@Frame (@Full, 12, `[(|java.lang.String[]|), (|int|), (|int|)
    , (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|)
    , (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
    samples.HelloWorld2[]|), (|top|), (|top|)], 1, `[(|java.lang.Object|),
     (|int|), (|int|), (|top|)])>
#L00002Ab. [|v13|] := "Exception" @Type (|java.lang.Object|);
#L00003Aa. <@Frame (@Same, 0, `[(|java.lang.String[]|), (|int|), (|int|),
     (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|),
     (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
    samples.HelloWorld2[]|), (|top|), (|top|)], 0, `[(|java.lang.Object|),
     (|int|), (|int|), (|top|)])>
#L00003Ab. [|i|] := 2 @Type (|int|);
#L00004Aa. goto L00037Aa;
#L00005Aa. <@Frame (@Same1, 0, `[(|java.lang.String[]|), (|int|), (|int|)
    , (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|)
    , (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
    samples.HelloWorld2[]|), (|top|), (|top|)], 1, `[(|java.lang.Object|),
     (|int|), (|int|), (|top|)])>
#L00005Ab. [|v13|] := "Exception" @Type (|java.lang.Object|);
#L00037Aa. <@Frame (@Same, 0, `[(|java.lang.String[]|), (|int|), (|int|),
     (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|),
     (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
    samples.HelloWorld2[]|), (|top|), (|top|)], 0, `[(|java.lang.Object|),
     (|int|), (|int|), (|top|)])>
#L00037Ab. switch [|i|]
             | 1 => goto L00038Aa
             | 200 => goto L00039Aa
             | 3000 => goto L00040Aa
             | => goto L00041Aa;
#L00038Aa. <@Frame (@Same, 0, `[(|java.lang.String[]|), (|int|), (|int|),
     (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|),
     (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
    samples.HelloWorld2[]|), (|top|), (|top|)], 0, `[(|java.lang.Object|),
     (|int|), (|int|), (|top|)])>
#L00038Ab. [|i|] := 2 @Type (|int|);
#L00039Aa. <@Frame (@Same, 0, `[(|java.lang.String[]|), (|int|), (|int|),
     (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|),
     (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
    samples.HelloWorld2[]|), (|top|), (|top|)], 0, `[(|java.lang.Object|),
     (|int|), (|int|), (|top|)])>
#L00039Ab. [|i|] := 3 @Type (|int|);
#L00040Aa. <@Frame (@Same, 0, `[(|java.lang.String[]|), (|int|), (|int|),
     (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|),
```

```
      (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
      samples.HelloWorld2[]|), (|top|), (|top|)], 0, `[(|java.lang.Object|),
      (|int|), (|int|), (|top|)])>
  #L00040Ab. [|i|] := 4 @Type (|int|);
  #L00041Aa. <@Frame (@Same, 0, `[(|java.lang.String[]|), (|int|), (|int|),
      (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|),
      (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
      samples.HelloWorld2[]|), (|top|), (|top|)], 0, `[(|java.lang.Object|),
      (|int|), (|int|), (|top|)])>
  #L00041Ab. switch [|i|]
               | 1 => goto L00043Aa
               | 2 => goto L00044Aa
               | 3 => goto L00045Aa
               | => goto L00042Aa;
  #L00043Aa. <@Frame (@Same, 0, `[(|java.lang.String[]|), (|int|), (|int|),
      (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|),
      (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
      samples.HelloWorld2[]|), (|top|), (|top|)], 0, `[(|java.lang.Object|),
      (|int|), (|int|), (|top|)])>
  #L00043Ab. return @void;
  #L00044Aa. <@Frame (@Same, 0, `[(|java.lang.String[]|), (|int|), (|int|),
      (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|),
      (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
      samples.HelloWorld2[]|), (|top|), (|top|)], 0, `[(|java.lang.Object|),
      (|int|), (|int|), (|top|)])>
  #L00044Ab. return @void;
  #L00045Aa. <@Frame (@Same, 0, `[(|java.lang.String[]|), (|int|), (|int|),
      (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|),
      (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
      samples.HelloWorld2[]|), (|top|), (|top|)], 0, `[(|java.lang.Object|),
      (|int|), (|int|), (|top|)])>
  #L00045Ab. return @void;
  #L00042Aa. <@Frame (@Same, 0, `[(|java.lang.String[]|), (|int|), (|int|),
      (|int|), (|int|), (|long|), (|java.lang.Object|), (|int[]|), (|int|),
      (|java.lang.Object|), (|java.lang.Object|), (|org.sireum.test.jvm.
      samples.HelloWorld2[]|), (|top|), (|top|)], 0, `[(|java.lang.Object|),
      (|int|), (|int|), (|top|)])>
  #L00042Ab. return @void;
    catch (|java.lang.ArithmeticException|) @[L00000Aa..L00001Aa] goto
      L00002Aa;
    catch (|java.lang.ArithmeticException|) @[L00003Aa..L00004Aa] goto
      L00005Aa;
}
procedure (|int|) {|org.sireum.test.jvm.samples.HelloWorld2.sum(II)I|} ((|
    org.sireum.test.jvm.samples.HelloWorld2|) [|this|], (|int|) [|i|], (|int
    |) [|j|])
  @MaxLocals 3
  @MaxStack 2
  @Owner (|org.sireum.test.jvm.samples.HelloWorld2|)
  @Access (PUBLIC)
  @Desc "org.sireum.test.jvm.samples.HelloWorld2.sum(II)I"
```

```
  {
    local jmp;

    #L00000Aa. return ([|i|]+[|j|]);
  }

record (|org.sireum.test.jvm.samples.HelloWorld2$Point|)
    @Source "HelloWorld2.java"
    @Type class
    @AccessFlag ()
    @InnerClass (
      @Name (|org.sireum.test.jvm.samples.HelloWorld2$Point|),
      @OuterName (|org.sireum.test.jvm.samples.HelloWorld2|),
      @InnerName Point,
      @AccessFlag ()
     )
  extends
    (|java.lang.Object|)
{
  (|int|) <|HelloWorld2$Point.x|> @AccessFlag ();
  (|int|) <|HelloWorld2$Point.y|> @AccessFlag ();
  (|org.sireum.test.jvm.samples.HelloWorld2|) <|HelloWorld2$Point.this$0|>
      @AccessFlag (FINAL,SYNTHETIC);
}

  procedure (|void|) {|org.sireum.test.jvm.samples.HelloWorld2$Point.<init>(
      Lorg.sireum.test.jvm.samples.HelloWorld2;II)V|} ((|org.sireum.test.jvm.
      samples.HelloWorld2$Point|) [|this|], (|org.sireum.test.jvm.samples.
      HelloWorld2|) [|v1|], (|int|) [|x|], (|int|) [|y|])
    @MaxLocals 4
    @MaxStack 2
    @Owner (|org.sireum.test.jvm.samples.HelloWorld2$Point|)
    @Access (CONSTRUCTOR)
    @Desc "org.sireum.test.jvm.samples.HelloWorld2$Point.<init>(Lorg.sireum.
        test.jvm.samples.HelloWorld2;II)V"
  {
    local jmp;
    s0;

    #L00000Aa. [|this|].<|org.sireum.test.jvm.samples.HelloWorld2$Point.
        this$0|> := [|v1|] @Type (|org.sireum.test.jvm.samples.HelloWorld2|);
    #L00000Ab. call s0 := {|java.lang.Object.<init>()V|}([|this|])
        @ClassDescriptor (|java.lang.Object|) @Type special;
    #L00001Aa. [|this|].<|org.sireum.test.jvm.samples.HelloWorld2$Point.x|>
        := [|x|] @Type (|int|);
    #L00002Aa. [|this|].<|org.sireum.test.jvm.samples.HelloWorld2$Point.y|>
        := [|y|] @Type (|int|);
    #L00003Aa. return @void;
  }
  procedure (|void|) {|org.sireum.test.jvm.samples.HelloWorld2$Point.print()V
      |} ((|org.sireum.test.jvm.samples.HelloWorld2$Point|) [|this|])
```

```
    @MaxLocals 1
    @MaxStack 2
    @Owner (|org.sireum.test.jvm.samples.HelloWorld2$Point|)
    @Access ()
    @Desc "org.sireum.test.jvm.samples.HelloWorld2$Point.print()V"
  {
    local jmp;
    s0;
    s1;

    #L00000Aa. s0 := +|java.lang.System.out|+ @Type (|java.io.PrintStream|);
    #L00000Ab. call s1 := {|java.io.PrintStream.println(I)V|}(s0,9)
        @ClassDescriptor (|java.io.PrintStream|) @Type virtual;
    #L00001Aa. return @void;
  }
```

Appendix B

User Manual

B.1 Setting up Development Environment

1. Download and install Sireum by following insutrctions at http://sireum.org/download.

2. Lauch Sireum Development Environment using

   sireum launch sireumdev

   Select any directory for workspace when asked. You need to run Scala diagonistics; enable JDT Weaving for Scala IDE and then restart the development environment.

3. Add "SIREUM_HOME/apps/platform/java" in Eclipse's Java Installed JREs preference page, and make it the default. You also need to set Eclipse's Java compiler compilance level to 1.7. Note: You might get a warning saying "Subversion Native Library Not Available". You can correct this by setting SVN Client interface to "SVNKit" in Eclipse's Team-¿SVN preference page.

4. Import all projects in

   - Clone Sireum Prelude Repo

     git clone https://github.com/sireum/prelude.git

   - Clone Sireum Pilar Parser

git clone https://github.com/sireum/parser.git

- Clone Sireum Core Repo

  git clone https://github.com/sireum/core.git

- Clone Sireum JVM

  git clone https://github.com/sireum/jvm.git

5. Run the project sireum-core-test to test if Sireum core has been imported properly.

6. Now, run sireum-jvm-test to test if Sireum JVM has been imported properly.

B.2 Sireum JVM Command Line manual

Sireum JVM is part of Sireum tools cmd.

```
sireum tools jvm
Usage:
  sireum tools jvm [options] <classes>

  where the available options are:

  -h | --help
  -d | --directory Output Directory [Default: "(current direcory)"]

Example usage :
  sireum tools jvm scala.collection
  sireum tools jvm ./Strings.class
```