A Survivor's Guide to Java Program Analysis with Soot

Árni Einarsson and Janus Dam Nielsen
BRICS, Department of Computer Science
University of Aarhus, Denmark
`{arni,jdn}@brics.dk`

`Version 1.1`

07/17/2008

Abstract

These notes are written to provide our own documentation for the Soot framework from McGill University. They focus exclusively on the parts of Soot that we have used in various projects: parsing class files, performing points-to and null pointer analyses, performing data-flow analysis, and extracting abstract control-flow graphs. The notes also contain the important code snippets that make everything work since it is our experience, that the full Soot API leaves novice users in a state of shock and awe.

Contents

Acknowledgments

The authors would like to thank the participants of the Soot seminars held during the spring of 2006 at BRICS for their many comments and ideas, and for reading drafts of this note: Claus Brabrand, Aske Simon Christensen, Martin Mosegaard Jensen, Christian Kirkegaard, Anders Møller and Michael I. Schwartzbach. Furthermore we want to thank Ondřej Lhoták for readily answering our Paddle related questions and helping us get Paddle to work.

1 Introduction

This guide provides detailed descriptions and instructions on the use of Soot, a Java optimization framework [13]. More specifically on how we have used and are using Soot in various projects involving different forms of analysis. Soot is a large framework which can be quite challenging to navigate without getting quickly lost. With this guide, we hope to provide the insight necessary to make that navigation a little more comfortable. The reader is assumed to be familiar with basic static analysis on a level similar to [12] and a firm command of the Java programming language and Java bytecode.

Soot is a product of the Sable research group from McGill University, whose objective is to provide tools leading to the better understanding and faster execution of Java programs. The Soot website is at `http://www.sable.mcgill.ca/soot/`.

One of the main benefits of Soot is that it provides four different *Intermediate Representations* (IR) for analysis purposes. Each of the IRs have different levels of abstraction that give different benefits when analyzing, they are: Baf, Grimp, Jimple and Shimple.

Soot builds data structures to represent:

Scene. The `Scene` class represents the complete environment the analysis takes place in. Through it, you can set e.g., the application classes(The classes supplied to Soot for analysis), the main class (the one that contains the main method) and access information regarding interprocedural analysis (e.g., points-to information and call graphs).

SootClass. Represents a single class loaded into Soot or created using Soot.

SootMethod. Represents a single method of a class.

SootField. Represents a member field of a class.

Body. Represents a method body and comes in different flavors, corresponding to different IRs (e.g., `JimpleBody`).

These data structures are implemented using Object-Oriented techniques, and designed to be easy to use and generic where possible.

This guide is a practitioners survival guide so it is best read sitting in front of a computer with the source code for each section loaded into your favorite text editor and running the examples as we go along. There's no substitute for hands-on practice.

1.1 Setting up Soot

Soot is available to download from the Sable group's website: `http://www.sable.mcgill.ca/soot/soot_download.html`. The easiest and fastest way is to get the pre-compiled Jars, you will need all of sootclasses, jasminclasses and polyglotclasses. During the compilation of this guide we have used Soot version 2.2.2, but we have not tested any newer release. [1]

To use Soot you need the soot, jasmin, and polyglot jars to be on the classpath. To test the setup, try executing:

```
java -cp jasminclasses-2.3.0.jar:polyglot.jar:sootclasses-2.3.0.jar:.
                              soot.Main --help
```

at the command line and you should receive instructions on how to use the tool.

Note that you need at least JDK 1.3 to use Soot and at least JDK 1.4 to use the Eclipse plugin. The newer releases has some support for JDK 1.5 but we have not tested that.

Developing with Soot in Eclipse

To develop using Soot in Eclipse, you start by creating an empty project. Then you need to add the three Jars as libraries to your project. To do this, right click on your project and select *Properties*. From the tree on the right, select *Java Build Path* and from there the *Libraries* tab. Select *Add External Jars*, navigate to where Jars are located and select the first Jar. Click *OK*. Repeat this for the other two Jars.

Setting up the Soot Eclipse plugin

For instruction on how to set up the Eclipse plugin, refer to `http://www.sable.mcgill.ca/soot/eclipse/updates/`.

1.2 Road-map to this guide

This Guide is best served when read in the order it is presented. However, to briefly prepare the reader for what he/she is about to read, we present the following road-map.

Internal Representations describes the four IRs in Soot: Baf, Jimple, Shimple and Grimp, in some detail.

Basic Soot Constructs describes briefly the basic objects that constitute a method body.

Soot as a stand-alone tool describes how to use Soot as an isolated tool. To that end, this section describes the inner workings of Soot, using Soot at the command-line and the various options it accepts, some of the built-in analyses it provides and how to extend the tool with user-defined analyses.

The Data-Flow Framework describes in detail how to utilize the power of the data-flow framework within Soot. It is accompanied by a complete example implementation of a very-busy expressions analysis. This section

[1] The call-graph example has been updated to Soot version 2.3.0.

includes a description of how to tag code for the Eclipse plugin to present results visually.

Call Graph Construction describes how to access a call graph during a whole-program analysis and use it to extract various information.

Points-To analysis describes how to set up and use two of the more advanced frameworks for doing points-to analysis in Soot: SPARK and Paddle.

Extracting Abstract Control-Flow Graphs describes how to use Soot to extract a custom IR of an abstract control-flow graph to be used as a starting point for an analysis which may benefit from the simplifications made during the abstraction, like the Java String Analysis[3].

Furthermore all examples used in this note are complete and can be obtained at `http://www.brics.dk/SootNote/`.

2 Basic Soot Constructs

In this section we describe the basic objects used commonly throughout the use of Soot. More specifically, we focus on the objects that make up the code of a method. These are fairly brief descriptions due to the fact that these are very simple constructs.

2.1 Method bodies

The Soot class `Body` represents a single method body, it comes in different flavors for each of the IR s used to represent the code. These are:

- `BafBody`
- `GrimpBody`
- `ShimpleBody`
- `JimpleBody`

We can use a `Body` to access various information, most notably we can retrieve a `Collection` (Soot uses its own implementation of a `Collection`, called `Chain`) of the locals declared (`getLocals()`), the statements which constitute the body (`getUnits()`) and all exceptions handled in the body (`getTraps()`).

2.2 Statements

A statement in Soot is represented by the interface `Unit`, of which there are different implementations for different IRs — e.g., Jimple uses `Stmt` while Grimp uses `Inst`.

Through a `Unit` we can retrieve values used (`getUseBoxes()`), values defined (`getDefBoxes()`) or even both (`getUseAndDefBoxes()`). Additionally, we can get at the units jumping to this unit (`getBoxesPointingToThis()`) and units this unit is jumping to (`getUnitBoxes()`) — i.e., by jumping we mean control flow other than falling through. Unit also provides various methods of querying about branching behavior, such as `fallsThrough()` and `branches()`.

Values

A single datum is represented as a `Value`. Examples of values are: locals (`Local`), constants (in Jimple `Constant`), expressions (in Jimple `Expr`), and many more. An expression has various implementations, e.g. `BinopExpr` and `InvokeExpr`, but in general can be thought of as carrying out some action on one or more `Value`s and returns another `Value`.

References

References in Soot are called `boxes`, of which there are two different types: `ValueBox` and `UnitBox`.

UnitBoxes refer to `Unit`s. Used when a single unit can have multiple successors, i.e. when branching.

ValueBoxes refer to `Value`s. As previously described, each unit has a notion of values used and defined in it, this can be very useful for replacing use or def boxes in units, for instance when performing constant folding.

3 Intermediate Representations

The Soot framework provides four intermediate representations for code: Baf, Jimple, Shimple and Grimp. The representations provide different levels of abstraction on the represented code and are targeted at different uses e.g., baf is a bytecode representation resembling the Java bytecode and Jimple is a stackless, typed 3-address code suitable for most analyses. In this section we will give a detailed description of the Jimple representation and a short description of the other representations.

3.1 Baf

Baf is a streamlined stack-based representation of bytecode. Used to inspect Java bytecode as stack code, but abstracts away the constant pool and abstracts type dependent variations of instructions to a single instruction (e.g. in Java bytecode there are a number of instructions for adding integers, longs, etc. in Baf they have all been abstracted into a single instruction for addition). Instructions in Baf correspond to Soot Units and so all implementations of instructions implement the *Inst* interface which implements the *Unit* and *Switchable* interfaces.

The implementation of the Baf representation resides in the soot.baf and soot.baf.internal packages and the very curious reader is encouraged to investigate these packages, but be aware that there is no documentation of the individual classes.

Baf is useful for bytecode based analyses, optimizations and transformations, like peephole optimizations.

Optimizations available as part of the Soot framework based on the Baf representation can be found in the package soot.baf.toolkits.base.

3.2 Jimple

Jimple is the principal representation in Soot. The Jimple representation is a typed, 3-address, statement based intermediate representation.

Jimple representations can be created directly in Soot or based on Java source code(up to and including Java 1.4) and Java bytecode/Java class files(up to and including Java 5).

The translation from bytecode to Jimple is performed using a naïve translation from bytecode to untyped Jimple, by introducing new local variables for implicit stack locations and using subroutine elimination to remove jsr instructions. Types are inferred for the local variables in the untyped Jimple and added [4]. The Jimple code is cleaned for redundant code like unused variables or assignments. An important step in the transformation to Jimple is the linearization (and naming) of expressions so statements only reference at most 3 local variables or constants. Resulting in a more regular and very convenient representation for performing optimizations. In Jimple an analysis only has to handle the 15 statements in the Jimple representation compared to the more than 200 possible instructions in Java bytecode.

In Jimple, statements correspond to Soot Units and can be used as such. Jimple has 15 statements, the core statements are: NopStmt, IdentityStmt and AssignStmt. Statements for intraprocedural control-flow: IfStmt, GotoStmt, TableSwitchStmt(corresponds to the JVM tableswitch instruction) and LookupSwitchStmt(corresponds to the JVM lookupswitch instruction). Statements for interprocedural control-flow: InvokeStmt, ReturnStmt and ReturnVoidStmt. Monitor statements: EnterMonitorStmt and ExitMonitorStmt. The last two are: ThrowStmt, RetStmt (return from a JSR, not created when making Jimple from byte code).

As an example lets generate Jimple code for the following class:

```
public class Foo {

  public static void main(String[] args) {
    Foo f = new Foo();
    int a = 7;
    int b = 14;
    int x = (f.bar(21) + a) * b;
  }

  public int bar(int n) { return n + 42; }
}
```

Running Soot using the command *java soot.Main -f J Foo* yields the file Foo.Jimple in the directory sootOutput also shown below - for more information on how to run Soot from the command line or Eclipse see Section 4.

```
public static void main(java.lang.String[]) {
  java.lang.String[] r0;
  Foo $r1, r2;
  int i0, i1, i2, $i3, $i4;

  r0 := @parameter0: java.lang.String[];
  $r1 = new Foo;
  specialinvoke $r1.<Foo: void <init>()>();
  r2 = $r1;
  i0 = 7;
  i1 = 14;
  // InvokeStmt
  $i3 = virtualinvoke r2.<Foo: int bar()>(21);
  $i4 = $i3 + i0;
  i2 = $i4 * i1;
  return;
}

public int bar() {
  Foo r0;
  int i0, $i1;
  r0 := @this: Foo; // IdentityStmt
  i0 := @parameter0: int; // IdentityStmt
  $i1 = i0 + 21; // AssignStmt
  return \$i1; // ReturnStmt
}
```

In the code fragment above we see the Jimple code generated for the main and bar methods. Jimple is a hybrid between Java source code and Java byte code. We recognize the statement-based structure from Java with declarations of local variables and assignments, but the control flow and method invocation style is similar to the one in Java bytecode. The local variables which start with a `$` sign represent stack positions and not local variables in the original program whereas those without `$` represent real local variables e.g. `i0` in the `main` method corresponds to `a` in the Java source.

The linearization process has split up the statement `int x = (f.bar(21) + a) * b` into the three statements `$i4 = $i3 + i0` and `i2 = $i4 * i1` and thus enforced the 3-address form.

In Jimple, parameter values and the `this` reference are assigned to local variables using IdentityStmt's e.g. the statements `i0 := @parameter0: int;` and `r0 := @this: Foo` in the `bar` method. By using IdentityStmt's it is ensured that all local variables have at least one definition point and so it becomes explicit in the code where `this` in `this.m();` is defined.

All the local variables are typed. The type information can be used with great advantage during analysis.

Jimple is a very good foundation for most analyses which do not need the explicit control flow and SSA form of Shimple. The versatility of the Jimple representation is best illustrated by the many built-in analyses provided as part of the Soot framework.

Be aware that Jimple is not Java source, especially the introduction of new unique variables can result in great difference between result and expectations when you compare the Java source code to the produced Jimple code.

The Jimple intermediate representation is available in the packages soot.jimple, soot.jimple.internal and an extensive collection of optimizations are available in soot.jimple.toolkits.* especially soot.jimple.toolkits.scalar and soot.jimple.-toolkits.annotation.*.

3.3 Shimple

The Shimple intermediate representation is a Static Single Assignment-form version of the Jimple representation. SSA-form guarantees that each local variable has a single static point of definition which significantly simplifies a number of analyses.

Shimple is almost identical to Jimple with the two differences of the single static point of definition and the so-called phi-nodes, and so Shimple can be treated almost in the same way as Jimple.

As an example we use the `ShimpleExample` class with the test method shown below[2]:

```
public int test() {
  int x = 100;
```

[2]The example is based on a similar example found on the Soot homepage

```
  while(as_long_as_it_takes) {
    if(x < 200)
        x = 100;
    else
        x = 200;
  }
  return x;
}
```

Producing Jimple based on the `ShimpleExample` class using `java soot.Main -f jimple ShimpleExample` yields the following Jimple code:

```
public int test() {
    ShimpleExample r0;
    int i0;
    boolean $z0;

    r0 := @this: ShimpleExample;
    i0 = 100;

  label0:
    $z0 = r0.<ShimpleExample: boolean as_long_as_it_takes>;
    if $z0 == 0 goto label2;

    if i0 >= 200 goto label1;

    i0 = 100;
    goto label0;

  label1:
    i0 = 200;
    goto label0;

  label2:
    return i0;
}
```

Where we see three assignments to the variable `i0` which violates the static single assignment form.

If we produce the corresponding Shimple code using `java soot.Main -f shimple ShimpleExample` we get:

```
public int test() {
    ShimpleExample r0;
    int i0, i0_1, i0_2, i0_3;
    boolean $z0;
```

```
    r0 := @this: ShimpleExample;
(0) i0 = 100;

  label0:
    i0_1 = Phi(i0 #0, i0_2 #1, i0_3 #2);
    $z0 = r0.<ShimpleExample: boolean as_long_as_it_takes>;
    if $z0 == 0 goto label2;

    if i0_1 >= 200 goto label1;

    i0_2 = 100;
(1) goto label0;

  label1:
    i0_3 = 200;
(2) goto label0;

  label2:
    return i0_1;
}
```

Which is identical to the Jimple code except for two things. The introduction of a `Phi`-node and the variable `i0` has been spilt into four variables `i0`, `i0_1`, `i0_2`, and `i0_3`.

Phi-nodes are needed in SSA-form because the value of `i0_1` depends on the path taken in the control flow graph. The value may arrive from either `(0)`, `(1)`, or `(2)`. The `Phi`-node can be seen as a function which returns the value of `i0` if the flow arrives from `(0)`, the value of `i0_2` if the flow arrives from `(1)` or the value of `i0_3` if the flow arrives from `(2)`. The paper [2] is a good reference on SSA-form.

Shimple encodes control-flow explicitly and so we can easily make control-flow sensitive analysis on Shimple code. The careful reader will have noticed that `x` is constant in the above example so the `test` method could have been constant folded to a single return statement since most of the control structures are unnecessary.

To illustrate the differences between the Shimple representation and the Jimple representation let's optimize the program based on each representation and compare the outcome. If we run the Jimple constant propagator and folder on the `ShimpleExample` class - just to make the point clear we apply every available optimization in the Soot arsenal: `java soot.Main -f jimple -O ShimpleExample` which yields the following result:

```
public int test() {
    ShimpleExample r0;
```

```
    int i0;
    boolean $z0;

    r0 := @this: ShimpleExample;
    i0 = 100;

  label0:
    $z0 = r0.<ShimpleExample: boolean as_long_as_it_takes>;
    if $z0 == 0 goto label2;

    if i0 >= 200 goto label1;

    i0 = 100;
    goto label0;

  label1:
    i0 = 200;
    goto label0;

  label2:
    return i0;
}
```

Which is exactly equal to the output we saw earlier, when running Soot with no optimization! Based on the Jimple representation the optimizations are not able to deduce that `x` is constant.

Running the Shimple constant propagator and folder `java soot.Main -f jimple -via-shimple -O ShimpleExample` yields:

```
public int test() {
    ShimpleExample r0;
    boolean $z0;

    r0 := @this: ShimpleExample;

  label0:
    $z0 = r0.<ShimpleExample: boolean as_long_as_it_takes>;
    if $z0 == 0 goto label1;

    goto label0;

  label1:
    return 100;
}
```

Which is what we expected. Since the field variable `as_long_as_it_takes` is non-static the while loop can not be completely removed, but as we see the conditional assignments have been removed since the optimizer deduced that `x` is constant and so the phi-node chose between three identical values and got optimized away.

Conclusion: Shimple exposes the control structure explicitly and variables only have a static single assignment.

The Shimple intermediate representation is available in the packages soot.-shimple, soot.shimple.internal and a constant-folder is available in soot.shimple.-toolkits.scalar.

3.4 Grimp

Grimp is similar to Jimple, but allows trees of expressions together with a representation of the `new` operator — in this respect Grimp is closer to resembling Java source code than Jimple is and so is easier to read and hence the best intermediate representation for inspecting disassembled code by a human reader.

As an example of support for trees of expressions we run the Foo example through Soot in order to produce Grimp code using the command *java soot.Main -f G Foo* which yields the file Foo.grimple in the directory sootOutput. Below we show the `main` method from the Foo.grimple files:

```
public static void main(java.lang.String[]) {
  java.lang.String[] r0;
  Foo r2;
  int i0, i1, i2;

  r0 := @parameter0: java.lang.String[];
  r2 = new Foo();
  i0 = 7;
  i1 = 14;
  i2 = (r2.<Foo: int bar(int)>(21) + i0) * i1;
  return;
}
```

There are three very clear differences between the two `main` methods. One, expression trees are not linearized. Two, object instantiation and constructor call has been collapsed into the `new` operator. Three, since expression trees are not linearized new temporary locals(locals starting with `$`) are not created, but we do need new temporary locals in connection with e.g. `while`.

The Grimp representation is good for some kinds of analyses like available expressions, if you want complex expressions as well as simple expressions. Grimp is also a good starting point for decompilation.

The Grimp intermediate representation is available in the packages soot.grimp, soot.grimp.internal and some optimizations are available in soot.grimp.toolkits.-base.

4 Soot as a stand-alone tool

Most of the information contained within this section is a summation from `http://www.sable.mcgill.ca/soot/tutorial/usage/`. We wish to stress the fact that said information is by far not a complete list, but rather a compilation of features we have found especially useful.

Soot can be used as a stand-alone tool for many different purposes e.g., applying some of the built-in analyses or transformations to your own code. The Soot tool can be executed either using the command line or the Eclipse plug-in. The many different uses are reflected in the huge number of configuration options available. This section will describe how to use Soot as a stand-alone tool, how the options are grouped according to their use and some of the most often used options are described in detail (further descriptions can be found at the previously mentioned URL).

Soot can be invoked from the command line as follows, if Soot is included in your classpath:

```
java [javaOptions] soot.Main [sootOptions] classname
```

where `[sootOptions]` represent the various options Soot accepts and `classname` is the class to analyze. Soot can also be run through the Eclipse plugin from either the Project menu or the Navigator pop-up menu and choose Soot → Process (all) source file(s) → Run. The Eclipse plug-in provides a GUI interface to all the options that Soot accepts.

To get a list of options supported by Soot, run:

```
java soot.Main -h
```

from the command-line.

Class categorization. In Soot we distinguish between three kinds of classes: argument classes, application classes, and library classes.

The argument classes are the classes you specify to Soot. Using the command-line tool they would be the classes listed explicitly or those found in a directory given by the -process-dir option. In Eclipse the argument classes are the selected classes if you access the Soot plug-in using the Navigator pop-up menu, or the classes of the entire project if you access the Soot plug-in using the Project menu. All argument classes are also application classes.

The application classes are the classes to be analyzed or transformed by Soot, and turned into output.

Library classes are those classes that are referred to by application classes but are not application classes. They are used in the analyses and transformations but are not themselves transformed or output.

There are also two modes that affect the behavior of how classes are categorized: application mode and non-application mode. In application mode all classes referred to by the argument classes become application classes themselves, excluding class from the Java runtime system. In non-application mode those same classes would be library classes.

Soot provides further options to influence which classes are application classes in application mode.

-i PKG, -include PKG Those classes in packages whose names begin with PKG will be treated as application classes.

-x PKG, -exclude PKG Those classes in packages whose names begin with PKG will be treated as library classes.

-include-all All classes referred to by any argument classes will be treated as application classes.

Several other options to control this behavior are available.

Input options. Soot provides several option to control how input to Soot is handled, we will describe the most relevant.

-cp PATH, -soot-class-path PATH, -soot-classpath PATH Sets PATH as the classpath to search for classes.

-process-dir DIR Sets all the classes found in DIR as argument classes.

-src-prec FORMAT Sets the precedence of source files to use. Valid FORMAT strings are: *c* (or class) to prefer class files (the default); *J* (or jimple) to prefer Jimple files; *java* to prefer java files.

Output options. The output options control what to actually output from Soot and then in what format. Classes that have been categorized as application classes will be output as class files by default. This can be overridden by specifying a value to the output format option (-f or -output-format). A format exists for each of the intermediate representations and their abbreviated format — e.g. to output Jimple code specify -f J or -f jimple. For a complete list of the accepted formats refer to the previously mentioned URL.

Other output options worth mentioning are:

-d DIR, -output-dir DIR Specify the folder DIR to store output files (the default is sootOutput).

-xml-attributes Save all tags to an XML file. This is used by the Eclipse plug-in to visually convey the results of an analysis.

-outjar, -output-jar Save all output in a JAR file instead of in a directory.

4.1 Soot phases

The execution of Soot is separated into several phases called packs. The first step is to produce Jimple code to be fed into the various packs. This is done by parsing class, jimple or java files and then passing their result through the *Jimple Body* (jb) pack.

The pack naming scheme is fairly simple. The first letter designates which IR the pack accepts; s for Shimple, j for Jimple, b for Baf and g for Grimp. The second letter designates the role of the pack; b for body creation, t for user-defined transformations, o for optimizations and a for attribute generation (annotation). A p at the end of the pack name stands for "pack". For instance the jap (Jimple annotations pack) contains all the built in intra-procedural analyses.

The packs of special interest are those that allow user-defined transformations: jtp (Jimple transformation pack) and stp (Shimple transformation pack). Any user defined transformations (e.g. tagging information from analyses) can be injected into these packs and they will then be included in the execution of Soot[3]. The execution flow through packs is best described in Figure 1. Each application class is processed through a path in this execution flow but they don't have access to any information generated from the processing of other application classes.

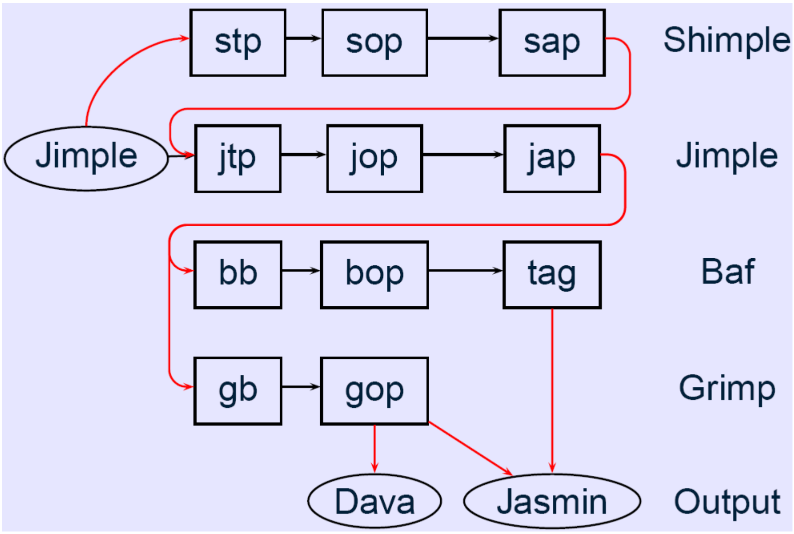

Figure 1: Intra-procedural execution flow. Image taken from [5].

Inter-procedural analysis With inter-procedural analyses, the execution flow is a little different. When conducting an inter-procedural analysis in Soot

[3]These packs are intended for intra-procedural analyses.

we need to put Soot into *Whole-program mode* and we do that by specifying the option *-w* to Soot. In this mode Soot includes three other packs in the execution cycle: cg (call-graph generation), wjtp (whole Jimple transformation pack, and wjap (whole Jimple annotation pack). Additionally, to add whole-program optimizations (e.g. static inlining) we specify the option *-W* which further adds the wjop (whole Jimple optimization pack) into the mix. The difference between these packs and the intra-procedural ones is that the information generated in these packs are made available to the rest of Soot through the Scene — i.e., the same information is available for each application class being processed. See Figure 2 for a visual representation of the execution flow.

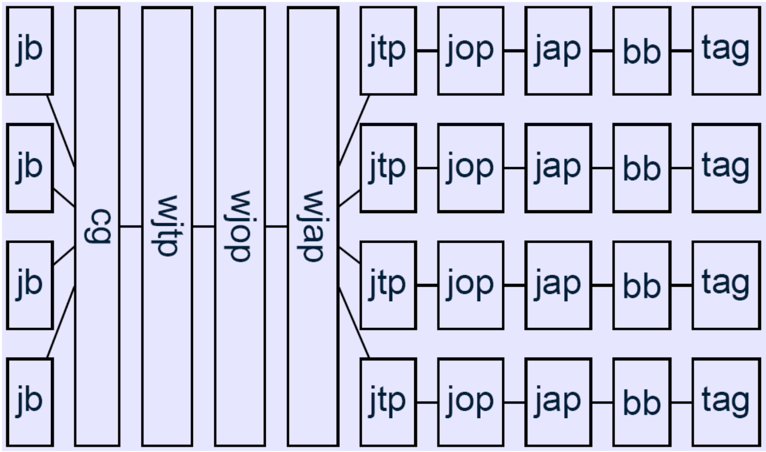

Figure 2: Inter-procedural execution flow. Image taken from [5].

Phase options. To produce a list of all available packs in Soot, execute the command:

```
java soot.Main -pl
```

from the command-line. This information can be used to get further help on what options are available for the different packs and the operations they contain (e.g. built in analyses). To list help and available options for a pack run Soot like this:

```
java soot.Main -ph PACK
```

where `PACK` is one of the pack names listed from running Soot with `-pl`.

To set an option to a pack you specify the -p option followed by the pack name and a key-value pair of the form *OPT:VAL*, where OPT is the option you want to set and VAL is the value you want to set it to. For example, to turn off all user-defined intra-procedural transformations you do:

```
java soot.Main -p jtp enabled:false MyClass
```

where `MyClass` is the class you wish analyzed.

4.2 Off-The-Shelf Analyses

Soot includes several example analyses to demonstrate its capabilities. This section describes how to run a few of these analyses from the command-line and using the Eclipse plug-in.

Null Pointer Analysis. The built-in null pointer analysis is located in the jap pack and is furthermore split up into two separate entities: the null pointer checker and the null pointer colorer. The former finds instructions which have the potential to throw `NullPointerException` while the latter uses that information to add tagging information for the Eclipse plug-in. To run the null pointer colorer to produce some visualization of nullness within our program we can do:

```
java soot.Main -xml-attributes -f J -p jap.npcolorer on MyClass
```

which will produce a Jimple file. When this file is viewed in Eclipse, reference types will be color coded according to their nullness (green for definitely not null, blue for unknown and red for definitely null).

To run the same analysis to produce color codes for the source file from Eclipse, right-click the class you want to analyze, navigate to Soot → Process Source File and click Run Soot... . This will bring up the Soot options dialog. First click Output Options in the tree on the left and select Jimple File as the desired Output Format from the options on the right. Next expand the Phase Options tree, expand the Jimple Annotation Pack and select Null Pointer Colorer. Check the box next to Enabled. Press Run. Now open the source file and you should get something like depicted in Figure 3.

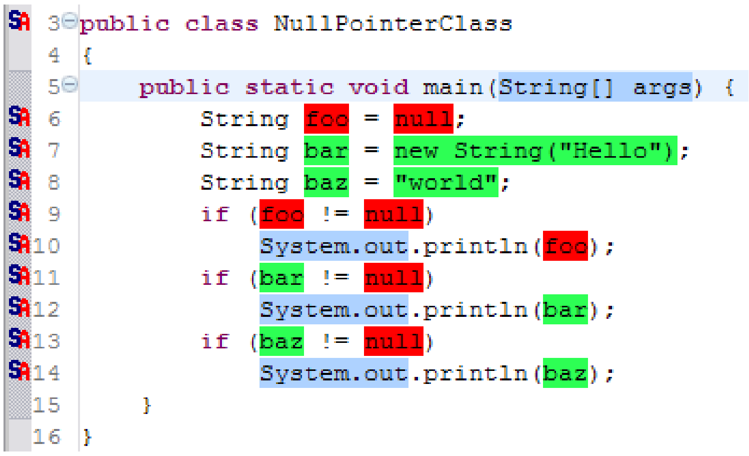

Figure 3: Null pointer analysis visualized.

Array Bounds Analysis. Another good example of a built-in analysis, is the array bounds checker. The analysis checks whether array bounds might be violated. This kind of analysis could enable the compiler to perform optimization by not inserting explicit array bounds checks in the bytecode. This analysis is also located in the jap pack under jap.abc. The simplest way to run it to produce visualizations is:

```
java soot.Main -xml-attributes -f J -p jap.abc on -p jap.abc
                  add-color-tags:true MyClass
```

The results of the analysis indicate for both the upper bound and lower bound, whether there's a potentially unsafe access or the access is guaranteed to be safe.

Liveness Analysis. For the final example, let's look at liveness analysis. The built in liveness analysis colors variables that are definitely live out of a statement. It has only one option, enabled or not. To run it:

```
java soot.Main -xml-attributes -f J -p jap.lvtagger on MyClass
```

The results indicate all live variables out of a statement and furthermore, variables that are either used or defined in a statement and are live out of it get colored green.

4.3 Extending Soot's Main class

After having designed and implemented an analysis, we might need to be able to use that in conjunction with other features (e.g. built-in analyses) from Soot. To do this we need to extend Soot's Main class to include our own analysis. Note that this is not an extension in the Java meaning, but rather an injection of an intermediate step where our analysis is put into Soot. In other words we want Soot to run our analysis and still process all other options we might want to pass to it.

How this is done depends on whether the analysis being injected is an inter- or intra-procedural analysis. The former needs to be injected into the *wjtp* phase while the latter goes into the *jtp* phase. The following code example shows how to inject an instance of the hypothecial class `MyAnalysisTagger` (which performs some intraprocedural analysis) into Soot.

```
public class MySootMainExtension
{
  public static void main(String[] args) {
    // Inject the analysis tagger into Soot
    PackManager.v().getPack("jtp").add(new
           Transform("jpt.myanalysistagger",
                     MyAnalysisTagger.instance()));
    // Invoke soot.Main with arguments given
    Main.main(args);
  }
}
```

5 The Data-Flow Framework

In general we can think of designing a flow analysis as a four step procedure.

1. Decide what the nature of the analysis is. Is it a backwards or forwards flow analysis? Do we need to consider branching specially, or not?
2. Decide what is the intended approximation. Is it a may or a must analysis? In effect, you are deciding whether to union or intersect when merging information flowing through a node.
3. Performing the actual flow; essentially writing equations for each kind of statement in the intermediate representation — e.g. how should assignment statements be handled?
4. Decide the initial state or approximation of the entry node (exit node if it is a backwards flow) and inner nodes — generally the empty set or the full set, depending on how conservative the analysis will be.

Performing data-flow analysis we need some sort of structure representing how data flows through a program, such as a control flow graph (cfg). The Soot data-flow framework is designed to handle any form of cfg implementing the interface `soot.toolkits.graph.DirectedGraph`.

For instructional purposes we will use a very-busy expressions analysis as a running example. The full code for the examples can be found in the source.

5.1 Step 1: Nature of the analysis

Soot provides three different implementations of analyses: `ForwardFlowAnalysis`, `BackwardFlowAnalysis` and `ForwardBranchedFlowAnalysis`. The first two are the same except for flow direction, the result of which are two maps: from nodes to IN sets and from nodes to OUT sets. The last one provides the ability to propagate different information through each of the branches of a branching node — e.g., the information flowing out of a node containing the statement `if(x>0)` can be $x > 0$ to one branch and $x \leq 0$ to the other. Thus the results of that analysis are three maps: from nodes to IN sets, from nodes to fall-through OUT sets and from nodes to branch OUT sets. All of these provide an implementation of the fixed-point mechanism using a worklist algorithm. If you want to implement this in some other way you can extend one of the abstract super-classes: `AbstractFlowAnalysis` (the top one), `FlowAnalysis` or `BranchedFlowAnalysis`. Otherwise, the way to plug your specific analysis into the framework is to extend one of the first three classes.

In the case of very-busy expressions, we need a backwards flowing analysis, so our class signature will be:

```
class VeryBusyExpressionAnalysis extends BackwardFlowAnalysis
```

Now, in order to utilize the functionality from the framework we need to provide a constructor. In this constructor we need to do two things: (1) call the super's constructor and (2) invoke the fixed-point mechanism. This is accomplished like this:

```
public VeryBusyExpressionAnalysis(DirectedGraph g) {
  super(g);
  doAnalysis();
}
```

For information regarding `DirectedGraph` and other control flow graphs provided by Soot, refer to Section 5.7.

5.2 Step 2: Approximation level

The approximation level of an analysis is decided by how the analysis performs JOINs of lattice elements. Generally, an analysis is either a *may* or a *must* analysis. In a may analysis we want to join elements using union, and in a must analysis we want to join elements using intersection. In the flow analysis framework joining is performed in the `merge` method. Very-busy expression analysis is a must analysis so we use intersection to join:

```
protected void merge(Object in1, Object in2, Object out) {
  FlowSet inSet1 = (FlowSet)in1,
          inSet2 = (FlowSet)in2,
          outSet = (FlowSet)out;
  inSet1.intersection(inSet2, outSet);
}
```

As can be seen from this, the flow analysis framework is designed with such abstraction that it doesn't assume anything about how the lattice element is represented. In our case we use the notion of a `FlowSet`, described in detail in Section 5.6. Because of this abstraction we also need to provide a way of copying the contents of one lattice element to another:

```
protected void copy(Object source, Object dest) {
  FlowSet srcSet = (FlowSet)source,
          destSet = (FlowSet)dest;
  srcSet.copy(destSet);
}
```

5.3 Step 3: Performing flow

This is where the real work of the analysis happens, the actual flowing of information through nodes in the cfg. The framework method involved is `flowThrough`. We can think of this process as having two parts: (1) we need to move information from the IN set to the OUT set, excluding the information that the node *kills*; and (2) we need to add information to the OUT set that the node *generates*. In the case of very-busy expressions, the node kills expressions containing references to locals that are defined in the node. Furthermore, it generates expressions that are used in the node.

```
protected void flowThrough(Object in, Object node, Object out) {
  FlowSet inSet = (FlowSet)source,
          outSet = (FlowSet)dest;
  Unit u = (Unit)node;
  kill(inSet, u, outSet);
  gen(outSet, u);
}
```

The `kill` and `gen` methods are not part of the framework, but rather user-defined methods. For the actual implementation of these methods, refer to the example source code; better yet, try implementing them on your own first.

5.4 Step 4: Initial state

This step involves deciding the initial contents of the lattice element for the entry point, and of the lattice elements of all the other points. In the flow analysis framework, this is achieved by overriding two methods: `entryInitialFlow` and `newInitialFlow`. In the case of very-busy expressions, the entry point is the last statement (the exit point) and we want it to be initialized with the empty set. As for other lattice points, we also want them initialized with the empty set.

```
protected Object entryInitialFlow() {
  return new ValueArraySparseSet();
}

protected Object newInitialFlow() {
  return new ValueArraySparseSet();
}
```

Note that `ValueArraySparseSet` is not a Soot construct, but rather our own specialization of `ArraySparseSet`. For more information refer to Section 5.6.

5.5 Limitations

With an analysis such as very-busy expressions analysis, we need to keep in mind what is actually being analyzed. In our case, we are analyzing Jimple code and being a three-address code, compound expressions will be broken up into intermediate parts (e.g. $a+b+c$ becomes $temp = a+b$ and $temp+c$). This brings us to the realization that our particular analysis, without modification, can only analyze a fraction of possible expressions in the original source code. In this particular case we could analyze Grimp code instead, and consider compound expressions specially.

5.6 Flow sets

In Soot, flow sets represent data associated with a node in the control-flow graph (e.g. for busy expressions, a node's flow set is a set of expressions busy at that node).

There are two different notions of a flow set, bounded (the interface `BoundedFlowSet`) and unbounded (the interface `FlowSet`). A bounded set is one that knows its universe of possible values, while unbounded is the opposite.

Classes implementing the `FlowSet` interface need to implement (among others) the methods:

- `clone()`
- `clear()`
- `isEmpty()`
- `copy(FlowSet dest) // deep copy of this into dest`
- `union(FlowSet other, FlowSet dest) // dest <- this ∪ other`
- `intersection(FlowSet other, FlowSet dest) // dest <- this ∩ other`
- `difference(FlowSet other, FlowSet dest) // dest <- this - other`

These operations are enough to make a flow set a valid lattice element.

In addition, when implementing `BoundedFlowSet`, it needs to provide methods for producing the set's complement and its topped set (i.e., a lattice element containing all the possible values).

Soot provides four implementations of flow sets: `ArraySparseSet`, `ArrayPackedSet`, `ToppedSet` and `DavaFlowSet`. We will describe only the first three.

ArraySparseSet is an unbounded flow set. The set is represented as an array of references. Please note that when comparing elements for equality, it uses the method `equals` inherited from `Object`. The twist here is that soot elements (representing some code structure) don't override this method. Instead they implement the interface `soot.EquivTo`. So if you need a flow set containing for example binary operation expressions, you should implement your own version using the `equivTo` method to compare for equality.

ArrayPackedSet is a bounded flow set. Requires that the programmer provides a `FlowUniverse` object. A `FlowUniverse` object is simply a wrapper for some sort of collection or array, and it should contain all the possible values that might be put into the set. The set is represented by a bidirectional map between integers and object (this map contains the universe), and a bit vector indicating which elements of the universe are contained within the set (i.e. if bit at index 0 is set, then the set contains the element that the integer 0 maps to in the map). Be advised that this implementation suffers from the same limitations as `ArraySparseSet` concerning element equality.

ToppedSet wraps another flow set (bounded or not) adding information regarding whether it is the top set ($\top$) of the lattice.

In our very-busy expressions example, we need to have flow sets containing expressions and as such we want them to be compared for equivalence — i.e., two different occurrences of $a + b$ will be different instantiations of some class implementing `BinopExpr`; thus they will never compare equal. To remedy this, we use a modified version of `ArraySparseSet`, where we have changed the implementation of the `contains` method as such:

```
public boolean contains(Object obj) {
  for (int i = 0; i < numElements; i++)
    if (elements[i] instanceof EquivTo
        && ((EquivTo) elements[i]).equivTo(obj))
      return true;
    else if (elements[i].equals(obj))
      return true;
  return false;
}
```

5.7 Control flow graphs

Soot provides several different control flow graphs (CFG) in the package `soot.toolkits.graph`. At the base of these graphs sits the interface `DirectedGraph`; it defines methods for getting: the entry and exit points to the graph, the successors and predecessors of a given node, an iterator to iterate over the graph in some undefined order and the graphs size (number of nodes).

The implementations we will describe here are those that represent a CFG in which the nodes are Soot `Units`. Furthermore, we will only describe those that represent an intraprocedural flow.

The base class for these kinds of graphs is `UnitGraph`, an abstract class that provides facilities to build CFGs. There are three different implementations of it: `BriefUnitGraph`, `ExceptionalUnitGraph` and `TrapUnitGraph`.

BriefUnitGraph is very simple in the sense that it doesn't have edges representing control flow due to exceptions being thrown.

ExceptionalUnitGraph includes edges from `throw` clauses to their handler (`catch` block, referred to in Soot as `Trap`), that is if the trap is local to the method body. Additionally, this graph takes into account exceptions that might be implicitly thrown by the VM (e.g. `ArrayIndexOutOfBoundsException`). For every unit that might throw an implicit exception, there will be an edge from each of that units predecessors to the respective trap handler's first unit. Furthermore, should the excepting unit contain side effects an edge will also be added from it to the trap handler. If it has no side effects this edge can be selectively added or not with a parameter passed to one of the graphs constructors. This is the CFG generally used when performing control flow analyses.

TrapUnitGraph like ExceptionalUnitGraph, takes into account exceptions that might be thrown. There are three major differences:

1. Edges are added from every trapped unit (i.e., within a `try` block) to the trap handler.
2. There are no edges from predecessors of units that may throw an implicit exception to the trap handler (unless they are also trapped).
3. There is always an edge from a unit that may throw an implicit exception to the trap handler.

To build a CFG for a given method body you simply pass the body to one of the CFG constructors — e.g.

```
UnitGraph g = new ExceptionalUnitGraph(body);
```

5.8 Wrapping the results of the analysis

The results of a particular analysis are available through `AbstractFlowAnalysis`' `getFlowBefore` method, `FlowAnalysis`' `getFlowAfter` method and `BranchedFlowAnalysis`' `getBranchFlowAfter` and `getFallFlowAfter` methods. These methods all simply return the object representing the lattice element. To make this more solid, implementers are encouraged to provide an interface to their analyses, masking the results to return an unmodifiable list of the elements in the lattice.

For our very-busy expressions example we have chosen to follow the convention used in the built in Soot analyses — i.e., provide a general interface and one possible implementation of that. The interface is very simple, just providing accessors to the relevant data.

```
public interface VeryBusyExpressions {
  public List getBusyExpressionsBefore(Unit s);
  public List getBusyExpressionsAfter(Unit s);
}
```

The implementation of this interface (which we have named `SimpleVeryBusyExpressions`) performs the actual analysis and collects the data into its own maps of units to unmodifiable lists of expressions. For implementation details refer to the example source code.

Here is a short example of how to run the very-busy expressions analysis manually:

```
// Set up the class we're working with
SootClass c = Scene.v().loadClassAndSupport("MyClass");
c.setApplicationClass();
// Retrieve the method and its body
SootMethod m = c.getMethodByName("myMethod");
Body b = m.retrieveActiveBody();
// Build the CFG and run the analysis
UnitGraph g = new ExceptionalUnitGraph(b);
VeryBusyExpressions an = new SimpleVeryBusyExpressions(g);

// Iterate over the results
Iterator i = g.iterator();
while (i.hasNext()) {
  Unit u = (Unit)i.next();
  List IN = an.getBusyExpressionsBefore(u);
  List OUT = an.getBusyExpressionsAfter(u);
  // Do something clever with the results
}
```

But what is this "clever" thing we can do with our results? We'll see an example of that in Section 6.2.

6 Annotating code

The annotation framework in Soot was originally designed to support optimizations of Java programs using Java class file attributes [11]. The idea is to tag information onto relevant bits of code, the virtual machine can then use these tags to perform some optimization — e.g. excluding unnecessary array bounds checks. This framework (located in `soot.tagkit`) consists of four major concepts: `Host`s, `Tag`s, `Attribute`s and `TagAggregator`s.

Hosts are any objects that can hold and manage tags. In Soot `SootClass`, `SootField`, `SootMethod`, `Body`, `Unit`, `Value` and `ValueBox` all implement this interface.

Tags are any objects that can be tagged to hosts. This is a very general mechanism to connect name-value pairs to hosts.

Attributes are an extension to the tag concept. Anything that is an attribute can be output to a class file. In particular any tag that is tagged to a class, a field, a method or a body should implement this interface. Attributes are meant to be mapped into class file attributes and because Soot uses a tool called Jasmin to output bytecode, anything that should be output to a class file must extend `JasminAttribute`. One such implementation in Soot is `CodeAttribute`.

TagAggregators are `BodyTransformers` (see Section 6.1) that collect tags of some type and generate a new attribute to be output to a class file. The aggregator must decide where to tag the relevant information — e.g. a single unit could be transformed into several bytecode instructions, so the aggregator must decide which one any annotation on that unit should refer to. Soot provides several aggregators for its built in tags — e.g. `FirstTagAggregator` associates a tag with the first instruction tagged with it. Generally, if we use only the built in tags we don't need to worry about aggregators.

Later, with the development of the Soot plugin for Eclipse, this annotation framework was used to convey information to the plugin, to display things like analysis results visually [6]. More specifically, they introduced two new tags: `StringTag` and `ColorTag`.

StringTag is a tag whose value is just a string. These are generally tagged to units and the Eclipse plugin displays them with a popup balloon when the mouse pointer hovers over the associated source code line.

ColorTag can be tagged to anything (that is a host). The Eclipse plugin will use these tags to color either the foreground or the background (set in the tag) of the associated part — e.g. tagging this to an expression will prompt the plugin to color that expression.

6.1 Transformers

Transformers are not really a part of the tagging framework, but are used to, among other things, annotate code. In general, a transformer is an object that transforms some block of code to some other block of code. In Soot there are two different transformers: `BodyTransformer` and `SceneTransformer`. They are designed to make transformations on a single method body (i.e., intraprocedural) and on a whole application (i.e., interprocedural), respectively. To implement a transformer, one extends either one of the transformers and provides an implementation of the `internalTransform` method.

6.2 Annotating very-busy expressions

Let's take a look at how we can use this tagging mechanism to convey the results of running our very-busy expressions analysis visually to the user.

Since our example is an intraprocedural analysis, to use the results to tag code we extend `BodyTransformer` and implement its `internalTransform` method. What we would like to do is tag `StringTag`s to a unit for each busy expression flowing out of that unit. Additionally, we want to tag a `ColorTag` to each expression used in a unit that is also busy after the unit. With this information the user can easily see the flow of busy expression through his methods and quickly identify where optimization is possible. The pseudo code for this process is as follows (refer to example source code for full details):

```
internalTransform(body)
  analysis <- run very-busy exps analysis
  foreach Unit ut in body
    veryBusyExps <- busy exps after ut according to analysis
    foreach expression e in veryBusyExps
      add StringTag to ut describing e
      uses <- uses in ut
      foreach use u in uses
        if u is equivalent to e
          add ColorTag to u
        end if
      end foreach
    end foreach
  end foreach
```

To plug our analysis into Soot, we override Soot's `Main` class to inject our tagger into the Jimple transformation pack (as described in Section 4). Furthermore, to set up the Eclipse plugin to use our new main class, we need to drop it (along with whatever it depends on) into the folder "myclasses" in the soot plugin folder (`<eclipsehome>/plugins/ca.mcgill.sable.soot_<version>/myclasses/`). Note that it is also possible to put a symlink into that folder pointing to the folder containing the class files.

Now when we run Soot through Eclipse, we can tell it to use our custom main class instead of the standard Soot one (as shown in Figure 4). Figure 5 shows how the results are presented. The "SA" icons on the left side of the editor indicate that there is some analysis information there, and by hovering the mouse pointer over a statement we get a balloon with the relevant information (the `StringTag` values). Furthermore, we can see that one expression is colored red (because of the `ColorTag`), indicating that this particular expression will be evaluated again with the same value.

Another, more "clever" thing to do, is to implement a `BodyTransformer` that uses the analysis results to perform code hoisting and move the expression to the earliest program point where it is busy. This is left as an exercise for the reader.

7 Call Graph Construction

When performing an interprocedural analysis, the call graph of the application is an essential entity. When a call graph is available (only in whole-program mode), it can be accessed through the environment class (`Scene`) with the method `getCallGraph`. The `CallGraph` class and other associated constructs are located in the `soot.jimple.toolkits.callgraph` package. The simplest call graph is obtained through *Class Hierarchy Analysis* (CHA), for which no setup is necessary. CHA is simple in the fact that it assumes that all reference variables can point to any object of the correct type. The following is an example of getting access to the call graph using CHA.

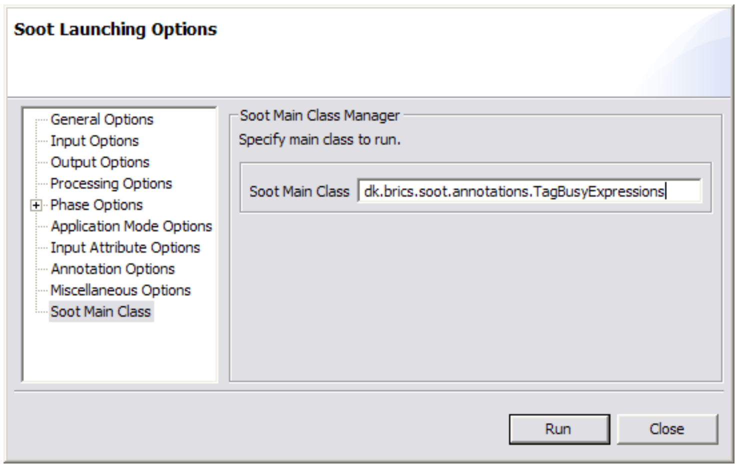

Figure 4: Setting the main class for Soot in Eclipse.

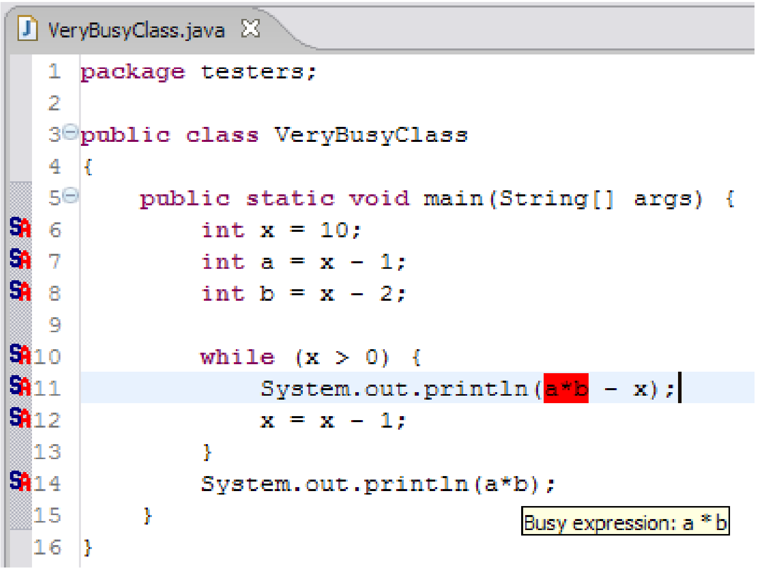

Figure 5: Very-busy expressions analysis visualized.

```
CHATransformer.v().transform();
SootClass a = Scene.v().getSootClass("testers.A");

SootMethod src = Scene.v().getMainClass().getMethodByName("doStuff");
CallGraph cg = Scene.v().getCallGraph();
```

Refer to Section 8 for points-to analyses that will produce more interesting call graphs.

7.1 Call Graph Representation

A call graph in Soot is a collection of edges representing all known method invocations. This includes:

- explicit method invocations
- implicit invocations of static initializers
- implicit calls of `Thread.run()`
- implicit calls of finalizers
- implicit calls by `AccessController`
- and many more

Each edge in the call graph contains four elements: source method, source statement (if applicable), target method and the kind of edge. The different kinds of edges are e.g. for static invocation, virtual invocation and interface invocation.

The call graph has methods to query for the edges coming into a method, edges coming out of method and edges coming from a particular statement (`edgesInto(method)`, `edgesOutOf(method) and edgesOutOf(statement), respectively`). Each of these methods return an `Iterator` over `Edge` constructs. Soot provides three so-called adapters for iterating over specific parts of an edge.

Sources iterates over source methods of edges.

Units iterates over source statements of edges.

Targets iterates over target methods of edges.

So, in order to iterate over all possible calling methods of a particular method, we could use the code:

```
public void printPossibleCallers(SootMethod target) {
  CallGraph cg = Scene.v().getCallGraph();
  Iterator sources = new Sources(cg.edgesInto(target));
  while (sources.hasNext()) {
    SootMethod src = (SootMethod)sources.next();
    System.out.println(target + " might be called by " + src);
  }
}
```

7.2 More specific information

Soot provides two more constructs for querying the call graph in a more detailed way: `ReachableMethods` and `TransitiveTargets`.

ReachableMethods. This object keeps track of which methods are reachable from entry points. The method `contains(method)` tests whether a specific method is reachable and the method `listener()` returns an iterator over all reachable methods.

TransitiveTargets. Very useful for iterating over all methods possibly called from a certain method or any other method it calls (traversing call chains). The constructor accepts (aside from a call graph) an optional `Filter`. A filter represents a subset of edges in the call graph that satisfy a given `EdgePredicate` (a simple interface of which there are two concrete implementations, `ExplicitEdgesPred` and `InstanceInvokeEdgesPred`).

8 Points-To Analysis

In this section we present two frameworks for doing points-to analysis in Soot, the SPARK and Paddle frameworks.

The goal of a points-to analysis is to compute a function which given a variable returns the set of possible targets. The sets resulting from a points-to analysis are necessary in order to do many other kinds of analysis like alias analysis or for improving precision of e.g. a call graph.

Soot provides the `PointsToAnalysis` and `PointsToSet` interfaces which any points-to analysis should implement. The `PointsToAnalysis` interface contains the method `reachingObjects(Local l)` which returns the set of objects pointed to by `l` as a `PointsToSet`. `PointsToSet` contains methods for testing for non-empty intersection with other `PointsToSet`s and a method which returns the set of all possible runtime types of the objects in the set. These methods are useful for implementing alias analysis and virtual method dispatching. The current points-to set can be accessed using the `Scene.v().getPointsToAnalysis()` method. How to create the current points-to set depends on the implementation used.

Soot provides three implementations of the points-to interface: CHA (a dumb version), SPARK and Paddle. The dumb version simply assumes that every variable might point to every other variable which is conservatively sound but not terribly accurate. Nevertheless the dumb points-to analysis may be of some value e.g. to create an imprecise call graph which may be used as starting point for e.g. a points-to analysis from which a more precise call graph might be constructed.

The Soot Pointer Analysis Research Kit (SPARK) framework and Paddle framework provide a more accurate analysis at the cost of more complicated setup and speed. Both are subset based, like Anderson's algorithm as opposed to the equivalence based Steensgaard's algorithm (see the lecture note [12]). We will discuss and show how to setup and use each of the two frameworks in the following subsections.

8.1 SPARK

SPARK is a framework for experimenting with points-to analysis in Java and supports both subset-based and equivalence based points-to analysis and anything in between. SPARK is very modular which makes it excellent for benchmarking different techniques for implementing parts of points-to analysis.

In this section we want to show how to use SPARK to set up and experiment with the basic points-to analysis provided by SPARK, and so we leave it for the curious reader to extend the various parts of SPARK with more efficient implementations.

SPARK is provided as part of the Soot framework and is found in the `soot.jimple.spark.*` packages. A points-to analysis is provided as part of SPARK and a number of facets of the analysis can be controlled using options e.g. the propagation algorithm can be either a naïve iterative algorithm or a more efficient worklist algorithm. In the example below we go through some of the most important options.

Using SPARK

The great modularity of SPARK gives a rich set of possible options and makes it less than easy to set up SPARK for anything. In this subsection we show how to setup SPARK and discuss some of the most important options.

In order for you to play with SPARK we recommend using Eclipse to either load the example code from the example source or create a new project and add the Jasmin, Polyglot and Soot jar files to the classpath. When setting up a run configuration you should add the following parameters to the JVM *-Xmx512m -Xss256m* to increase the VM memory.

We now introduce an example[4] method we wish to analyze using SPARK. The method uses the `Container` class and its `Item` class shown in Figure 6 and 7. The `Container` class has one private `item` field and a pair of get/set methods for the `item` field and the `Item` class has one package private field `data` of type object.

In Figure 8 we see the `go` method which uses the container class to create two new containers and inserts an item in each container. Furthermore a third container is declared and assigned the reference to the second container.

We would like to run SPARK on this example and expect it to discover that the points-to set of `c1` does not intersect with either of the points-to sets of `c2` or `c3` whereas `c2` and `c3` should share the same points-to set. Furthermore we would expect the `item` field of the container object allocated at `(1)` and `(2)` to point to different objects `i1` and `i2`.

To run SPARK we setup the Soot Scene to load the `Container` and `Item` classes together with the class containing the `go` method from Figure 8 using

[4]The example is similar to Example 4.4 in [8]

```
public class Container {
  private Item item = new Item();

  void setItem(Item item) {
    this.item = item;
  }

  Item getItem() {
    return this.item;
  }
}
```

Figure 6: An implementation of a simple container class

```
public class Item {
  Object data;
}
```

Figure 7: The items of the container class in Figure 6

`Scene.v().loadClassAndSupport(name);` and `c.setApplicationClass();` shown in Figure 9.

The code that does the magic of setting up SPARK is found in Figure 10 where we have listed the most interesting options and show how to use the `transform` method of the `SparkTransformer` class taking the map of options as argument to run the SPARK analysis. In the source example we have shown many more options for you to play with, you might also want to consult [7] for a description of all the options.

The options shown are:

verbose which makes SPARK print various information as the analysis goes along.

propagator SPARK supports two points-to set propagation algorithms, a naïve iterative algorithm and a more efficient worklist based algorithm.

simple-edges-bidirectional if true this option makes all edges bidirectional and hence allows an equivalence based points-to analysis like Steensgaard's algorithm.

on-fly-cg if a call graph is created on the fly which in general gives a more precise points-to analysis and resulting call graph.

set-impl describes the implementation of points-to set. The possible values are hash, bit, hybrid, array and double. Hash is an implementation based

```
  public void go() {
(1) Container c1 = new Container();
    Item i1 = new Item();
    c1.setItem(i1);

(2) Container c2 = new Container();
    Item i2 = new Item();
    c2.setItem(i2);

    Container c3 = c2;
  }
```

Figure 8: A method using the container class

```
private static SootClass loadClass(String name,
                                   boolean main) {
  SootClass c = Scene.v().loadClassAndSupport(name);
  c.setApplicationClass();
  if (main) Scene.v().setMainClass(c);
  return c;
}

public static void main(String[] args) {
  loadClass("Item",false);
  loadClass("Container",false);
  SootClass c = loadClass(args[1],true);
  ...
```

Figure 9: Code for loading classes into the Soot Scene

on the Java Collections hash set. Bit is implemented using a bit vector. Hybrid is a set, which keeps an explicit list of up to 16 elements and switches to bit vectors when the set gets larger. Array is implemented using an array always kept in sorted order. Double is implemented using two sets, one for the set of new points-to objects which have not yet been propagated and one for old points-to object which have been propagated and need to be reconsidered.

double-set-old and **double-set-new** describes implementation of the new and the old set of points-to objects in the double implementation and double-set-old and double-set-new only have effect when double is the value of set-impl.

```
HashMap opt = new HashMap();
opt.put("verbose","true");
opt.put("propagator","worklist");
opt.put("simple-edges-bidirectional","false");
opt.put("on-fly-cg","true");
opt.put("set-impl","double");
opt.put("double-set-old","hybrid");
opt.put("double-set-new","hybrid");

SparkTransformer.v().transform("",opt);
```

Figure 10: SPARK options

Running SPARK on the example using the code from the example source gives the output shown in Figure 11. The numbers in the left column refers to variable initialization points e.g. `[4,8] Container intersect? false` refers to the variable `c1` initialized at line 4 and the variable `c2` initialized at line 8. The right column describes whether the points-to set of the two variables have an empty intersection or not (i.e., true if the intersect, false otherwise).

First as a simple consistency check we see that every variable has an intersecting points-to set with itself e.g. `[4,4]`. As expected the points-to sets of variables `c1` and `c2`, and `c1` and `c3` are non-intersecting. Whereas the points-to sets of `c2` and `c3` are intersecting.

As for the item field we see that all the points-to sets are intersecting with each other even if they pertain to different container objects. The reason for this mismatch between the results and our expectations is an error in our expectations. We expected SPARK to tell the difference between the two calls to `setItem`, but SPARK is context-insensitive and so only analyzes the `setItem` method once and merges the points-to sets from each invocation of the `setItem` method. With this in mind the output corresponds exactly to what we should have expected.

SPARK is a large and robust framework for experimenting with many different aspects of context-insensitive points-to analysis. We have only covered a small number of the many options and many more combinations are available and you should use the source examples to familiarize yourself with SPARK and try out some of the other combinations than covered here.

8.2 Paddle

Paddle is a context-sensitive points-to analysis and call graph construction framework for Soot, implemented using Binary Decision Diagrams (BDD) [1]. Paddle is of comparable accuracy to SPARK for context-insensitive analysis, but also provides very good accuracy for context-sensitive analysis. The use of BDDs promises efficiency in terms of time and space especially on large programs [1]

```
[4,4]    Container intersect? true
[4,8]    Container intersect? false
[4,12]   Container intersect? false
[8,8]    Container intersect? true
[8,12]   Container intersect? true
[12,12]  Container intersect? true

[4,4]    Container.item intersect? true
[4,8]    Container.item intersect? true
[4,12]   Container.item intersect? true
[8,8]    Container.item intersect? true
[8,12]   Container.item intersect? true
[12,12]  Container.item intersect? true
```

Figure 11: SPARK output

since BDDs provide a more compact set representation than the ones used in SPARK and other frameworks. The current implementation is very slow mainly due to the patchwork of different programs that make up the implementation.

Paddle is written in Jedd [10], an extension to the Java programming language for implementing program analysis using BDDs.

Obtaining and setup of Paddle

The Paddle "front-end" is distributed along with the Soot framework. The "back-end" is distributed separately in order to avoid the need for Jedd when compiling Soot. In the following we describe how to obtain and install the backend.

Paddle requires a BDD implementation and the Jedd runtime environment to run, which in turn requires Polyglot and a SAT solver. Two BDD implementations are provided along with the Jedd runtime, the BuDDy (default) and CUDD BDD implementations; other BDD implementations like JavaBDD and SableJBDD can also be used. In the following we use the BuDDy implementation.

All these prerequisites make it complicated to setup Paddle correctly so we now give a thorough walk-through of how to setup Paddle.

1 Download the latest Paddle distribution, you should use the nightly build for the latest updates and bug fixes. The nightly build can be obtained from `http://www.sable.mcgill.ca/~olhota/build/` You should place the files in some directory e.g. `~/soot/paddle`.

2 Download the zChaff SAT solver from `http://www.princeton.edu/~chaff/zchaff.html` to some directory on

your path e.g. `~/bin/zChaff` and install by running `make all`. Make sure the programs `zchaff` and `zverify_df` are executable by you.

3 Download the Jedd runtime from `http://www.sable.mcgill.ca/jedd` and untar it to some directory e.g. `~/soot/paddle/jedd/`. Copy the scripts: `scripts/zchaff` and `scripts/zcore` to a directory on your path e.g. `~/bin/` and edit the path in the files so it points to the directory where you placed zChaff in step 2. Make sure the scripts are executable by you. Furthermore you should download and use the Jedd runtime and translator jars from `http://www.sable.mcgill.ca/~olhota/build/` to ensure that they are comparable with the downloaded Soot classes and Paddle "Back-end".

4 Download the pre-compiled jars containing the Jasmin, Polyglot and Soot classes from `http://www.sable.mcgill.ca/~olhota/build/` to a directory e.g. `~/soot/paddle/`.

In order for you to play with Paddle we recommend using Eclipse to set up a new project. In Eclipse create a new project and add the Paddle, Jedd-runtime, Jedd-translator, Jasmin, Polyglot and Soot jar files to the classpath.

When setting up a run configuration you should add the following parameters to the JVM `-Djava.library.path=absolute\_path\_to\_libjeddbuddy.so` to where `absolute\_path\_to\_libjeddbuddy.so` is the absolute path (excluding the filename) to the libjeddbuddy.so file found in the `runtime/lib` subdirectory of the directory where you placed the Jedd runtime in step 3 above. As an alternative to Eclipse you can use the make script in the example source as a starting point for setting up your own Paddle project.

You are now ready for the fun part — using Paddle to do points-to analysis.

Using Paddle

Paddle is a modular framework for context-sensitive points-to analysis which allows benchmarking of various components of the analysis e.g. benchmarking variations of context-sensitivity making it a very interesting tool. Paddle is also an early α-release and so be aware Paddle is less than robust.

In this subsection we show how to use Paddle to analyze the example in Figure 8. Furthermore we discuss and show how to use the most interesting options in the Paddle framework.

Paddle is equipped with a large set of options for configuring the analysis for your specific needs. A complete description of the options can be obtained using the Soot commandline tool: `java soot.Main -phase-help cg.paddle`

The options used for the example are shown in Figure 12. The Paddle options are specified similar to options in SPARK using a map of option names and values. The options *verbose*, *set-impl*, *double-set-new*, and *double-set-old* are the same as for SPARK. The *q* option determines how queues are implemented, and *enabled* needs to be true for the analysis to run. *propagator* controls which propagation algorithm is used when propagating points-to sets, we leave it up to Paddle to chose and set it to auto. *conf* controls whether a call graph should be created on-the-fly or ahead of time. The implementation of Paddle is subset-based but equivalence-based analysis can be simulated by setting the *simple-edges-bidirectional* option to true. The last four options are the most essential for the working of Paddle so we describe them in some detail.

bdd - The **bdd** option toggles BDD on or off. If *true* then use the BDD version of Paddle, if false don't. Default is false.

backend - The **backend** option selects the BDD backend. Either *buddy* (BuDDy), *cudd* (CUDD), *s*able (SableJBDD), *javabdd*(JavaBDD) or *none* for no BDDs. Default is *buddy.*

context - The **context** option controls the degree of context-sensitivity used in the analysis. Possible values are *insens*, Paddle performs a context-insensitive analysis like SPARK. *1cfa* Paddle performs a 1-cfa context-sensitive analysis. *kcfa* Paddle performs a k-cfa context-sensitive, where k is specified using the **k** option. *objsens* and *kobjsens* makes Paddle perform a 1-object-sensitive and k-object-sensitive analysis respectively. *uniqkobjsens* makes Paddle perform a unique-k-object-sensitive analysis. Default is *insens.*

k - The **k** option specifies the maximum length of a call string or receiver object string used as context when the value of the **context** option is either of *kcfa*, *kobjsens*, or *uniqkobjsens.*

A short introduction to k-cfa context-sensitive analysis is appropriate, we only intend to provide the intuition of the subject and encourage the reader to read Section 4.1.2.1 of [8] for a thorough introduction to call-site context-sensitive analyses.

k-cfa context-sensitive analysis is based on strings of call-sites as contexts and the k describes the maximum length of these strings. A context-sensitive analysis only using the callsite as context gives good results for examples like the one in Figure 8, but if an additional layer of indirection is added e.g. an identity function is called from `setItem` then a context-sensitive analysis depending only on the call-site merges the points-to sets of the two calls to the `setItem` method, since they use the same call-sites to the identity function. By using a string of call-sites we are able to distinguish the two calls to the identity function and hence keep the points-to sets separate. The number of indirections in the program can be arbitrarily large so we need to fix the length to some k — hence the k-cfa.

We will now use Paddle to analyze the same example as we used in the SPARK section above see Figure 8.

As a sanity check we start by analyzing the method in Figure 8 using Paddle with the **context** option set to *insens.* We would expect the result to be the

```
HashMap opt = new HashMap();
opt.put("enabled","true");
opt.put("verbose","true");
opt.put("bdd","true");
opt.put("backend","buddy");
opt.put("context","1cfa");
opt.put("k","2");
opt.put("propagator","auto");
opt.put("conf","ofcg");
opt.put("order","32");
opt.put("q","auto");
opt.put("set-impl","double");
opt.put("double-set-old","hybrid");
opt.put("double-set-new","hybrid");
opt.put("pre-jimplify","false");

PaddleTransformer pt = new PaddleTransformer();
PaddleOptions paddle_opt = new PaddleOptions(opt);
pt.setup(paddle_opt);
pt.solve(paddle_opt);
soot.jimple.paddle.Results.v().makeStandardSootResults();
```

Figure 12: Paddle options

same as for the SPARK analysis since Paddle and SPARK are of similar accuracy when performing context-insensitive analysis. The result of the analysis fully meets our expectations and we get the same result as in Figure 11.

The more interesting example is to run Paddle with context-sensitivity enabled to see if Paddle can deduce that the item field created as part of the Container variable declared on line 4 does not share points-to set with any of the other item fields. Running Paddle on the example in Figure 8 with the **context** option set to *1cfa* gives the same result as in the SPARK case (see Figure 11). As expected the points-to information for the Container variables is similar to the information obtained using SPARK since we do not need context-sensitivity to distinguish those points-to sets. But the information for the Container.item variables is also the same as for the SPARK example which is not what we expected!

The reason for this unexpected behavior is the line: `private Item item = new Item();` in the Container class. Paddle does not use context-sensitive heap abstraction as default and so the item field of every container object is represented using the same abstract Item. If we change the line to `private Item item;` then the item field of the containers do not point to the same abstract object and running Paddle on the new Container class yields the expected result as shown in Figure 13.

Another way to obtain the correct result is to turn on the context-sensitive heap abstract `opt.put("context-heap","true");` which allows Paddle to distinguish the different Items assigned to the different Containers.

```
[4,4]    Container intersect? true
[4,8]    Container intersect? false
[4,12]   Container intersect? false
[8,8]    Container intersect? true
[8,12]   Container intersect? true
[12,12]  Container intersect? true
[4,4]    Container.item intersect? true
[4,8]    Container.item intersect? false
[4,12]   Container.item intersect? false
[8,8]    Container.item intersect? true
[8,12]   Container.item intersect? true
[12,12]  Container.item intersect? true
```

Figure 13: Paddle output

Paddle is a framework for experimenting with a great number of aspects of context-sensitive points-to analysis. We have only covered a small number of the many options and even more possible combinations are available and you should use the source example to learn more about the many features of Paddle. But be aware that Paddle is still α-software and that a number of options might only work in some combinations with other options. Check the Soot mailing-list or visit the Paddle homepage and download the nightly build often.

8.3 What to do with the points-to sets?

In the last two subsections we described in some detail how to use the SPARK and Paddle points-to analysis frameworks to obtain points-to sets for the variables in a given program using either context-insensitive or context-sensitive analysis. We saw that it is rather complicated to setup the two frameworks so it is natural to say a few words why we should bother to do so in the first place.

Points-to (or alias) information is a necessity in order to obtain precision in many analysis and transformations. For example a precise points-to analysis can be used to get a precise null pointer analysis or a more precise call graph, which in turn may lead to more precision in other analysis. Precise points-to information is also vital for the accuracy of e.g. partial evaluation of imperative and object oriented languages.

9 Extracting Abstract Control-Flow Graphs

In this section we will show how to use Soot to extract a custom *intermediate representation* of an abstract control-flow graph usable as a starting point for your own analysis and transformations.

An abstract control-flow graph captures the relevant parts of the control-flow and abstracts away the irrelevant parts. Removing the irrelevant parts is often necessary in order to get a tractable and yet sufficient representation upon which to implement an analysis.

A good example is the Java String Analysis (JSA)[3] where various operations like concatenation on Java strings are tracked and analyzed. In JSA only the part of the control-flow having to do with strings is relevant for the analysis of strings, hence the other parts of the control-flow are removed during abstraction and the analysis is phrased on the abstract representation.

```
public class Foo {
  private int i;
  public int foo(int j) {
    this.i = j;
  }
}
```

Figure 14: The Foo class which we want to track throughout the program

In the following we show you how to use Soot to create an abstract control-flow graph. We will use the Foo class shown in Figure 14 which has one method manipulating the state of the object the `foo` method (could be the concatenation method of java.lang.String). We want to be able to track how Foo objects evolves during program execution and so we need an abstract control-flow graph only concerned with the parts of the control-flow which has to do with Foo objects.

9.1 The Abstract Foo Control-flow Graph

The abstract Foo control-flow graph is a description of the control-flow related to Foo programs. We represent the control-flow graph as a very small subset of Java which we call the Foo intermediate representation described in the BNF in Figure 15. The Foo intermediate representation only contains six different kinds of statements and three types and hence is very compact and manageable.

A program in the Foo intermediate representation is a number of methods containing a number of statements described by the BNF in Figure 15 where **f** ranges over identifiers of type Foo, **m** ranges over method names, int is the Java integer type and τ is either the type Foo or any other type.

The six kinds of statements are: initialization of Foo objects, method call on the `foo` method of the Foo class, method call on some other method than `foo`

stmt ::= **Foo f = new Foo()** - Foo initialization
| **f.foo(int)** - Foo method call
| **m**(τ^*) - Some method call
| **f = m**(τ^*) - Some method call with Foo return type
| **f1 = f2** - Foo assignment
| **nop** - Nop

τ ::= **Foo**
| **SomeOtherTypeThanFoo**

Figure 15: BNF for statements in the Foo intermediate representation

with a different return type than Foo, method call on some other method than `foo` with return type equal to Foo and last a nop statement.

Statements in Java map as one would expect. An instantiation of a Foo object maps to Foo initialization. A method call to the `foo` method maps to a Foo method call, an assignment of type Foo translates to a Foo assignment, method calls other than on the `foo` method translates to some method call with or without Foo return type respectively. In the intermediate representation we treat non-Foo object instantiation as some method call which just happens to be to a constructor method and we treat casts to `Foo` as some method call with Foo as return type. We completely disregard control-structure and exceptions for sake of brevity. Any analysis based on this abstract representation is not sound for programs throwing exceptions which might interfere with the value of Foo objects.

9.2 Implementation

Implementing the transformation from Java to the Foo intermediate representation involves two steps: First implementing the Foo intermediate representation and Second implementing the translation between Java and the intermediate representation.

The implementation of the Foo intermediate representation is straight forward. Each kind of statement extends the Statement class which provides some general functionality needed in order to be a node in the control-flow graph e.g. store the set of predecessor and successor statements.

Along with this note we provide a proof of concept implementation of the transformation presented in this section, you should consult it as you read along. Be aware that the provided code does not cover all cases necessary for a complete and robust translation of Java, but only shows how to use the concepts of this section to do the transformation.

The code is organized in five subpackages of the `dk.brics.soot.intermediate` package. The `main` subpackage contains the main program(Main.java) which uses the transformation to show a textual representation of the abstract control-flow graph of the test program FooTest.java from the `foo` subpackage this should be the starting point when running or inspecting the code. In the `foo` subpackage you also find the Foo class as described above. In the `representation` subpackage you find the implementation of the intermediate representation and some additional classes for variables and methods. Furthermore a visitor (the StatementProcessor.java) is provided for traversing statements. In the subpackage `foonalasys` you find the class Foonalasys which is the representation of the analysis one might want to do upon the intermediate representation and so the first step of the Foonalasys is to instantiate and run the translation from Java to the Foo intermediate representation. The translation classes are located in the `translation` subpackage. Here you will find the three classes responsible for the translation: JavaTranslator, StmtTranslator, and ExprTranslator.

The translation from Java is done via Jimple and so the translation requires a good understanding of Jimple and how various Java constructs are mapped to Jimple in order to recognize them when translating from Jimple to the Foo intermediate representation e.g. object creation and initialization is done in one new expression in Java, but has been separated into two constructs in Jimple like in Java bytecode and so care must be taken to handle this in the translation from Jimple to the Foo intermediate representation.

Soot provides infrastructure for the translation. Especially the `AbstractStmtSwitch` and the `soot.jimple.AbstractJimpleValueSwitch` classes are useful. `AbstractStmtSwitch` is an abstract visitor which provides methods (operations) for the different kinds of statements available in Java (Soot). Similarly, `soot.jimple.AbstractJimpleValueSwitch` is an abstract visitor which provides methods (operations) for the different Jimple values like virtualInvoke, specialInvoke, and add expressions.

The translation is implemented using the three classes: `JavaTranslator`, `StmtTranslator`, and `ExprTranslator`. The `JavaTranslator` class is responsible for translating the various methods of the given Java program and connecting the translated statements together. `JavaTranslator` maintains an array of methods which is initialized in the `makeMethod` method. The `translate` method is the main method where the actual translation and linking of the translated statements is done.

The individual statements are translated using the `StmtTranslator` class which extends the `AbstractStmtSwitch` class. In our experience the subset we use in `StmtTranslator` should be sufficient for most uses. The complete list of methods provided by `AbstractStmtSwitch` is available in the Soot javadoc. The main method of `StmtTranslator` is the `translateStmt` method which applies `StmtTranslator` to the statement and maintains a map from Soot statements to the first statement of the corresponding code in the Foo representation. The map is handy for error reporting. Furthermore `StmtTranslator` has a method `addStatement` which adds the given Foo statement to the correct method and maintains a reference to the first statement. The `addStatement` method is the one used throughout to add statements during the translation.

Subexpressions are translated using the `ExprTranslator` which is an extension of the `soot.jimple.AbstractJimpleValueSwitch` class and so overrides a number of methods in order to implement translation. The entry point is the `translateExpr` method which basically applies `ExprTranslator` to the given ValueBox. We only implement the methods needed to make our example run since our goal is only to introduce how an abstract control-flow graph is created. The most interesting methods are the `caseSpecialInvokeExpr` and `handleCall` methods. `caseSpecialInvokeExpr` tests if we are dealing with an initialization of a Foo object and if so creates a Foo initialization statement, if not it is just some other method call and the handleCall method is executed. At this point the source code differs from the representation described above, in the source we do not distinguish between the two kinds of other method calls, this is left as an exercise for the reader.

Implementing the full transformation from Java to the Foo intermediate representation is just hard and tedious work matching the Jimple code sequences to Foo language constructs.

Concluding

Soot provides some support for creating your own abstract control-flow graph but Java is a large language and you have to consider many different aspects when implementing the translation from Jimple to your own representation, furthermore you have to figure out how Java constructs map to Jimple and then how Jimple constructs should be mapped to your representation. But besides this the two abstract classes `AbstractStmtSwitch` and `soot.jimple.AbstractJimpleValueSwitch` provide good starting points for the translation and work very well.

10 Conclusion

In the previous sections we have presented our own documentation for the Soot framework. We have documented the parts of the Soot framework we have used earlier in various projects: parsing class files, performing points-to and null pointer analysis, performing data-flow analysis, and extracting abstract control-flow graphs. It is our hope that this note will leave a novice users in a less of a state of shock and awe and so may provide some of the stepping stones which could make Soot more easily accessible.

References

[1] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Inplementation*, pages 103–114. ACM Press, 2003.

[2] G. Bilardi and K. Pingali. The static single assignment form and its computation, 1999.

[3] A. Christensen, A. Mller, and M. Schwartzbach. Precise analysis of string expressions, 2003.

[4] E. Gagnon and L. Hendren. Intraprocedural inference of static types for java bytecode, 1999.

[5] Laurie Hendren, Patrick Lam, Jennifer Lhotak, Ondrej Lhotak, , and Feng Qian. Soot, a tool for analyzing and transforming java bytecode.

[6] Jennifer Lhoták, Ondřej Lhoták, and Laurie Hendren. Integrating the Soot compiler infrastructure into an IDE. In E. Duesterwald, editor, *Compiler Construction, 13th International Conference*, volume 2985 of *LNCS*, pages 281–297, Barcelona, Spain, April 2004. Springer.

[7] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[8] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[9] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[10] Ondřej Lhoták and Laurie Hendren. Jedd: a bdd-based relational extension of java. *SIGPLAN Not.*, 39(6):158–169, 2004.

[11] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In R. Wilhelm, editor, *CC 2001*, volume 2027 of *Lecture Notes in Computer Science*, pages 334+, 2001.

[12] Michael I. Schwartzbach. Lecture note on static analysis.

[13] R. Vall, e Phong, C. Etienne, G. Laurie, H. Patrick, and L. Vijay. Soot - a java bytecode optimization framework, 1999.