SOOT: A JAVA BYTECODE OPTIMIZATION FRAMEWORK

by
Raja Vallée-Rai

School of Computer Science
McGill University, Montreal

October 2000

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Java provides many attractive features such as platform independence, execution safety, garbage collection and object orientation. These features facilitate application development but are expensive to support; applications written in Java are often much slower than their counterparts written in C or C++. To use these features without having to pay a great performance penalty, sophisticated optimizations and runtime systems are required.

We present SOOT, a framework for optimizing Java bytecode. The framework is implemented in Java and supports three intermediate representations for representing Java bytecode: BAF, a streamlined representation of bytecode which is simple to manipulate; JIMPLE, a typed 3-address intermediate representation suitable for optimization; and GRIMP an aggregated version of JIMPLE suitable for decompilation. SOOT also contains a set of transformations between these intermediate representations, and an application programming interface (API) is provided to write optimizations and analyses on Java bytecode in these forms.

In order to demonstrate the usefulness of the framework, we have implemented intraprocedural and whole program optimizations. To show that whole program bytecode optimization can give performance improvements, we provide experimental results for 10 large benchmarks, including 8 SPECjvm98 benchmarks running on JDK 1.2. These results show a speedup of up to 38%.

Résumé

Java possède beaucoup de propriétés attrayantes telles que l'indépendance de plateforme, la sûreté d'exécution, le ramasse-miettes et l'orientation d'objets. Ces dispositifs facilitent le développement d'applications mais sont chers à supporter; les applications écrites en Java sont souvent beaucoup plus lentes que leurs contre-parties écrites en C ou C++. Pour utiliser ces dispositifs sans devoir payer une grande pénalité d'exécution, des optimisations sophistiquées et des systèmes d'exécution sont exigés.

Nous présentons SOOT, un cadre pour optimiser le bytecode de Java. Le cadre est programmé en Java et supporte trois représentations intermédiaires pour le bytecode de Java: BAF, une représentation simplifiée du bytecode qui est simple à manipuler; JIMPLE, une représentation intermédiaire à 3 addresses appropriée à l'optimisation; et GRIMP, une version agrégée de JIMPLE appropriée à la décompilation. SOOT contient également un ensemble de transformations entre ces représentations intermédiaires, et une interface de programmation d'application (api) est fournie pour écrire des optimisations et des analyses sur le bytecode de Java sous ces formes.

Afin de démontrer l'utilité du cadre, nous avons implémenté des optimisations intraprocedurales et globales de programme. Pour prouver que l'optimisation globale de bytecode de programme peut donner des améliorations d'exécution, nous fournissons des résultats expérimentaux pour 10 applications, y compris 8 programmes de SPECjvm98 exécutant sur JDK 1.2. Les résultats produisent une amélioration allant jusqu'à 38%.

Acknowledgments

The SOOT framework was really a monumental team effort.

I would like to thank my advisor, Laurie Hendren, who played a huge role in leading the project and keeping the mood of the development team optimistic even when the going got tough. I can not thank her enough for her support and constant encouragement.

The co-developers of SOOT were key in making SOOT a reality. Large scale projects need many developers; thanks to all those people who contributed to SOOT. In particular, I would like to thank (1) Patrick Lam for his superb help with the maintainence and development of the later phases of the framework, (2) Etienne Gagnon for his excellent work on developing a robust typing algorithm for JIMPLE, and (3) Patrice Pominville for his great BAF bytecode optimizer.

What is a framework without users? I would like to thank all the SOOT users for the feedback that they gave. In particular, I would like to give a special thanks to the two super users, Vijay Sundaresan and Chrislain Razafimahefa, who gave me a great amount of support in the early days of SOOT and who tolerated the constant changes that I made to the API.

Special thanks goes to my good friends Paul Catanu and Karima Kanji-Tajdin who encouraged me and helped me out both in Victoria and in Montreal. I would also like to thank Derek Rayside for his encouragement and advice as I re-established myself in Montréal in Spring of 2000.

And last, but not least, I would like to thank my family members. They have been extremely supportive despite the troubled times that we have gone through recently. Thanks Mom, Dad, Anne-Sita and Manuel!

This work was supported by the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche as well as IBM's Centre for Advanced Studies.

Contents

Chapter 1

Introduction

1.1 Motivation

Java provides many attractive features such as platform independence, execution safety, garbage collection and object orientation. These features facilitate application development but are expensive to support; applications written in Java are often much slower than their counterparts written in C or C++. To use these features without having to pay a great performance penalty, sophisticated optimizations and runtime systems are required. Using a Just-In-Time compiler[1], or a Way-Ahead-Of-Time Java compiler[19] [18], to convert the bytecodes to native instructions is the most often used method for improving performance. There are other types of optimizations, however, which can have a substantial impact on performance:

Optimizing the bytecode directly: Some bytecode instructions are much more expensive than others. For example, loading a local variable onto the stack is inexpensive; but virtual methods calls, interface calls, object allocations, and catching exceptions are all expensive. Traditional C-like optimizations, such as copy propagation, have little effect because they do not target the expensive bytecodes. To perform effective optimizations at this level, one must consider more advanced optimizations such as method inlining, and static virtual method call resolution, which directly reduce the use of these expensive bytecodes.

Annotating the bytecode: Java's execution safety feature guarantees that all potentially illegal memory accesses are checked for safety before execution. In some situations it can be determined at compile-time that particular checks are unnecessary. For example, many array bound checks can be determined to be completely unnecessary[12]. Unfortunately, after having determined the safety of some array accesses, we can not eliminate the bounds checks directly from the bytecode, because they are implicit in the array access bytecodes and can not be separated out. But if we can communicate the safety of these instructions to the Java Virtual Machine by some annotation mechanism, then the Java Virtual Machine could speed up the execution by not performing these redundant checks.

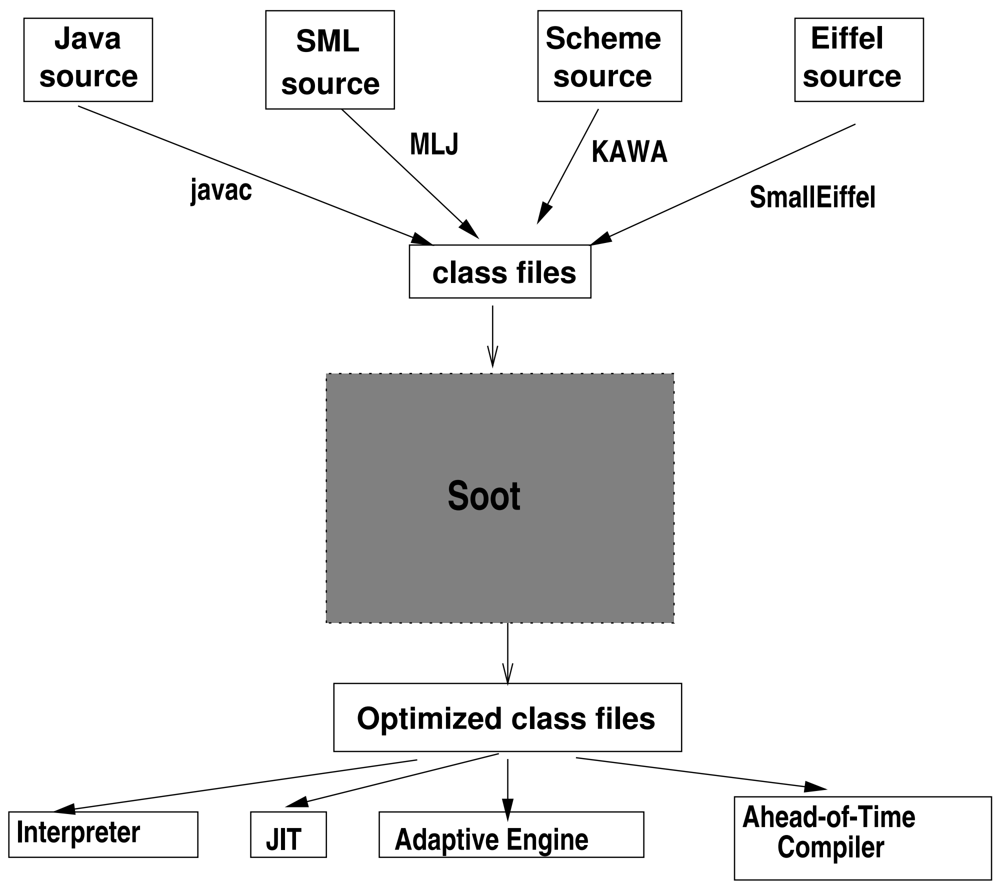

Figure 1.1: An overview of Soot and its usage.

The goal of this work is to provide a framework which simplifies the task of optimizing Java bytecode, and to demonstrate that significant optimization can be achieved. The

SOOT[22] framework provides a set of intermediate representations and a set of Java APIs for optimizing Java bytecode directly. The SOOT framework is used as follows: (figure 1.1)

1. Bytecode is produced from a variety of sources, such as the `javac` compiler.
2. This bytecode is fed into SOOT, and SOOT transforms and/or optimizes the code and produces new classfiles.
3. This new bytecode can then be executed using any standard Java Virtual Machine (JVM) implementation, or it can be used as the input to a bytecode→C or bytecode→native-code compiler or other optimizers.

Based on the SOOT framework we have implemented both intraprocedural optimizations and whole program optimizations. The framework has also been designed so that we will be able to add support for the annotation of Java bytecode. We have applied our tool to a substantial number of large benchmarks, and the best combination of optimizations implemented so far can yield a speed up reaching 38%.

1.2 Contributions of Thesis

The contributions of this thesis are the design, implementation and experimental validation of the SOOT framework.

1.2.1 Design

The SOOT framework was designed to simplify the process of developing new optimizations for Java bytecode. The design can be split into two parts. The first part is the actual design of the three intermediate representations for Java bytecode:

- BAF, a streamlined representation of bytecode which is simple to manipulate;
- JIMPLE, a typed 3-address intermediate representation suitable for optimization;
- GRIMP, an aggregated version of JIMPLE suitable for decompilation.

GRIMP was designed with Patrick Lam, and the development of JIMPLE was built upon a prototype designed by Clark Verbrugge. Optimizing Java bytecode in SOOT consists of transforming bytecode to the JIMPLE representation and then back. BAF and GRIMP are used in the transformation process.

1.2.2 Implementation

SOOT was a team effort. However, I was the main designer and implementator. In particular, I implemented large portions of the framework and then coordinated the implementation of many aspects of the framework. In particular, I implemented the following:

- implementation of the base framework, consisting of the main classes such as `Scene`, `SootClass`, `SootMethod`, `SootField` and all the miscellaneous classes,
- the bytecode to verbose JIMPLE transformation (used Clark Verbrugge's code as a prototype),
- compacting the verbose JIMPLE code by copy and constant propagation,
- implementation of the flow analysis framework and various flow analyses such as live variable analysis, reaching defs and using defs,
- implementation of a simple register colorer, of local variable splitting and of local variable packing,
- implementation of unreachable code elimination and dead assignment elimination.

1.2.3 Experimental Validation

A large amount of effort has been placed in validating our framework on a large set of benchmarks on a variety of virtual machines. In particular, we tested our framework on five different virtual machines (three under Linux and two under NT), and report results on ten large programs, seven of which originate from the standard specSuite set.

1.3 Related Work

Related work falls into five different categories:

Bytecode optimizers: The only other Java tool that we are aware of which performs significant optimizations on bytecode and produces new class files is Jax[26]. The main goal of Jax is application compression where, for example, unused methods and fields are removed, and the class hierarchy is compressed. They also are interested in speed optimizations, but at this time their current published speed up results are more limited than those presented in this paper. It would be interesting, in the future, to compare the results of the two systems on the same set of benchmarks.

Bytecode manipulation tools: there are a number of Java tools which provide frameworks for manipulating bytecode: JTrek[14], Joie[3], Bit[15] and JavaClass[13]. These tools are constrained to manipulating Java bytecode in their original form, however. They do not provide convenient intermediate representations such as BAF, JIMPLE or GRIMP for performing analyses or transformations.

Java application packagers: There are a number of tools to package Java applications, such as Jax[26], DashO-Pro[5] and SourceGuard[23]. Application packaging consists of code compression and/or code obfuscation. Although we have not yet applied SOOT to this application area, we have plans to implement this functionality as well.

Java native compilers: The tools in this category take Java applications and compile them to native executables. These are related because they all are forced to build 3-address code intermediate representations, and some perform significant optimizations. The simplest of these is Toba[19] which produces unoptimized C code and relies on GCC to produce the native code. Slightly more sophisticated, Harissa[18] also produces C code but performs some method devirtualization and inlining first. The most sophisticated systems are Vortex[6] and Marmot[8]. Vortex is a native compiler for Cecil, C++ and Java, and contains a complete set of optimizations. Marmot is also a complete Java optimization compiler and is SSA based. Each of these systems include their customized intermediate representations for dealing with Java bytecode, and produce native code directly. There are also numerous commercial Java native compilers, such as the IBM High Performance for Java Compiler[20], Tower Technology's TowerJ[27], and SuperCede[25], but they have very little published information. The intention of our work is to provide a publically available infrastructure for bytecode optimization. The optimized bytecode could be used as input to any of these other tools.

Java compiler infrastructures: There are at least two other compiler infrastructures for Java bytecode. Flex is a Java compiler infrastructure for embedded parallel and distributed systems. It uses a form of SSA intermediate representation for its bytecode called QuadSSA. Optimizations and analyses can be written on this intermediate representation, but new bytecodes can not be produced.[9]

The other well known compiler infrastructure is the Suif compiler system [24]. It possesses a front-end which translates Java bytecode to `OSUIF` code which is an object oriented version of Suif code which can express object orientation primitives. Analyses and transformations can be written on this intermediate representation and numerous back-ends enable the compilation of the code to native code. It is not possible to produce new bytecode, however.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 describes the three intermediate representations contained in SOOT. Chapter 3 presents the transformations which are required to transform code between these intermediate representations. Chapter 4 presents the experimental results which validate the framework. Chapter 5 presents the application programming interface (API) which we developed to enable the use of the framework, and chapter 6 presents some experiences that we had while developing the framework. Finally, chapter 7 gives our conclusions and future work.

Chapter 2

Intermediate Representations

SOOT provides the three intermediate representations BAF, JIMPLE, and GRIMP. Each representation is discussed in more detail below, and figures 2.7, 2.12 and 2.16 provide an example program in each form. Figure 2.1 gives the example program in the original Java form. As a starting point, the following section discusses some general aspects of stack-based representations.

```
int a;
int b;
public int stepPoly(int x)
{
    int[] array = new int[10];

    if(x > 10)
    {
        array[0] = a;
        System.out.println("foo");
    }
    else if(x > 5)
        x = x + 10 * b;

    x = array[0];

    return x;
}
```

Figure 2.1: `stepPoly` in its original Java form.

2.1 Java Bytecode as an Intermediate Representation

In this section we discuss the benefits of stack-based code in general (subsection 2.1.1), and then examine some disadvantages of analyzing and optimizing stack-based code directly (subsection 2.1.2).

2.1.1 Benefits of stack-based representations

The stack machine model for the Java Virtual Machine was perhaps a reasonable choice for a few reasons. Stack machine interpreters are relatively easy to implement and this was originally important because the goal was to implement the Java Virtual Machine on as many different platforms as possible. More relevantly, stack-based code tends to be compact and this is essential to allow class files to be rapidly downloaded over the Internet. A third justification for this model is that it simplifies the task of code generation. Since the operand stack can be used to store intermediate results, simple traversals of the code's abstract syntax tree(AST) suffice to generate Java bytecode.

There are two good reasons for manipulating Java bytecode directly:

The stack code is immediately available: No transformations are required to get the stack code in this form, as it is the native form found in Java classfiles. This is important if execution speed is critical (such as for JIT compilers).

The resultant stack code is final: Since the code does not need to be transformed to be stored in the classfiles, we have complete control over what gets stored. This is important for obfuscation since many of the obfuscation techniques make heavy use of the stack to confuse decompilers, and a 3-address code representation hides the stack.

In specific cases, such as those mentioned above, a stack-based representation is useful. However, in the general case where we optimize classfiles offline, these advantages pale in comparison to the following disadvantages.

2.1.2 Problems of optimizing stack-based code

Even though there are advantages for choosing a stack-based intermediate representation, there are potential disadvantages with respect to program analysis and optimization. To analyze and transform Java bytecode directly, one is forced to add an extra layer of complexity to deal with the complexities of the stack-based model. Given that it is of critical importance to optimize Java this drawback is very important and must be eliminated to allow the clearest and most efficient development of optimizations on Java.

Below, we enumerate some ways in which stack-based Java bytecode is complicated to optimize.

Expressions are not explicit: In 3-address code, expressions are explicit. Usually they only occur in assignment statements (such as `x = a + b`) and branch statements (such as a `if a<b goto L1`). There is a fixed set of possible expressions, simplifying analyses by restricting the number of cases to consider. For the purposes of this section, we shall distinguish two classes of Java bytecode instructions: the *expression* instructions, and *action* instructions. Expression instructions are those which only produce an effect on the operand stack. Examples of this class are: `iload, iadd, imul, pop`. Action instructions, on the other hand, produce a side effect, such as modifying a field (`putfield`), calling a method (`invokestatic`) or storing into a local variable (`istore`). These instructions have concrete effects, whereas the expression instructions are merely used to build arguments on the stack.

Thus in order to determine the expression being acted upon by an action instruction, you need to parse the expression instructions and reconstruct the expression tree, whereas in JIMPLE these are readily available. And as the next points illustrate, this reconstruction process is not a trivial problem.

Expressions can be arbitrarily large: In order to determine the expression being computed by expression instructions, the analysis must examine the instructions preceding the action instruction and build an expression tree. For a simple case such as:

```
iconst 5
iload 0
iadd
istore 1
```

it is easy to determine that the expression being stored in var1 is `5 + var0`. In some cases, such as:

```
iload 3
iconst 5
iload 6
iload 3
iadd
imul
idiv
istore 0
```

the expression tree is more complex. In this case it is `(var3 + 5) * var6 / var0`. Variable expression length is a complication, some analyses such as common subexpression elimination require having simple 3-address code expressions available to be implemented efficiently. To use expression trees in such analyses, they would need to first be simplified to use temporary locals, which the JIMPLE form provides directly.

Concrete expressions can not always be constructed: Due to the nature of the operand stack, the associated expression instructions for a given action instruction are not necessarily immediately next to it. The following store still stores `var0 + 5` in var1, despite the intermingled bytecode instructions which store `var2 * var3` in var4.

```
iconst 5
iload 0
iadd
iload 2
iload 3
imull
istore 4
istore 1
```

If a complete sequence of expression instructions reside in a basic block, then it is always possible to recover the computed expression. However, since the Java Virtual Machine does not require a zero stack depth across control flow junctions, an expression used in a basic block can be partially computed in a different basic block. Consider the following example:

```
iload 0
iload 2
if_icmpeq label1
goto label2

label1:
ineg

label2:
istore 1
```

When computing the possible definitions for a variable in a 3-address code intermediate representation, the number of possible definitions can not exceed the number of assignments to that variable. This example illustrates that this is not the case with stack code, for a single assignment (`istore 0`) can yield two different definitions (-var0 or var0). By allowing the control flow to produce such conditional expressions obviously increases the complexity of analyses such as reaching definitions and optimizations such as copy and constant propagation. Instead of considering just assignments, they must consider the origins of expressions and their possible multiplicity.

Simple transformations become complicated: The main reason why stack code complicates analyses and transformations is its piecemeal form. The fact that the expression is split into several pieces and is separated from the action instruction causes almost all the complications, for as a result, you can interleave these instructions with other instructions, and spread them over control flow boundaries. Transforming the code in this form is difficult because all the separate pieces need to be kept track of and maintained. To illustrate this point, this subsection considers the problems associated with performing dead code elimination.

In 3-address code, eliminating a statement is often accomplished by simply deleting it from a graph or list of statements. In Java bytecode, removing an action instruction is similar, except that you must also remove all the associated expression instructions, in order to avoid accumulating unused arguments on the stack. This sounds relatively simple, but there is a catch: if the set of expression instructions cross a control flow boundary, then this may not be possible, because other paths depend on the stack depth to be a certain height. For example:

```
      iload 0
      iload 1
    /        \
iadd          imul
istore 5      istore 5
...           use(5)
              ...
```

Despite the fact that on the left hand side the local variable 5 is dead, the `iadd` and `istore 5` cannot be simply deleted, because we must ensure the two arguments on the stack are still consumed. The best we can do is replace the two instructions with two `pops`.

For developing analyses and transformations, it should be clear that working with 3-address code is much simpler and more efficient than dealing with stack code.

2.2 BAF

BAF is a bytecode representation which is stack-based, but without the complications that are present in Java bytecode. Although the primary goal of the Soot framework is to avoid having to deal with bytecode as stack code, it is still sometimes necessary to analyze or optimize bytecode in this form. The following subsections give BAF's motivation, a description of BAF, its design features, and then a walkthrough of some sample code.

2.2.1 Motivation

The main motivation for BAF is to simplify the development of analyses and transformations which absolutely must be performed on stack code. In our case, there are two such occurances. First, in order to produce JIMPLE code, it is necessary to calculate the stack height before every instruction. Second, before producing new bytecode, it is convenient to perform some peephole optimizations and stack manipulations to eliminate redundant load/stores. These analyses and transformations could be performed on Java bytecode directly, but it is much more convenient to implement them on BAF because of its design features, which are described below.

2.2.2 Description

BAF is a stack-based intermediate representation of Java bytecode which consists of a set of orthogonal instructions. See figure 2.2 for the list of BAF instructions. The words in italics represent attributes for the instructions. For example, *t* means a type, so actual instances of add.*t* can be `add.i`, `add.l`, `add.f`, `add.d` depending on whether the type of the `add` instruction is an integer, long, float or double. For instructions such as `load` or `store`, there are two attributes, the local variable and the type of the instruction. For `dup2`, there are two types as well, the two types to be duplicated on the stack. Most of these instructions follow the Java Virtual Machine specification[16], except that the instruction names have been made more consistent by requiring that the more specific portion of the variable name be on the left. Hence the name interfaceinvoke as opposed to invokeinterface.

local := @this	interfaceinvoke *method n*	add.*t*
local := @parameter*n*	specialinvoke *method*	and.*t*
local := @exception	staticinvoke *method*	cmpg.*t*
dup1.*t*	virtualinvoke *method*	cmp.*t*
dup1_x1.*t_t*	fieldget *field*	cmpl.*t*
dup1_x2.*t_tt*	fieldput *field*	div.*t*
dup2.*tt*	staticget *field*	mul.*t*
dup2_x1.*tt_t*	staticput *field*	neg.*t*
dup2_x2.*tt_tt*	load.*t local*	or.*t*
t2t	store.*t local*	rem.*t*
checkcast *refType*	inc.i *local constant*	shl.*t*
instanceof *type*	new *refType*	shr.*t*
lookupswitch	newarray *type*	ushr.*t*
{case $value_1$: goto $label_1$	newmultiarray *type n*	xor.*t*
...	arraylength	ifne *label*
case $value_n$: goto $label_n$	ifcmpne.*t label*	ifeq *label*
default: goto *defaultLabel*}	ifcmpeq.*t label*	ifge *label*
tableswitch	ifcmpge.*t label*	ifle *label*
{case *low*: goto *lowLabel*	ifcmple.*t label*	ifgt *label*
...	ifcmpgt.*t label*	iflt *label*
case *high*: goto *highLabel*	ifcmplt.*t label*	goto *label*
default: goto *defaultLabel*}	nop	return.*t*
entermonitor	breakpoint	throw.r
exitmonitor	push *constant*	pop.*t*

Figure 2.2: The list of BAF statements.

2.2.3 Design feature: Quantity and quality of bytecodes

One of the headaches in dealing with Java bytecodes directly is the massive number of different instructions present. Upon inspection of the Java Virtual Machine specification, we can see that there are over 201 different bytecodes. BAF, on the other hand, contains only about 60 bytecode instructions. This compaction has been achieved in two ways: (1) by introducing type attributes such as the `.i` and `.l` in `add.i`, `add.l`, and (2) by eliminating multiple forms of the same instruction such as `iload_0`, `iload_1`. This can be achieved because BAF is not concerned with the compactness of the encoding, as were the designers of the Java Virtual Machine specification, but with the compactness of the representation. Thus we can compact twenty different variants of the `load` instruction into one `load` which has two type attributes: a type and a local variable name.

2.2.4 Design feature: No JSR-equivalent instruction

The jump subroutine bytecode (JSR) instruction present in Java bytecode is often very complicated to deal with, because it is essentially an interprocedural feature inserted into a traditionally intraprocedural context. Analyses and transformations on BAF can be simplified considerably by requiring that the BAF instruction set not have a JSR-equivalent instruction. One might think that this means that BAF can not represent a large variety of Java programs which contain JSRs. But in fact, most JSR bytecode instructions can be eliminated through the use of subroutine duplication. The idea is to transform each `jsr x` into a `goto y` where `y` is a duplicate copy of the subroutine `x` with the final `ret` having been transformed into a `goto` to the instruction following the `jsr x`. See figure 2.3 for an example of code duplication in action.

The code growth in worst case is exponential (with respect to the number of nested JSRs), but in practice the technique produces very little code growth because JSRs are not used that much, and the subroutines represent a small fraction of the total amount of code.

2.2.5 Design feature: No constant pool

One of the many encoding issues in Java bytecode is the constant pool. Bytecode instructions must refer to indices in this pool to access fields, methods, classes, and constants, and this constant pool must be tediously maintained. BAF abstracts away the constant pool, and thus it is easier to manipulate BAF code. In textual BAF (when it is written out to a text

```
    ...                         ...
    istore_2                    istore_2
    jsr label1                  goto label1
                            label_ret_1:
    ...
    istore_3                    ...
    jsr label1                  istore_3
                                goto label1
label1:                     label_ret_2:
    invokestatic f
    ret                     label1:
                                invokestatic f
                                goto label_ret1

                            label2:
                                invokestatic f
                                goto label_ret2
```

Figure 2.3: Example of code duplication to eliminate `JSRs`.

file) the method signature or field is written out explicitly. For example:

```
bytecode:invokevirtual #10
Baf:     virtualinvoke <java.io.PrintStream:
                           void println(java.lang.String)>;
```

Or, another example:

```
bytecode: ldc #1
Baf:      push "foo";
```

Internally, these references are represented directly in the code. For example, the BAF instruction `PushInst` has a field modifiable by get/setString().

2.2.6 Design feature: Explicit local variables

BAF also has another comestic advantage over Java bytecode as it has explicit names for its local variables. This allows for more descriptive names when the BAF code is produced from JIMPLE code in which the local variables might already have names. There are two local variables types in BAF, `word` and `dword`. These correspond to 32-bit and 64-bit quantities respectively. We split the local variables into these two categories to simplify code generation, and local variable packing, a technique that we use to minimize the number of variables used. Note that because the local variables have names and not slot numbers then some sort of equivalence must be established between the local variables which contain the initial values such as `this`, or the first parameter, etc.

2.2.7 Design feature: Typed instructions

A large proportion of the bytecodes are typed, in the sense that their effect on the stack is built into the name of the instruction. For example, `iload` loads an integer on the stack and `iadd` adds two integers from the stack. There are, however, a few instructions such as `dup, dup2, swap` which are not typed, and this causes some complications. Consider the following code:

```
...
istore_3
dup2
```

What does the dup2 instruction do? It duplicates the top two 32-bit elements of the stack. Although this operation is easy to implement for a Java virtual machine whose stack representation is a series of 32-bit elements, if you are performing typed analyses on this bytecode you may run into some confusion as to what exactly is occuring with the dup2 instructions. In particular, if you are attempting to convert this code to typed 3-address code, there are two possible conversions, based on the types present on the stack. If the types present on the stack before the `dup2` is a 64-bit quantity then it should be converted to a single copy such as:

```
long $s1, $s0;

...
$s1 = $s0
```

But if the types present before the `dup2` are `ints` say, then you get a conversion such as:

```
int $s3, $s2, $s1, $s0;

...
$s3 = $s1;
$s2 = $s0;
```

So which one do you pick when you perform a conversion to 3-address code? Well that depends on the contents of the stack. Unfortunately, the contents of the stack can not be determined locally by the type of the dup2 because it is untyped. So in order to determine the exact effect of an instruction such as `dup2` on the stack, one must perform an abstract stack simulation. This means that the cumulative effect of each bytecode preceeding the `dup2` must be considered to determine the exact contents of the stack preceeding the `iload`. This is easy to implement but non-trivial because this is a fixed point iteration problem that spans basic blocks.

We avoid having to perform abstract stack simulations on BAF by imposing the following constraint on BAF: each BAF instruction is fully typed; its effect on the stack is fully specified by the type attribute, as in `iload.i` which means load an integer and `dup.f` means duplicate a float.

This constraint simplifies analyses on BAF considerably. Note that in order to create BAF instructions from Java bytecode some abstract stack interpretation must be performed. But at least this is performed by the SOOT framework, and not by analysis writers who work with the BAF code directly. More on this topic in chapter 3.

2.2.8 Design feature: Stack of values

The BAF stack is a stack of values, as opposed to a stack of words. Effectively, all elements on the stack have size 1. See figure 2.4 for an illustration.

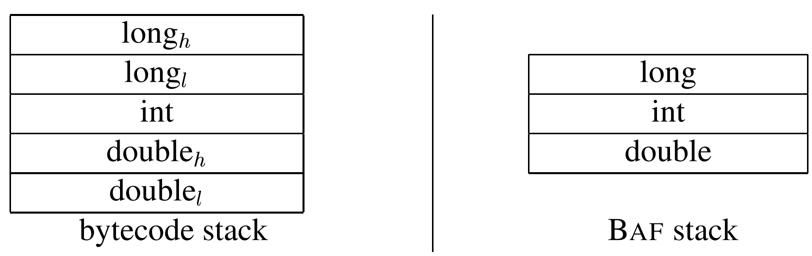

Figure 2.4: The bytecode stack has size 5, and the BAF stack has size 3.

This means that a `dup` instruction is interpreted to duplicate the top element of the stack, no matter what the implementation size of that element is. Similarly, `dup2` duplicates two elements. This allows BAF to be somewhat more expressive than bytecode because `dup2.dl` means duplicate the double and long which are on the stack, a total of 128-bit elements.

The goal of this design feature is also to simplify analyses.

2.2.9 Design feature: Explicit exception ranges

Exceptions in Baf are represented as explicit exceptions ranges using labels. This mirrors the Java bytecode representation as opposed to the Java representation which uses structured try-catch pairs. See figure 2.5 for an illustration of this.

```
try {
    System.out.
       println("try-block");
}
catch(Exception e)
{
    System.out.
       println("exception");
}
```

original Java code

```
   word r0;

   r0 := @this;

label0:
   staticget java.lang.System.out;
   push "try-block";
   virtualinvoke println;

label1:
   goto label3;

label2:
   store.r r0;
   staticget java.lang.System.out;
   push "exception";
   virtualinvoke println;

label3:
   return;

   catch java.lang.Exception from
     label0 to label1 with label2;
```

BAF code

Figure 2.5: Example demonstrating explicit exception ranges.

2.2.10 Walkthrough of example

This subsection highlights the differences explicitly between figures 2.6 and 2.7. Figure 2.6 consists of dissassembled Java bytecode as produced by `javap`, and 2.7 consists of BAF code. Notice:

1. there are local variables in the BAF example which are named `this`, `x`, and `array` and given the type `word`.
2. the instructions with the `:=`. These are the identity instructions which identify which local variables are pre-loaded with meaning, such as `this` or the contents of `parameters`.
3. the pushing of constants in the javap code come in two different forms: `bipush` and `iconst_0`. But in BAF, there is a single instruction which does the job: `push`.
4. The code layout is also interesting because in the javap code all code is referred to by index, but in BAF labels are used which make it much easier to read, and makes the basic blocks in the code more evident.
5. the constant pool has been eliminated in BAF, and that all references to methods and fields are directly inlined into the instructions such as in
   `fieldget <Main: int a>`
6. the clearer names in BAF. For example, an array read is `iaload` in javap code but in BAF is `arrayread.i`.

2.3 JIMPLE

This section describes JIMPLE. The motivation for JIMPLE is given, its description, a list of its design features, and then finally a walkthrough of the example piece of code.

2.3.1 Motivation

Optimizing stack code directly is awkward for multiple reasons, even if the code is in a streamlined form such as BAF. First, the stack implicitly participates in every computation; there are effectively two types of variables, the implicit stack variables and explicit local variables. Second, the expressions are not explicit, and must be located on the stack. These disadvantages were discussed in detail in section 2.1.2.

```
Method int stepPoly(int)
   0 bipush 10
   2 newarray int
   4 astore_2
   5 iload_1
   6 bipush 10
   8 if_icmple 29
  11 aload_2
  12 iconst_0
  13 aload_0
  14 getfield #7 <Field int a>
  17 iastore
  18 getstatic #9 <Field java.io.PrintStream out>
  21 ldc #1 <String "foo">
  23 invokevirtual #10 <Method void println(java.lang.String)>
  26 goto 44
  29 iload_1
  30 iconst_5
  31 if_icmple 44
  34 iload_1
  35 bipush 10
  37 aload_0
  38 getfield #8 <Field int b>
  41 imul
  42 iadd
  43 istore_1
  44 aload_2
  45 iconst_0
  46 iaload
  47 istore_1
  48 iload_1
  49 ireturn
```

Figure 2.6: `stepPoly` in disassembled form, as produced by javap.

```
   word this, x, array;

   this := @this: Test;
   x := @parameter0: int;
   push 10;
   newarray.i;
   store.r array;
   load.i x;
   push 10;
   ifcmple.i label0;

   load.r array;
   push 0;
   load.r this;
   fieldget <Test: int a>;
   arraywrite.i;
   staticget <java.lang.System: java.io.PrintStream out>;
   push "foo";
   virtualinvoke <java.io.PrintStream: void println(java.lang.String)>;
   goto label1;

label0:
   load.i x;
   push 5;
   ifcmple.i label1;

   load.i x;
   push 10;
   load.r this;
   fieldget <Test: int b>;
   mul.i;
   add.i;
   store.i x;

label1:
   load.r array;
   push 0;
   arrayread.i;
   return.i;
```

Figure 2.7: `stepPoly` in BAF form.

```
public void example()
{
    int x;

    if(a)
    {
       String s = "hello";
       s.toString();
    }
    else
    {
        int sum = 5;
        x = sum;
    }
}
```

(a) Original Java code

```
 0 aload_0
 1 getfield #6
 4 ifeq 18
 7 ldc #1
 9 astore_2
10 aload_2
11 invokevirtual #7
14 pop
15 goto 22
18 iconst_5
19 istore_2
20 iload_2
21 istore_1
22 return
```

(b) bytecode

Figure 2.8: Example of type overloading.

A third difficulty is the untyped nature of the stack and of the local variables in the bytecode. For example, in the bytecode in figure 2.8.

The local variable in slot 2 is used in one case as an `java.lang.String` at instructions 9 and 10, but then another situation at instructions 19 and 20 as a `int`. This type overloading can confuse analyses which expect explicitly typed variables.

2.3.2 Description

Jimple is a typed and compact 3-address code representation of bytecode. It is our ideal form for performing optimizations and analyses, both traditional optimizations such as copy propagation and more advanced optimizations such as virtual method resolution that object-oriented languages such as Java require. See figures 2.9 and 2.10 for the complete Jimple grammar. There are essentially 11 different types of Jimple statements:

- The *assignStmt* statement is the most used Jimple instruction. It has four forms: assigning a *rvalue* to a local, or an *immediate* (a local or a constant) to a static field, to an instance field or to an array reference. Note that a *rvalue* is a field access, array reference, an immediate or an expression. We can see that with this grammar any significant computation must be broken down into multiple statements, with *locals* being used as temporary storage locations. For example, a field copy such as `this.x = this.y` must be represented as two Jimple statements, a field read and a field write (`tmp = this.y` and `this.x = tmp`.)

- *identityStmts* are statements which define locals to be pre-loaded (upon method entry) with special values such as parameters or the `this` value. For example, `l0 := @this: A` defines local `l0` to be the `this` of the method. This identification is necessary because the local variables are not numbered (normally the `this` variable is the variable in local variable slot 0 at the bytecode level.)

- *gotoStmt*, and *ifStmt* represent unconditional and conditional jumps, respectively. Note the use of labels instead of bytecode offsets.

- *invokeStmt* represents an invoke without an assignment to a local. (The assignment of a return value is handled by a *assignStmt* with an *invokeExpr* on the right hand side of the assignment operator.)

- *switchStmt* can either be a `lookupswitch` or a `tableswitch`. The lookupswitch takes a set of integers values, whereas `tableswitch` takes a range of integer values for the lookup values, as the `tableswitch` and `lookupswitch` Java bytecodes do. The target destinations are specified by labels.

- *monitorStmt* represents the enter/exitmonitor bytecodes. They take a local or constant as the monitor lock.

- *returnStmt* can either represent a return void, or a return of a value, specified by a local or a constant.

- *throwStmt* represents the explicit throwing of an exception.

- *breakpointStmt* and *nopStmt* represent the `breakpoint` and `nop` (no operation) bytecode instructions respectively.

2.3.3 Design feature: Stackless 3-address code

Every statement in JIMPLE is in 3-address code form. 3-address code is a standard representation where the instructions are kept as simple as possible, and where most of them are of the form `x = y op z`[17].

Every statement in JIMPLE is stackless. Essentially, the stack has been eliminated and replaced by additional local variables. Implicit references to stack positions have been transformed into explicit references to local variables. Figure 2.11 illustrates this transformation of references. Note how the local variables representing stack positions are prefixed with dollar signs. Note also that each instruction in the original BAF code corresponds to a new instruction in the JIMPLE form. Further, this code can be compacted into `x = y + z`. This will be discussed further in the section on transformations.

stmt ⟶ *assignStmt* | *identityStmt* |
gotoStmt | *ifStmt* | *invokeStmt* |
switchStmt | *monitorStmt* |
returnStmt | *throwStmt* |
breakpointStmt | *nopStmt*;

assignStmt ⟶ local = *rvalue*; |
field = *imm*; |
local`.`field = *imm*; |
local`[`*imm*`]` = *imm*;

identityStmt ⟶ local `:=` @this: type; |
local `:=` @parameter*n*: type; |
local `:=` @exception;

gotoStmt ⟶ `goto` label;

ifStmt ⟶ `if` *conditionExpr* `goto` label;

invokeStmt ⟶ `invoke` *invokeExpr*;

switchStmt ⟶ `lookupswitch` *imm*
{`case` value_1: `goto` label_1;
...
`case` value_n: `goto` label_n;
`default:` `goto` defaultLabel;}; |
`tableswitch` *imm*
{`case` low: `goto` lowLabel;
...
`case` high: `goto` highLabel;
`default:` `goto` defaultLabel;}

monitorStmt ⟶ `entermonitor` *imm*; |
`exitmonitor` *imm*;

returnStmt ⟶ `return` *imm*; |
`return`;

throwStmt ⟶ `throw` *imm*;

breakpointStmt ⟶ `breakpoint`;
nopStmt ⟶ `nop`;

Figure 2.9: The JIMPLE grammar (statements)

imm $\longrightarrow$ local | constant
conditionExpr $\longrightarrow$ imm_1 *condop* imm_2 *condop* $\longrightarrow$ $>$ | $<$ | = | $\neq$ | $\leq$ | $\geq$
rvalue $\longrightarrow$ *concreteRef* | *imm* | *expr* *concreteRef* $\longrightarrow$ field | local . field | local [*imm*]
invokeExpr $\longrightarrow$ `specialinvoke` local.m(imm_1, ..., imm_n) | `interfaceinvoke` local.m(imm_1, ..., imm_n) | `virtualinvoke` local.m(imm_1, ..., imm_n) | `staticinvoke` m(imm_1, ..., imm_n)
expr $\longrightarrow$ imm_1 *binop* imm_2 | (type) *imm* | *imm* `instanceof` type | *invokeExpr* | `new` refType | `newarray` (type) [*imm*] | `newmultiarray` (type) [imm_1] ... [imm_n] []* | `length` *imm* | `neg` *imm*
binop $\longrightarrow$ + | - | $>$ | $<$ | = | $\neq$ | $\leq$ | $\geq$ | $*$ | / | $<<$ | $>>$ | $<<<$ | % | `rem` | & | | | `cmp` | `cmpg` | `cmpl`

Figure 2.10: The JIMPLE grammar (support productions)

```
word x, y, z

load.i x
load.i y
add.i
store.i z
```

(a) BAF

```
int $s0, $s1, x, y, z

$s0 = x
$s1 = y
$s0 = $s0 + $s1
z = $s0
```

(b) JIMPLE

Figure 2.11: Example of bytecode to JIMPLE code equivalence.

2.3.4 Design feature: Compact

Java bytecode has about 200 different bytecode instructions, BAF has about 60 and JIMPLE has 19.

JIMPLE's compactness makes it an ideal form for writing analyses and optimizations. The simpler the intermediate representation, the simpler the task of writing optimizations and analyses for it, because fewer test cases cases need to be developed for it. For example, accesses from a field in JIMPLE are always of the form `local = object.f`, whereas in Java, field accesses can occur practically anywhere such as in a method call or arbitrarily nested array references.

2.3.5 Design feature: Typed and named local variables

The local variables in JIMPLE are named and fully typed. They are given a primitive, class or interface type. They are typed for two reasons.

To improve analyses

JIMPLE was designed to facilitate the implementation of optimizations and analyses. Having types for local variables allows subsequent analyses on JIMPLE to be more accurate. For example, class hierarchy analysis, which usually uses just the method signature to determine the possible method dispatches can also use the type of the variable which leads to strictly better answers. For example:

```
Map m = new HashMap();
m.get("key");
```

becomes the following JIMPLE code:

```
java.util.HashMap $r1, r2;

$r1 = new java.util.HashMap;
specialinvoke $r1.<java.util.HashMap: void <init>()>();
r2 = $r1;
interfaceinvoke r2.<java.util.Map:
                    java.lang.Object get(java.lang.Object)>("key");
```

since we know that `r2` is a `java.util.HashMap`, we can in fact statically resolve the `interfaceinvoke` to be a call to `<HashMap: Object get(Object)>`. If we did not know this type, the interfaceinvoke could map to any method which implements the Map interface.

As another example, having typed variables allows a coarse-grained side effect analysis to be performed. For example, take the following code:

```
Chair x;
House y;

x = someChair;
y = someHouse;
```

Suppose one is trying to re-order the assignment statements. If the types of `x` and `y` were unknown, then they could possibly be aliased, and a re-ordering would not be possible. But since in JIMPLE all variables are typed, and assuming that `x`, and `y` are distinct classes and one is not a subclass of another, they can be re-ordered.

To generate code

The second reason that it is necessary for JIMPLE to have typed local variables is because its operators are untyped. Consider the problem of generating BAF code for a Jimple statement such as `x + y`. The code generator must choose one of the following `adds`: (`add.i`, `add.f`, `add.d` or `add.l`) since BAF instructions are fully typed. Having the local variables typed allows this choice to be made without requiring that the operators be typed as well.

2.3.6 Walkthrough of example

This subsection describes the example found in 2.12. Note that:

1. all local variables are declared at the top of the method. They are fully typed; we have reference types such as `Test`, `int[]`, `java.io.PrintStream` and primitive types such as `int`. Variables which represent stack positions have their names prefixed with $.

2. identity statements follow the local variable declarations. This marks the local variables which have values upon method entry.

3. the code resembles simple Java code (hence the term JIMPLE).

4. assignment statements predominate the code.

```
int a;
int b;

public int stepPoly(int)
{
  Test this;
  int x, $i0, $i1, x;
  int[] array;
  java.io.PrintStream $r0;

  this := @this;
  x := @parameter0;
  array = newarray (int)[10];
  if x <= 10 goto label0;

  $i0 = this.a;
  array[0] = $i0;
  $r0 = java.lang.System.out;
  $r0.println("foo");
  goto label1;

label0:
  if x <= 5 goto label1;

  $i0 = this.b;
  $i1 = 10 * $i0;
  x = x + $i1;

label1:
  x = array[0];
  return x;
}
```

(a) JIMPLE

```
int a;
int b;

public int stepPoly(int x)
{
    int[] array = new int[10];

    if(x > 10)
    {
       array[0] = a;
       System.out.println("foo");
    }
    else if(x > 5)
       x = x + 10 * b;

    x = array[0];

    return x;
}
```

(b) Original Java code

Figure 2.12: `stepPoly` in JIMPLE form. Dollar signs indicate local variables representing stack positions.

2.4 GRIMP

This section describes GRIMP. The motivation for GRIMP is given followed by its description, its design features, and finally a walkthrough of an example piece of code.

2.4.1 Motivation

One of the common problems in dealing with intermediate representations is that they are difficult to read because they do not resemble structured languages. In general, they contain many goto's and expressions are extremely fragmented. Another problem is that despite its simple form, for some analyses, 3-address code is harder to deal with than complex structures. For example, we found that generating good stack code was simpler when large expressions were available.

2.4.2 Description

GRIMP is an unstructured representation of Java bytecode which allows trees to be constructed for expressions as opposed to the flat expressions present in JIMPLE. In general, it is much easier to read than BAF or JIMPLE, and for code generation, especially when the target is stack code, it is a much better source representation. It also has a representation for the `new` operator in Java which combines the `new` bytecode instruction with the `invoke-special` bytecode instruction. Essentially, GRIMP looks like a partially decompiled Java code. We are also using GRIMP as the foundation for a decompiler.

See figures 2.13 and 2.14 for the complete GRIMP grammar. Here are the main differences between the JIMPLE and the GRIMP grammar:

- references to *immediates* have been replaced with *objExpr* or *expr*, representing object expressions or general expressions.
- *invokeExpr* can now express a fifth possibility: `newinvoke`. This combines the `new` JIMPLE statement and the call to the constructor (via a `specialinvoke`.)

These differences are discussed further in the following two design features.

2.4.3 Design feature: Expression trees

The main design feature of GRIMP is that references to *immediates* have been replaced with references to *expressions* which can be nested arbitrarily deeply. Figure 2.15 explicitly

stmt $\longrightarrow$ *assignStmt* | *identityStmt* |
gotoStmt | *ifStmt* | *invokeStmt* |
switchStmt | *monitorStmt* |
returnStmt | *throwStmt* |
breakpointStmt | *nopStmt*;

assignStmt $\longrightarrow$ local = *expr*; |
field = *expr*; |
objExpr`.`field = *expr*; |
objExpr`[`*expr*`]` = *expr*;

identityStmt $\longrightarrow$ local `:=` @this: type; |
local `:=` @parameter*n*: type; |
local `:=` @exception;

gotoStmt $\longrightarrow$ `goto` label;

ifStmt $\longrightarrow$ `if` *conditionExpr* `goto` label;

invokeStmt $\longrightarrow$ `invoke` *invokeExpr*;

switchStmt $\longrightarrow$ `lookupswitch` *expr*
{`case` value$_1$: `goto` label$_1$;
...
`case` value$_n$: `goto` label$_n$;
`default:` `goto` defaultLabel;}; |
`tableswitch` *expr*
{`case` low: `goto` lowLabel;
...
`case` high: `goto` highLabel;
`default:` `goto` defaultLabel;}

monitorStmt $\longrightarrow$ `entermonitor` *objExpr*; |
`exitmonitor` *objExpr*;

returnStmt $\longrightarrow$ `return` *objExpr*; |
`return`;

throwStmt $\longrightarrow$ `throw` *objExpr*;

breakpointStmt $\longrightarrow$ `breakpoint`;
nopStmt $\longrightarrow$ `nop`;

Figure 2.13: The GRIMP grammar (statements)

conditionExpr $\longrightarrow$ *expr*$_1$ *condop* *expr*$_2$
condop $\longrightarrow$ $>$ | $<$ | = | $\neq$ | $\leq$ | $\geq$

concreteRef $\longrightarrow$ field |
objExpr`.`field |
objExpr`[`*expr*`]`

invokeExpr $\longrightarrow$ `specialinvoke` local.m(*expr*$_1$, ..., *expr*$_n$) |
`interfaceinvoke` local.m(*expr*$_1$, ..., *expr*$_n$) |
`virtualinvoke` local.m(*expr*$_1$, ..., *expr*$_n$) |
`staticinvoke` m(*expr*$_1$, ..., *expr*$_n$)
`newinvoke` type(*expr*$_1$, ..., *expr*$_n$)

expr $\longrightarrow$ *expr*$_1$ *binop* *expr*$_2$ |
(type) *expr* |
expr `instanceof` type |
`new` refType |
`length` *expr* |
`neg` *expr* |
objExpr |
`constant`

objExpr $\longrightarrow$ *concreteRef* |
(type) *objExpr* |
invokeExpr |
`newarray` (type) [*expr*] |
`newmultiarray` (type) [*expr*$_1$] ... [*expr*$_n$] []*|
local |
nullConstant | stringConstant

binop $\longrightarrow$ + | - | $>$ | $<$ | = | $\neq$ | $\leq$ | $\geq$ | $*$ | / |
$<<$ | $>>$ | $<<<$ | % | `rem` | & | | |
`cmp` | `cmpg` | `cmpl`

Figure 2.14: The GRIMP grammar (support productions)

illustrates this difference. Note that the original Java statement is one line long, but the equivalent GRIMP code is not. This is because GRIMP code, like JIMPLE, only allows for one side effect (memory modification) per statement. Java statements can, on the other hand, modify multiple memory locations such as with `x = y++ + z++` which modifies three locals. We impose this restriction to simplify analyses on GRIMP. This has the unfortunate consequence of not allowing the same compactness as the original Java code to be achieved which affects code generation and code decompilation. This is discussed further in chapter 3.

```
return this.a.x++ +
 m*n*(x*100+10);
```

(a) Java

```
$r0=this.a;
$i0=$r0.x;
$i1=$i0+1;
$r0.x=$i1;
$i2=m*n;
$i1=this.x;
$i1=$i1*100;
$i1=$i1+10;
$i2=$i2*$i1;
$i3=$i0+$i2;
return $i3;
```

(b) Jimple

```
$r0=this.a;
$i0=$r0.x;
$r0.x=$i0+1;
return $i0+m*n*this.x*100+10;
```

(c) Grimp

Figure 2.15: Example of nesting of expressions in Grimp.

2.4.4 Design feature: Newinvoke expression

The second key feature of GRIMP is its ability to represent the Java `new` construct as one expression. For example, the Java code `Object obj = new A(new B());` is represented in JIMPLE by:

```
A $r1, r3;
B $r2;

$r1 = new A;
$r2 = new B;
specialinvoke $r2.<B: void <init>()>();
specialinvoke $r1.<A: void <init>(B)>($r2);
r3 = $r1;
```

Normally, aggregation of expressions is performed by matching single use-defs such as `x = ...;` and `... = x;` Because the `specialinvokes` modify the receiver object without redefining it we can not aggregate `new` and `specialinvoke` statements in this way. This is a problem because `new` statements occur frequently and it is thus highly desirable to be able to aggregate them. To achieve this, we introduce a `newinvoke` expression to the GRIMP grammar to express pairs of the form `x = new Object() ... specialinvoke x.<Object: void <init>()>();`.

2.4.5 Walkthrough of example

This subsection describes the example found in 2.16. Note that:

1. the GRIMP code is extremely similar to the original Java code. The main differences are the unstructured nature of the code (no if-then-else), and the identity statements

2. the statements

   ```
   $i0 = this.<Test: int b>;
   $i1 = 10 * $i0;
   x = x + $i1;
   ```

   from the JIMPLE version in figure 2.12 have been aggregated to x = x + 10 * this.b in the GRIMP form.

2.4.6 Summary

This chapter has presented the three intermediate representations that SOOT provides: BAF, JIMPLE, and GRIMP. Their main design features were discussed, and their respective grammars were given. Each intermediate representation was also illustrated with an example program.

```
int a;
int b;

public int stepPoly(int)
{
    Test this;
    int x, x#2;
    int[] array;

    this := @this;
    x := @parameter0;
    array = newarray (int)[10];
    if x <= 10 goto label0;

    array[0] = this.a;
    java.lang.System.
        out.println("foo");
    goto label1;

 label0:
    if x <= 5 goto label1;

    x = x + 10 * this.b;

 label1:
    x = array[0];
    return x;
}
```

(a) GRIMP

```
int a;
int b;

public int stepPoly(int x)
{
    int[] array = new int[10];

    if(x > 10)
    {
       array[0] = a;
       System.out.println("foo");
    }
    else if(x > 5)
       x = x + 10 * b;

    x = array[0];

    return x;
}
```

(b) Original Java code

Figure 2.16: `stepPoly` in GRIMP form.

Chapter 3

Transformations

This chapter describes the transformations present in SOOT which enable it as an optimization framework. SOOT operates as follows: (see figure 3.1) Classfiles are produced from a variety sources, such as the `javac` or ML compiler. These classfiles are then fed into SOOT. A JIMPLE representation of the classfiles is generated (section 3.1) at which point the optimizations that the developer has written using the SOOT framework are applied. The resulting optimized JIMPLE code must then be converted back to bytecode, via one of two alternatives. The first alternative is covered in section 3.2 and consists of generating GRIMP code which is tree-like code and traversing it. The second alternative, covered in section 3.3, consists of generating naive BAF code which is stack code and then optimizing it. The newly generated classfiles consist of optimized bytecode which can then be fed into one of many destinations such as a Java Virtual Machine for execution.

3.1 Bytecode $\longrightarrow$ Analyzable JIMPLE

This section describes the five steps necessary to convert bytecode to analyzable Jimple code. This is a non-trivial process because the bytecode is untyped stackcode, whereas JIMPLE code is typed 3-address code. The five steps are illustrated in figure 3.2.

Throughout this chapter we use a running example to show how these five steps transform Java code. See figure 3.3 for the original Java code.

3.1.1 Direct translation to BAF with stack interpretation

The first step is to convert the bytecodes to the equivalent BAF instructions. Most of the bytecodes correspond directly to equivalent BAF instructions. For example, as seen in figure 3.4, `iconst_0` corresponds to `push 0`, `istore_2` corresponds to `store.i l2`

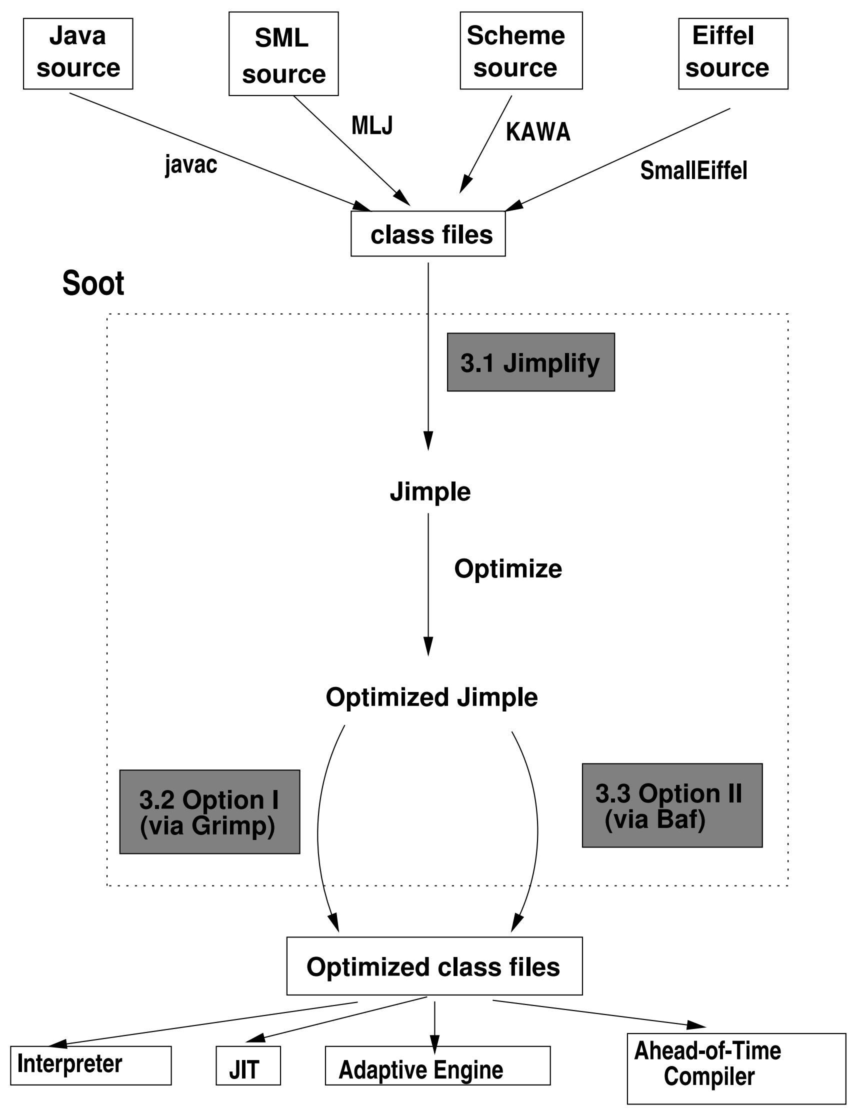

Figure 3.1: An overview of Soot and its usage.

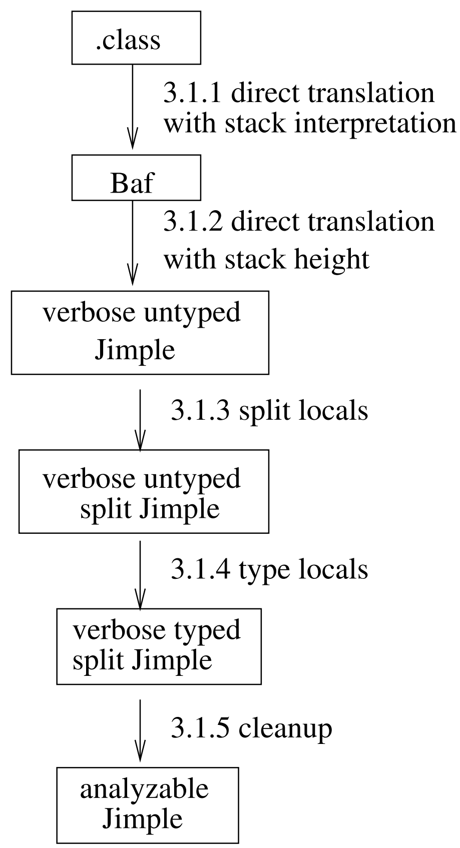

Figure 3.2: Bytecode to JIMPLE in five steps.

```java
public class Test
{
    A a;
    boolean condition;

    public int runningExample()
    {
        A a;
        int sum = 0;

        if(condition)
        {
            int z = 5;
            a = new B();
            sum += z++;
        }
        else
        {
            String s = "four";
            a = new C(s);
        }

        return a.value + sum;
    }
}

class A
{
    int value;
}

class C extends A
{
    public C(String s){}
}

class B extends A
{
}
```

Figure 3.3: The running example in its original Java form.

and so forth. Each untyped local variable slot in the bytecode gets converted to an explicit (but still untyped) local variable in BAF code.

The only difficulty lies in transforming the `dupxxx` class of instructions and the `swap` instruction. These bytecode instructions are untyped, but since BAF instructions are fully typed, we must determine which kind of data is being duplicated or swapped by these instructions before they can be transformed. This can be achieved by performing an abstract stack interpretation, in which we compute the contents of the computation stack after every instruction (abstract stack interpretation is discussed in detail in [16].) Figure 3.4(b) gives the contents of the stack explicitly, and it is used to determine that both `dups` should be converted to `dup.r`, because the top of the stack before each `dup` contains a `ref`.

3.1.2 Direct translation to JIMPLE with stack height

The next phase is to convert each BAF instruction to an equivalent JIMPLE instruction sequence. This is done in three steps:

1. Compute the stack height after each BAF instruction, by performing a depth-first traversal of the BAF instructions. Note that a simple traversal can compute the stack height exactly because the Java Virtual Machine specification [16] guarantees that every program point has a fixed stack height which can be be pre-computed.

2. Create a JIMPLE local variable for every BAF variable, and create a JIMPLE local variable for every stack position (numbered 0 to *maximum stack height - 1*.)

3. Create the equivalent JIMPLE instructions for every BAF instruction, mapping the implicit effect that the BAF instruction has on the stack positions to explicit references to the JIMPLE local variables which represent those stack positions (created in step 2.) For example, the first occurence of `push 0;` becomes `$stack0 = 0;` and both `dup1.r;` instructions become `$stack1 = $stack0;`. Note that an instruction such as `dup2.r` gets translated to multiple JIMPLE instructions, which is necessary to duplicate two stack positions.

This method of transforming bytecode is relatively standard and is also covered in the works of Proebsting *et al* [19] and Muller *et al* [18]. See figure 3.5 to see how the complete running example is transformed.

3.1.3 Split locals

To prepare for typing, the local variables must be split so that each local variable corresponds to one use-def/def-use web, because the JIMPLE code generated by the previous section may be untypable. In particular, in our running example (figure 3.5) we see that on one

```
 0 iconst_0
 1 istore_2
 2 aload_0
 3 getfield #10
   <Field boolean condition>
 6 ifeq 29

 9 iconst_5
10 istore_3
11 new #3
   <Class B>
14 dup
15 invokespecial #7
   <Method B()>
18 astore_1
19 iload_2
20 iload_3
21 iinc 3 1
24 iadd
25 istore_2
26 goto 41


29 ldc #1
   <String "four">
31 astore_3
32 new #4 <Class C>
35 dup
36 aload_3
37 invokespecial #9
   <Method C(java.lang.String)>
40 astore_1


41 aload_1
42 getfield #11
   <Field int value>
45 iload_2
46 iadd
47 ireturn
```

(a) Java bytecode

```
{int}
{}
{ref}
{int}

{}

{int}
{}
{ref}

{ref,ref}
{ref}

{}
{int}
{int,int}
{int,int}
{int}
{}
{}


{ref}

{}
{ref}
{ref,ref}
{ref,ref,ref}
{ref,ref}

{}


{ref}
{int}

{int,int}
{int}
{}
```

(b) stack

```
 push 0;
 store.i l2;
 load.r l0;
 fieldget
  <Test: boolean cond..>;
 ifeq label0;

 push 5;
 store.i l3;
 new B;

 dup1.r;
 specialinvoke
  <B: void <init>()>;
 store.r l1;
 load.i l2;
 load.i l3;
 inc.i l3 1;
 add.i;
 store.i l2;
 goto label1;

label0:
 push "four";

 store.r l3;
 new C;
 dup1.r;
 load.r l3;
 specialinvoke
  <C: void <init>(Str..)>;
 store.r l1;

label1:
 load.r l1;
 fieldget
  <A: int value>;
 load.i l2;
 add.i;
 return.i;
```

(c) Baf code

Figure 3.4: Running example: Java bytecode to Baf code.

```
  word l0, l1,
    l2, l3;

  l0 := @this: Type;
  push 0;
  store.i l2;
  load.r l0;
  fieldget
   <Test: boolean condition>;
  ifeq label0;

  push 5;
  store.i l3;
  new B;
  dup1.r;
  specialinvoke
   <B: void <init>()>;
  store.r l1;
  load.i l2;
  load.i l3;
  inc.i l3 1;
  add.i;
  store.i l2;
  goto label1;

label0:
  push "four";
  store.r l3;
  new C;
  dup1.r;
  load.r l3;
  specialinvoke
   <C: void <init>(String)>;
  store.r l1;

label1:
  load.r l1;
  fieldget
   <A: int value>;
  load.i l2;
  add.i;
  return.i;
```

(a) BAF code

```



1
0
1
1

0

1
0
1
2
1

0
1
2
2
1
0
0


1
0
1
2
3
2

0


1
1

2
1
0
```

(b) stack height

```
  unknown l0, $stack0, l2,
    l3, $stack1, l1, $stack2;

  l0 := @this: Type;
  $stack0 = 0;
  l2 = $stack0;
  $stack0 = l0;
  $stack0 = $stack0.condition;

  if $stack0 == 0 goto label0;

  $stack0 = 5;
  l3 = $stack0;
  $stack0 = new B;
  $stack1 = $stack0;
  specialinvoke
    $stack1.<init>();
  l1 = $stack0;
  $stack0 = l2;
  $stack1 = l3;
  l3 = l3 + 1;
  $stack0 = $stack0 + $stack1;
  l2 = $stack0;
  goto label1;

label0:
  $stack0 = "four";
  l3 = $stack0;
  $stack0 = new C;
  $stack1 = $stack0;
  $stack2 = l3;
  specialinvoke
    $stack1.<init>($stack2);
  l1 = $stack0;

label1:
  $stack0 = l1;
  $stack0 = $stack0.value;

  $stack1 = l2;
  $stack0 = $stack0 + $stack1;
  return $stack0;
```

(c) verbose untyped JIMPLE code

Figure 3.5: Running example: BAF code to verbose untyped JIMPLE code

branch of the if, `l3` is used as an `int` (in `$stack0 = 5; l3 = $stack0;`), and on the other branch, `l3` is used as a `String` (in `$stack0 = "four"; l3 = $stack0;`.)

To split the local variables, we simply compute the webs [17] by traversing the *use-def* and *def-use* chains, and associate one local variable with each produced web. A web is essentially a subset of all the uses and definitions of a particular local variable which are self-contained in the sense that this subset can be renamed without affecting the behavior of the code. Figure 3.6 illustrates this with an example.

Before	After renaming according to webs

`if(condition)`
`    x = 1;`
`else`
`    x = 2;`
 	
`print(x);`
`x = 1;`
`print(x);`
	`if(condition)`
	`    x1 = 1;`
	`else`
	`    x1 = 2;`
	
	`print(x1);`
	`x2 = 1;`
	`print(x2);`

Figure 3.6: Jimple code illustrating webs.

Figure 3.7 gives the running example after the splitting of the locals. Note how the code now has 23 different local variables, whereas the original code had only 7. In particular, `l3` was split into three different webs, and stack position 0 was split 12 times. This is to be expected because stack position 0 is the most used position for performing temporary computations.

With the local variables split in this way, the resulting JIMPLE code tends to be easier to analyze because it inherits some of the disambiguation benefits of SSA[4]. Local variables will have fewer definitions, and most will in fact have a single definition.

Here is an example to illustrate how Jimple code is easier to analyze with the local variables being split:

```
x = toCopy;
use(y);
z = x + y;
print("hello");
use(x);
```

with the variables split, there is a good chance that `toCopy` is only defined once (this can be deteremined easily with a linear sweep of the code.) If this is the case and since `use(x)` has only one reaching definition of `x`, `x=toCopy` can be propagated into `use(x)` without any further checks. Normally, one must perform an *available copies analysis* or check the interleaving statements for redefinitions of `toCopy`. In SOOT this simplification speeds up our analyses and transformations considerably.

```
public int runningExample()
{
    unknown l0, $stack0, l2, l3, $stack1, l1, $stack2, $stack0#2,
        $stack0#3, $stack0#4, $stack0#5, $stack0#6, $stack1#2,
        l3#2, $stack0#7, $stack0#8, l3#3, $stack0#9, $stack1#3,
        $stack0#10, $stack0#11, $stack1#4, $stack0#12;

    l0 := @this: Test;
    $stack0 = 0;
    l2 = $stack0;
    $stack0#2 = l0;
    $stack0#3 = $stack0#2.condition;
    if $stack0#3 == 0 goto label0;

    $stack0#4 = 5;
    l3 = $stack0#4;
    $stack0#5 = new B;
    $stack1 = $stack0#5;
    specialinvoke $stack1.<init>();
    l1 = $stack0#5;
    $stack0#6 = l2;
    $stack1#2 = l3;
    l3#2 = l3 + 1;
    $stack0#7 = $stack0#6 + $stack1#2;
    l2 = $stack0#7;
    goto label1;

 label0:
    $stack0#8 = "four";
    l3#3 = $stack0#8;
    $stack0#9 = new C;
    $stack1#3 = $stack0#9;
    $stack2 = l3#3;
    specialinvoke $stack1#3.<init>($stack2);
    l1 = $stack0#9;

 label1:
    $stack0#10 = l1;
    $stack0#11 = $stack0#10.value;
    $stack1#4 = l2;
    $stack0#12 = $stack0#11 + $stack1#4;
    return $stack0#12;
}
```

Figure 3.7: JIMPLE code of running example, after splitting the local variables (section 3.1.3.)

3.1.4 Type locals

The next step is to give each local variable a primitive, class or interface type. To do this, we invoke the typing algorithm developed by Etienne Gagnon et al. [10]. The general solution to this problem is non-trivial as it is NP-Hard. However, the typing algorithm in SOOT is an efficient multistage typing algorithm based on solving a type constraint system; each stage is attempted in turn to provide a solution, and each is progressively more complex. The first stage is described below, very briefly. See [10] for a complete discussion of the problem and its solution. The first stage consists of building a constraint system represented by a directed graph. These constraints are gathered from individual statements. The constraints of the first five statements of the running example in figure 3.7 follow.

$$
\begin{aligned}
T(\texttt{l0}) &\leftarrow \texttt{Test} \\
T(\$\texttt{stack0}) &\leftarrow \texttt{int} \\
T(\texttt{l2}) &\leftarrow T(\$\texttt{stack0}) \\
T(\$\texttt{stack0\#2}) &\leftarrow T(\texttt{l0}) \\
T(\$\texttt{stack0\#3}) &\leftarrow \texttt{int} \\
\texttt{Test} &\leftarrow T(\$\texttt{stack0\#2})
\end{aligned}
$$

(a) Collected constraints

```
l0 := @this: Test;
$stack0 = 0;
l2 = $stack0;
$stack0#2 = l0;
$stack0#3 =
    $stack0#2.condition;
```

(b) Original statements

$T(\mathrm{l0}) \leftarrow \mathrm{Test}$ corresponds to `l0 := @this: Test;` and indicates that `l0` must be able to contain an instance of class `Test`, that is, be a superclass of `Test`. Note that the statement `$stack0#3 = $stack0#2.condition;` creates two constraints: one on `$stack0#2`, indicating that `$stack0#2` must be a subclass of `Test` since the `condition` field is accessed and the second indicating that `$stack0#3` must be able to contain an `int`.

After the constraints are collected, they are represented as a graph with soft nodes representing the types of variables and hard nodes representing actual types. Based on this graph, cycles are collapsed, and the soft nodes are collapsed into the hard nodes. Based on this scheme we can see that from the constraints that `$stack0#2` and `l0` must be of type `Test`, and `$stack0` `$stack0#3` and `l2` must be of type `int` (assigning an `int` to a local variable means that it must be exactly of type `int`).

If this first stage fails, then the JIMPLE code is transformed by inserting assignment statements and the typing algorithm is repeated. If this second phase fails then casts are inserted as necessary, and the typing algorithm is repeated, with the modification that only the set of constraints corresponding to definitions are collected. The third phase is guaranteed to succeed and produce a typing solution for the local variables. See figure 3.8 for the typed JIMPLE code of the running example.

```
public int runningExample()
{
    Test l0, $stack0#2;
    int $stack0, l2, l3, $stack0#3, $stack0#4, $stack0#6,
        $stack1#2, l3#2, $stack0#7, $stack0#11, $stack1#4,
        $stack0#12;
    B $stack1, $stack0#5;
    A l1, $stack0#10;
    java.lang.String $stack2, $stack0#8, l3#3;
    C $stack0#9, $stack1#3;

    l0 := @this;
    $stack0 = 0;
    l2 = $stack0;
    $stack0#2 = l0;
    $stack0#3 = $stack0#2.condition;
    if $stack0#3 == 0 goto label0;

    $stack0#4 = 5;
    l3 = $stack0#4;
    $stack0#5 = new B;
    $stack1 = $stack0#5;
    specialinvoke $stack1.<init>();
    l1 = $stack0#5;
    $stack0#6 = l2;
    $stack1#2 = l3;
    l3#2 = l3 + 1;
    $stack0#7 = $stack0#6 + $stack1#2;
    l2 = $stack0#7;
    goto label1;

 label0:
    $stack0#8 = "four";
    l3#3 = $stack0#8;
    $stack0#9 = new C;
    $stack1#3 = $stack0#9;
    $stack2 = l3#3;
    specialinvoke $stack1#3.<init>($stack2);
    l1 = $stack0#9;

 label1:
    $stack0#10 = l1;
    $stack0#11 = $stack0#10.value;
    $stack1#4 = l2;
    $stack0#12 = $stack0#11 + $stack1#4;
    return $stack0#12;
}
```

Figure 3.8: JIMPLE code of the running example, after typing the local variables. (section 3.1.4).

3.1.5 Clean up JIMPLE

After the locals have been typed the code itself remains very verbose. The step discussed in this subsection consists of performing some compaction to eliminate the redundant copy statements which are present in the code due to the direct translation from bytecode.

We see from figure 3.8 that there are several statements which can be eliminated. For example, the pair `$stack0 = 0; l2 = $stack0;` can be optimized to `l2 = 0;` Note that, in general, copy propagation and constant propagation are not sufficient to fully eliminate the redundant copy statements. For example, in the code:

```
$x = f.a;
y = $x;
```

we do not have a copy to propagate forward, but instead a copy to propagate backwards. A combination of copy propagation and back copy propagation has been suggested as a solution to this exact problem[21]. We use, instead, the aggregation algorithm developed on GRIMP on JIMPLE code (see subsection 3.2.1). This simulates the back copy propagation phase as well as a limited form of copy propagation by collapsing single def-use pairs. However, we still need to perform a phase of copy propagation afterwards to catch patterns of the form:

```
$x = i0;
use($x);
use($x);
```

which are single def-multiple use n-tuples. These patterns usually originate from the use of `dups` which are used to implement statements with multiple side effects such as `x.f += a[i++]`

Compare figures 3.9 and 3.8. The compacted version has 18 statements whereas the original version has 30. Most of the eliminated references were references to stack variables.

3.2 Analyzable JIMPLE $\longrightarrow$ Bytecode (via GRIMP)

This section describes the first method of transforming JIMPLE code back to bytecode: via GRIMP(see figure 3.10 for an illustration of these two paths.)

A compiler such as `javac` is able to produce efficient bytecode because it has the structured tree representation for the original program, and the stack based nature of the bytecode is particularly well suited for code generation from trees[2]. Essentially, this

```
public int runningExample()
{
    Test l0;
    int l2, l3, $stack0#3, l3#2,
        $stack0#11, $stack0#12;
    A l1;
    B $stack0#5;
    java.lang.String l3#3;
    C $stack0#9;

    l0 := @this;
    l2 = 0;
    $stack0#3 = l0.condition;
    if $stack0#3 == 0 goto label0;

    l3 = 5;
    $stack0#5 = new B;
    specialinvoke $stack0#5.<init>();
    l1 = $stack0#5;
    l3#2 = l3 + 1;
    l2 = l2 + l3;
    goto label1;

 label0:
    l3#3 = "four";
    $stack0#9 = new C;
    specialinvoke $stack0#9.<init>(l3#3);
    l1 = $stack0#9;

 label1:
    $stack0#11 = l1.value;
    $stack0#12 = $stack0#11 + l2;
    return $stack0#12;
}
```

Figure 3.9: JIMPLE code of running example, after cleanup. (section 3.1.5)

method attempts to recover the original structured tree representation, by building GRIMP, an aggregated form of JIMPLE, and then producing stack code by standard tree traversal techniques.

There are two steps necessary for this transformation and they are covered in the following two subsections. This method was mainly developed and added to the framework by Patrick Lam and is described further in our overview paper[28]. A summary of the method is included here for completeness.

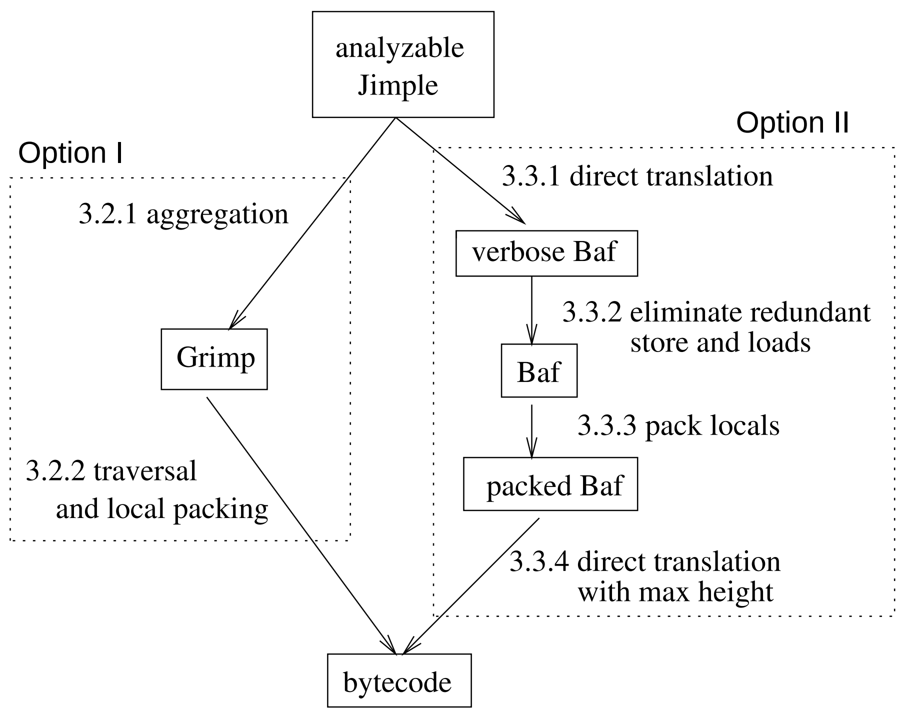

Figure 3.10: Two paths for the JIMPLE to bytecode transformation.

3.2.1 Aggregate expressions

Producing GRIMP code from JIMPLE is relatively straightforward given that GRIMP is essentially JIMPLE with arbitrarily deeply nested expressions and with a newinvoke expression construct. To build GRIMP there are two algorithms which must be applied: expression aggregation, and constructor folding.

1. *Expression aggregation*: for every single def/use pair, attempt to move the right hand side of the definition into the use. Currently, we only consider def-use pairs which

reside in the same extended basic block, but our results indicates that this covers almost all pairs. Some care must be taken to guard against violating data dependencies or producing side effects when moving the right hand side of the definition.

2. *Constructor folding*: pairs consisting of `new` and `specialinvoke` are collapsed into one GRIMP expression called `newinvoke`.

The expression aggregation algorithm is tricky to implement and is described in full detail below in the subsection entitled ”In Detail”.

Note that folding the constructors usually exposes additional aggregation opportunities (namely the aggregation of the `newinvokes`), and these are aggregated in a second aggregation step. See figure 3.11 for the results of performing these three transformations. Note that the definition `$stack0#3=l0` has been aggregated into the statement

```
if l0.condition == 0 goto label0;
```

and that the three statements

```
$stack0#11 = l1.value;
$stack0#12 = $stack0#11 + l2;
return $stack0#12;
```

have been collapsed down to

```
return l1.value + l2;
```

Furthermore, two constructor foldings have taken place, one for `new B` and one for `new C`. Note that the definition `l3#3="four"` has not been aggregated into `new C(l3#3)` because this assignment is present in the original Java code, and we are only aggregating stack variables.

The GRIMP code generated by these three steps is extremely similar to the original Java source code; almost all introduced stack variables are usually eliminated. Statements from Java which have multiple local variable side-effects, however, cannot be represented as compactly, and this complicates bytecode code generation. An example is given in figure 3.12 and this is further discussed in section 3.2.2.

```
public int runningExample()
{
    Test l0;
    int l2, l3, $stack0#3, l3#2,
        $stack0#11, $stack0#12;
    A l1;
    B $stack0#5;
    java.lang.String l3#3;
    C $stack0#9;

    l0 := @this;
    l2 = 0;
    $stack0#3 = l0.condition;
    if $stack0#3 == 0 goto label0;

    l3 = 5;
    $stack0#5 = new B;
    specialinvoke
        $stack0#5.<init>();
    l1 = $stack0#5;
    l3#2 = l3 + 1;
    l2 = l2 + l3;
    goto label1;

 label0:
    l3#3 = "four";
    $stack0#9 = new C;
    specialinvoke
        $stack0#9.<init>(l3#3);
    l1 = $stack0#9;

 label1:
    $stack0#11 = l1.value;
    $stack0#12 = $stack0#11 + l2;
    return $stack0#12;
}
```

JIMPLE code

```
public int runningExample()
{
    Test l0;
    int l2, l3, l3#2;
    A l1;
    java.lang.String l3#3;

    l0 := @this;
    l2 = 0;
    if l0.condition == 0
        goto label0;

    l3 = 5;
    l1 = new B();
    l3#2 = l3 + 1;
    l2 = l2 + l3;
    goto label1;

 label0:
    l3#3 = "four";
    l1 = new C(l3#3);

 label1:
    return l1.value + l2;
}
```

GRIMP code

Figure 3.11: GRIMP code of running example, after aggregation and constructor folding. (see subsection 3.2.1)

```
a[j++] = 5;
```

(a) Java code

```
aload_2
iload_1
iinc 1 1
iconst_5
iastore
```

(b) bytecode of Java code

```
$stack = j;
j = j + 1;
r1[$stack] = 5;
```

(c) equivalent GRIMP code

```
iload_1
istore_2
iinc 1 1
aload 0
iload 2
iconst_5
iastore
```

(d) bytecode of GRIMP code

Figure 3.12: Example of Java code which does not translate to compact GRIMP code.

In Detail

The ideas behind expression aggregation are relatively simple, but in practice we found it difficult to implement it correctly. Thus, we give the complete algorithm in figures 3.13 and 3.14 and explain it here in detail.

Overview: The algorithm consists of considering single use-def pairs and attempting to insert the def into the use, assuring that no data dependencies are violated by this move. The algorithm is a fixed point iteration; it iterates until no more aggregations can be performed. The following points comment on specific portions of the algorithm presented in figures 3.13 and 3.14. Refer to the figures to see the correspondence: each numbered step below corresponds to a labelled step in the figure.

1. Considering the statements in reverse pseudo topological order is desired in order to minimize the number of iterations. For example, in the following code:

   ```
   a = f();
   b = f();
   use(a, b);
   ```

   the first statement can not be aggregated past the second because of potential side effect violations. Thus the second should be aggregated first.

2. Only single use-def pairs are considered for aggregation. These are pairs of the form `x = ...; ...x...;` where there is only one definition of `x`, and only one use of `x`.

3. The use-def pair must be in the same exception context. For example, in the code:

```
try
{
    x = throwSomeException();
} catch(Exception e)
{
    System.out.println("caught!");
    return;
}

System.out.println(x);
```

   The x must not be aggregated out of the `try` block because if an exception is thrown then it would no longer be caught.

4. This section of code simply notes what kind of structures are on the right hand side of `s`.

5. The next step is to consider the path between `s` and `use` and determine if it is legal to move `s` past all those statements into `use`. Note that we consider only pairs of statements which are in the same extended basic block. This guarantees that the connecting path is unique.

6. A redefinition of a local in `s` occurs in the path, preventing aggregation. Here is an example:

```
x = a + b;
a = 1;
use(x);
```

7. This block of code prevents the moving of method calls past field writes, or field reads past field writes (of the same name), or method calls or array reads past array writes.

   It also prevents propagating method calls past EnterMonitorStmt or ExitMonitorStmt.

8. This block of code prevents the re-ordering of method calls with array references, field references or other method calls. Note that (8a) handles the following situation:

```
x = f();
z = 1;
use(x, m(), z);
```

   upon inspection of the statement `use` which in this case is `use(x,m(), z)` the verification of moving `f()` past other method calls stops at the first use of `x` because the arguments are evaluated from left to right.

9. At this point it is safe to aggregate `s` and `use` together.

```
hasChanged = true;

while hasChanged do
    hasChanged = false;

(1) for each statement s in reverse-pseudo-topological or-
der for graph G do
        canAggregate = false;

        if s is an assignment statement and lhs(s) is a local then
            x = lhs(s);

(2)         if s has only one use in G and that use u has s as its sole def
(3)             if s  and use are in the same exception zone
                    canAggregate = true;

        if not canAggregate then
           next statement;

        fieldRefList = emptyList;
        localsUsed = emptyList;
        propagatingInvokeExpr = false;
        propagatingFieldRef = false;
        propagatingArrayRef = false;

        for all values v in s do
            if v instanceof Local
                localsUsed.add(v);
            else if v instanceof InvokeExpr
                propagatingInvokeExpr = true;
(4)         else if v instanceof ArrayRef
                propagatingArrayRef = true;
            else if v instanceof FieldRef
                propagatingFieldRef = true;
                fieldRefList.add(v);

(5)     path = G. extendedBasicBlockPathBetween(s, use);
```

Figure 3.13: The aggregation algorithm. (part I)

```
    if path is null then
        next statement;

    for each node nodeStmt in path do
        if nodeStmt == s then
            next statement;

        if nodeStmt != use and nodeStmt is an assignment then
            def = lhs(nodeStmt);

(6)         if localsUsed.contains(def) then
                next statement;

            if propagatingInvokeExpr or propagatingFieldRef or
                propagatingArrayRef then
                if def instanceof FieldRef then
                    if propagatingInvokeExpr then
                        next statement;
(7)                 if propagatingFieldRef then
                        for f in fieldRefList do
                            if f = def.getField()
                                next statement;
                else if def instanceof ArrayRef then
                    if propagatingInvokeExpr or propagatingArrayRef then
                        next statement;

            if propagatingInvokeExpr and nodeStmt instanceof
               MonitorStmt then
               next statement;

         if propagatingInvokeExpr or propagatingFieldRef or
             propagatingArrayRef then
             for all values y in nodeStmt do
      (8a)       if use = nodeStmt and def == y
                     goto aggregate;
(8)
                 if y instanceof InvokeExpr or
                     (propagatingInvokeExpr and (def instanceof Field-
Ref or
                      def instanceof ArrayRef))
                     next statement;

(9)aggregate:
    aggregate(s, use)
    hasChanged = true;
```

Figure 3.14: The aggregation algorithm (part II)

3.2.2 Traverse GRIMP code and generate bytecode

Generating BAF code from GRIMP is straightforward because GRIMP consists of tree-like statements and BAF is a stack-based representation. Standard code generation techniques for stack machines are used here[2], that is, pre-order tree traversal. In the running example, for example, we have the following conversion:

```
return l1.value + l2;
```

GRIMP code

```
43 aload_2
44 getfield #20 <Field int value>
47 iload_1
48 iadd
49 ireturn
```

bytecode

The code generated in some cases by this tree traversal may be inefficient compared to the original Java bytecode. This occurs when the original Java source contained compact C-like constructs such as a[j++] = 5 in the example 3.12. Note how the bytecode generated from the GRIMP code has two extra bytecodes. This inefficiency may have a significant impact on the program execution time if such a statement occurs in loops (and they often do.)

To eliminate this source of inefficiency we perform peephole optimizations on the code generated from GRIMP. To optimize the increment case, we search for Grimp patterns of the form:

```
s1:    local = <lvalue>;
s2:    <lvalue> = local/<lvalue> + 1;
s3:    use(local)
```

and we ensure that the local defined in s_1 has exactly two uses, and that the uses in s_2, s_3 have exactly one definition. Given this situation, we emit code for only s_3. However, during the generation of code for s_3, when local is to be emitted, we also emit code to duplicate local on the stack, and increment <lvalue>.

This approach produces reasonably efficient bytecode. In some situations the peephole patterns fail and the complete original structure is not recovered. In these cases, the second option of producing bytecode via BAF performs better. See the chapter 4 for more details.

Before performing the tree traversal, we also perform a phase of register allocation which maps the GRIMP local variables to bytecode local variable slots. This mapping is performed twice; once for 32-bit quantities and another time for 64-bit quantities. The register allocation scheme is a simple scheme based on heuristic coloring which uses intereference graphs to prevent conflicts.

`$stack0#11 = l1.value;`	`load.r l1;`
	`fieldget <A: int value>;`
	`store.i $stack0#11;`

`$stack0#12 = $stack0#11 + l2;`	`load.i $stack0#11;`
	`load.i l2;`
	`add.i;`
	`store.i $stack0#12;`

`return $stack0#12;`	`load.i $stack0#12;` `return.i;`
(a) original JIMPLE code	(b) inefficient BAF code

Figure 3.15: Excerpt of the running example explicitly demonstrating the JIMPLE to BAF conversion statement by statement.

3.3 Analyzable JIMPLE to Bytecode (via BAF)

This section describes the second method of transforming JIMPLE code back to bytecode: via BAF. (see figure 3.10 for an illustration of these two paths.)

This method attempts to achieve efficient bytecode by producing BAF code naively and then optimizing it, as opposed to the GRIMP method which attempts to produce efficient BAF code directly.

There are four steps necessary for this transformation, and they are described in the following subsections. This method was mainly developed and added to the framework by Patrice Pominville and is described further in our overview paper [28]. It is included here for completeness.

3.3.1 Direct translation to BAF

The first step to produce bytecode from JIMPLE is to treat JIMPLE as a tree representation and convert it to BAF code directly, using standard tree traversal techniques as was done in section 3.2.2.

Generating code in this manner produces inefficient BAF code because local variables are used for the storage of intermediate results, instead of using the stack. For example, an excerpt of the running example is given in figure 3.15 explicitly showing the conversion.

We can see in figure 3.15 that the `store.i`, `load.i` pairs for variables `$stack0#11` and `$stack0#12` can be eliminated, because their values are just temporarily left on the stack. The next step of eliminating redundant load/store seeks to optimize this situation as well as others. The complete translation can be found in figure 3.16.

```
public int runningExample()
{
    Test l0;
    int l2, l3, $stack0#3, l3#2,
        $stack0#11, $stack0#12;
    A l1;
    B $stack0#5;
    java.lang.String l3#3;
    C $stack0#9;

    l0 := @this;
    l2 = 0;
    $stack0#3 = l0.condition;
    if $stack0#3 == 0 goto label0;

    l3 = 5;
    $stack0#5 = new B;
    specialinvoke
        $stack0#5.<init>();
    l1 = $stack0#5;
    l3#2 = l3 + 1;
    l2 = l2 + l3;
    goto label1;

 label0:
    l3#3 = "four";
    $stack0#9 = new C;
    specialinvoke
        $stack0#9.<init>(l3#3);
    l1 = $stack0#9;

 label1:
    $stack0#11 = l1.value;
    $stack0#12 = $stack0#11 + l2;
    return $stack0#12;
}
```

JIMPLE code

```
public int runningExample()
{
    word l0, l2, l3, l1, $stack0#3,
        $stack0#5, l3#2, l3#3, $stack0#9,
        $stack0#11, $stack0#12;

    l0 := @this: Test;
    push 0;
    store.i l2;
    load.r l0;
    fieldget <Test: boolean condition>;
    store.i $stack0#3;
    load.i $stack0#3;
    ifeq label0;

    push 5;
    store.i l3;
    new B;
    store.r $stack0#5;
    load.r $stack0#5;
    specialinvoke <B: void <init>()>;
    load.r $stack0#5;
    store.r l1;
    load.i l3;
    push 1;
    add.i;
    store.i l3#2;
    load.i l2;
    load.i l3;
    add.i;
    store.i l2;
    goto label1;

 label0:
    push "four";
    store.r l3#3;
    new C;
    store.r $stack0#9;
    load.r $stack0#9;
    load.r l3#3;
    specialinvoke
        <C: void <init>(java.lang.String)>;
    load.r $stack0#9;
    store.r l1;

 label1:
    load.r l1;
    fieldget <A: int value>;
    store.i $stack0#11;
    load.i $stack0#11;
    load.i l2;
    add.i;
    store.i $stack0#12;
    load.i $stack0#12;
    return.i;
}
```

After direct translation

Figure 3.16: BAF code of running example, before and after direct translation from JIMPLE. (section 3.3.1))

3.3.2 Eliminate redundant store/loads

After the naive BAF code is generated, it remains to be optimized. In particular, we must remove all the redundant store/load instructions which were introduced by the previous step. Although in theory there are many different patterns of redundant store/load instructions possible, in practice there are just a few which account for the majority:

store/load : a store instruction followed by a load instruction referring to the same local variable with no other uses. Both the store and load instructions can be eliminated, and the value will simply remain on the stack.

store/load/load : a store instruction followed by 2 load instructions, all referring to the same local variable with no other uses. The 3 instructions can be eliminated and a `dup` instruction introduced. The dup instruction replaces the second load by duplicating the value left on the stack after eliminating the store and the first load.

Eliminating redundant patterns is trivial when all the relevant instructions follow each other. If there are interleaving instructions (as is the case with `load.r l3#3`, `store.r l3#3` in the running example in figure 3.17) then some care must be taken to eliminate the pair safely. In particular, we compute the *minimum stack height variation* and *net stack height variation* for these interleaving instructions and only if these both are equal to zero can we eliminate the pair (or triple). If they are not zero, then some reordering of the BAF instructions is attempted. These eliminations are performed on basic blocks, and iterations are performed until there are no more changes. This optimization is discussed in complete detail in our overview paper[28].

In the running example in figure 3.17, the examples of load/store elimination are clear: three store/load triplets on the local variables `$stack0#3`, `$stack0#11` and `$stack0#12` and two store/load/load triplets on the local variables `$stack0#5` and `$stack0#9`.

3.3.3 Pack local variables

In bytecode, the local variables are untyped, whereas in BAF there are two types (`word` and `dword`). This step consists of performing a form of register allocation which attempts to minimize the number of local variables used at the BAF level.

The allocation scheme that we use is based on a greedy graph coloring using interference graphs to assure that locals with overlapping lifespans are given distinct colors. Colors are then used as the basis for new variable names. Two packings are actually performed, one for variables of type `word` and another for variables of type `dword`. This prevents the interchange of 64-bit wide local variable slots with two 32-bit wide local variable slots.

```
public int runningExample()
{
    word l0, l2, l3, l1, $stack0#3,
      $stack0#5, l3#2, l3#3,
      $stack0#9, $stack0#11,
      $stack0#12;

    l0 := @this: Test;
    push 0;
    store.i l2;
    load.r l0;
    fieldget
        <Test: boolean condition>;
    store.i $stack0#3;
    load.i $stack0#3;
    ifeq label0;

    push 5;
    store.i l3;
    new B;
    store.r $stack0#5;
    load.r $stack0#5;
    specialinvoke
        <B: void <init>()>;
    load.r $stack0#5;
    store.r l1;
    load.i l3;
    push 1;
    add.i;
    store.i l3#2;
    load.i l2;
    load.i l3;
    add.i;
    store.i l2;
    goto label1;

 label0:
    push "four";
    store.r l3#3;
    new C;
    store.r $stack0#9;
    load.r $stack0#9;
    load.r l3#3;
    specialinvoke <C: void
        <init>(java.lang.String)>;
    load.r $stack0#9;
    store.r l1;

 label1:
    load.r l1;
    fieldget
        <A: int value>;
    store.i $stack0#11;
    load.i $stack0#11;
    load.i l2;
    add.i;
    store.i $stack0#12;
    load.i $stack0#12;
    return.i;
}
```

before load/store elimination

```
public int runningExample()
{
    word l0, l2, l1, l3#2;

    l0 := @this: Test;
    push 0;
    store.i l2;
    load.r l0;
    fieldget
        <Test: boolean condition>;
    ifeq label0;

    load.i l2;
    push 5;
    dup1.i;
    new B;
    dup1.r;
    specialinvoke
        <B: void <init>()>;
    store.r l1;
    push 1;
    add.i;
    store.i l3#2;
    add.i;
    store.i l2;
    goto label1;

 label0:
    new C;
    dup1.r;
    push "four";
    specialinvoke <C: void
        <init>(java.lang.String)>;
    store.r l1;

 label1:
    load.r l1;
    fieldget
        <A: int value>;
    load.i l2;
    add.i;
    return.i;
}
```

after load/store elimination

Figure 3.17: BAF code of running example, before and after load store elimination.

For example, in figure 3.18 we see that when `l0` is used `i0` and `i1` are no longer needed. But in practice, separating the cases into two types produces reasonably small sets of local variables which are acceptable.

```
word r0, i0, i1;
dword l0;

r0 := @this: Test2;
push 0;
store.i i0;
push 1;
store.i i1;
load.i i0;
staticinvoke <Test2: void useInt(int)>;
load.i i1;
staticinvoke <Test2: void useInt(int)>;
push 1L;
store.l l0;
load.l l0;
staticinvoke <Test2: void useLong(long)>;
return;
```

Figure 3.18: Example of BAF code which could profit from untyped local variable coloring; when `l0` is used, `i0` and `i1` are no longer needed.

In our running example (figure 3.19), we see that variables `l3#2` and `l1` are mapped to `l2` and `l0` respectively, thus saving two local variables.

3.3.4 Direct translation and calculate maximum height

The Java Virtual Machine requires that the maximum stack height be given for each method. This can be computed by performing a simple depth first traversal of the BAF code and recording the accumulated effect that each Baf instruction has on the stack height.

Every BAF instruction is then converted to the corresponding bytecode instruction. This is a straightforward mapping which consists of two steps:

1. *Map the local variables.* Since the local variables are already packed we simply associate each BAF local variable with a Java bytecode variable. Note that specific local variables in the Java bytecode have special meaning, and these are allocated by parsing the BAF code for identity instructions which associate specific local variables with special roles, such as `l0 := @this: Test` which indicates that `l0` corresponds to `@this` and thus should be assigned to local variable slot 0.

2. *Map the bytecode instructions.* Each BAF instruction corresponds to one or more bytecode instructions. This is a straightforward association. For example, `push 0`

```
public int runningExample()
{
    word l0, l2, l1, l3#2;

    l0 := @this: Test;
    push 0;
    store.i l2;
    load.r l0;
    fieldget
      <Test: boolean condition>;
    ifeq label0;

    load.i l2;
    push 5;
    dup1.i;
    new B;
    dup1.r;
    specialinvoke
      <B: void <init>()>;
    store.r l1;
    push 1;
    add.i;
    store.i l3#2;
    add.i;
    store.i l2;
    goto label1;

 label0:
    new C;
    dup1.r;
    push "four";
    specialinvoke <C: void
      <init>(java.lang.String)>;
    store.r l1;

 label1:
    load.r l1;
    fieldget <A: int value>;
    load.i l2;
    add.i;
    return.i;
}
```

Before packing

```
public int runningExample()
{
    word l0, l2;

    l0 := @this: Test;
    push 0;
    store.i l2;
    load.r l0;
    fieldget
      <Test: boolean condition>;
    ifeq label0;

    load.i l2;
    push 5;
    dup1.i;
    new B;
    dup1.r;
    specialinvoke
      <B: void <init>()>;
    store.r l0;
    push 1;
    add.i;
    store.i l2;
    add.i;
    store.i l2;
    goto label1;

 label0:
    new C;
    dup1.r;
    push "four";
    specialinvoke <C: void
      <init>(java.lang.String)>;
    store.r l0;

 label1:
    load.r l0;
    fieldget <A: int value>;
    load.i l2;
    add.i;
    return.i;
}
```

After packing

Figure 3.19: BAF code of running example before and after local packing. (subsection 3.3.3)

corresponds to `iconst_0` and `load.i l0` may correspond to `iload 0` depending on the mapping of the local variables.

3.4 Summary

This chapter described the transformations present in SOOT which allow code in one intermediate representation to be transformed to intermediate representation. We presented the steps required to transform bytecode to JIMPLE, and then JIMPLE back to bytecode via two different paths.

Chapter 4

Experimental Results

Here we present the results of two experiments. The first experiment, discussed in section 4.3, validates that we can pass class files through the framework, without optimizing the JIMPLE code, and produce class files that have the same performance as the original ones. In particular, this shows that our methods of converting from JIMPLE to stack-based bytecode are acceptable. The second experiment, discussed in Section 4.4, shows the effect of applying method inlining on JIMPLE code and demonstrates that optimizing Java bytecode is feasible and desirable.

4.1 Methodology

All experiments were performed on dual 400Mhz Pentium II™ machines. Two operating systems were used, Debian GNU/Linux (kernel 2.2.8) and Windows NT 4.0 (service pack 5). Under GNU/Linux we ran experiments using three different configurations of the Blackdown Linux JDK1.2, pre-release version 2.[1] The configurations were: interpreter, Sun JIT, and a public beta version of Borland's JIT[2]. Under Windows NT, two different configurations of Sun's JDK1.2.2 were used: the JIT, and HotSpot (version 1.0.1)

Execution times were measured by running the benchmarks ten times, discarding the best and worst runs, and averaging the remaining eight. All executions were verified for correctness by comparing the output to the expected output.

[1] http://www.blackdown.org
[2] http://www.borland.com

	# JIMPLE Stmts	Linux Sun Int. (secs)	Linux Sun JIT	Linux Bor. JIT	NT Sun JIT	NT Sun Hot.
compress	7322	440.30	.15	.14	**.06**	.07
db	7293	259.09	.56	.58	.26	**.14**
jack	16792	151.39	.43	.32	**.15**	.16
javac	31054	137.78	.52	.42	**.24**	.33
jess	17488	109.75	.45	.32	.21	**.12**
jpat-p	1622	47.94	1.01	.96	.90	**.80**
mpegaudio	19585	368.10	.15	-	**.07**	.10
raytrace	10037	121.99	.45	.23	.16	**.12**
schroeder-s	9713	48.51	.64	.62	.19	**.12**
soot-c	42107	85.69	.58	.45	**.29**	.53
average	.	.	.49	.45	.25	.25
std. dev.	.	.	.23	.23	.23	.23

Figure 4.1: Benchmarks and their characteristics.

4.2 Benchmarks and Baseline Times

The benchmarks used consist of seven of the eight standard benchmarks from the SPECjvm98[3] suite, plus three additional applications from our collection. See figure 4.1. We discarded the *mtrt* benchmark from our set because it is essentially the same benchmark as *raytrace*. The program *soot-c* is a benchmark based on an older version of SOOT, and is interesting because it is heavily object oriented. The program *schroeder-s* is an audio editing program which manipulates sound files, and *jpat-p* is a protein analysis tool.

Figure 4.1 also gives basic characteristics such as size, and running times on the five platforms. All of these benchmarks are real world applications that are reasonably sized, and they all have non-trivial execution times. We used the Linux interpreter as the base time, and all the fractional execution times are with respect to this base.

Benchmarks for which a dash is given for the running time indicates that the benchmark failed validity checks. In all these cases, the virtual machine is to blame as the programs run correctly with the interpreter with the verifier explicitly turned on. Arithmetic averages and standard deviations are also given, and these automatically exclude those running times which are not valid.

For this set of benchmarks, we can draw the following observations. The Linux JIT

[3]http://www.spec.org/

is about twice as fast as the interpreter but it varies widely depending on the benchmark. For example, with *compress* it is more than six times faster, but for a benchmark like *schroeder-s* it is only 56% faster. The NT virtual machines also tend to be twice as fast as the Linux JIT. Furthermore, the performance of the HotSpot performance engine seems to be, on average, not that different from the standard Sun JIT. Perhaps this is because the benchmarks are not long running server side applications.

4.3 Straight through SOOT

Figure 4.2 compares the effect of processing applications with SOOT with BAF and GRIMP, without performing any optimizations. Fractional execution times are given, and these are with respect to the original execution time of the benchmark for a given platform. The ideal result is 1.00. This means that the same performance is obtained as the original application. For *javac* the ratio is .98 which indicates that *javac*'s execution time has been reduced by 2%. The benchmark *raytrace* has a ratio of 1.02 which indicates that it was made slightly slower; its execution time has been increased by 2%. The ideal arithmetic averages for these tables is 1.00 because we are trying to simply reproduce the program as is. The ideal standard deviation is 0 which would indicate that the transformation has a consistent effect, and the results do not deviate from 1.00.

On average, using BAF tends to reproduce the original execution time. Its average is lower than GRIMP's, and the standard deviation is lower as well. For the faster virtual machines (the ones on NT), this difference disappears. The main disadvantage of GRIMP is that it can produce a noticeable slowdown for benchmarks like *compress* which have tight loops on Java statements containing side effects, which it does not always catch.

Both techniques have similar running times, but implementing GRIMP and its aggregation is conceptually simpler. In terms of code generation for Java virtual machines, we believe that if one is interested in generating code for slow VMs, then the BAF-like approach is best. For fast VMs, or if one desires a simpler compiler implementation, then GRIMP is more suitable.

4.4 Optimization via Inlining

We have selected to investigate the feasibility of optimizing Java bytecode by implementing method inlining. Our approach is simple. We build an invoke graph using class hierarchy analysis[7] and inline method calls whenever they resolve to one method. Our inliner is a bottom-up inliner, and attempts to inline all call sites subject to the following restrictions: 1) the method to be inlined must contain less than 20 JIMPLE statements, 2) no method may

	BAF					GRIMP				
	Linux			NT		Linux			NT	
	Sun Int.	Sun JIT	Bor. JIT	Sun JIT	Sun Hot.	Sun Int.	Sun JIT	Bor. JIT	Sun JIT	Sun Hot.
compress	1.01	1.00	.99	.99	1.00	1.07	1.02	1.04	1.00	1.01
db	.99	1.01	1.00	1.00	1.00	1.01	1.05	1.01	1.01	1.02
jack	1.00	1.00	1.00	-	1.00	1.01	.99	1.00	-	1.00
javac	1.00	.98	1.00	1.00	.97	.99	1.03	1.00	1.00	.95
jess	1.02	1.01	1.04	.99	1.01	1.01	1.02	1.04	.97	1.00
jpat-p	1.00	.99	1.00	1.00	1.00	.99	1.01	1.01	1.00	1.00
mpegaudio	1.05	1.00	-	-	1.00	1.03	1.00	-	-	1.01
raytrace	1.00	1.02	1.00	.99	1.00	1.01	1.00	.99	.99	1.00
schroeder-s	.97	1.01	-	1.03	1.01	.98	.99	-	1.03	1.00
soot-c	.99	1.00	1.02	.99	1.03	1.00	1.01	1.00	1.01	1.01
average	1.00	1.00	1.01	1.00	1.00	1.01	1.01	1.01	1.00	1.00
std. dev.	.02	.01	.01	.01	.01	.02	.02	.02	.02	.02

Figure 4.2: The effect of processing classfiles with SOOT using BAF or GRIMP, without optimization.

contain more than 5000 JIMPLE statements, and 3) no method may have its size increase more than by a factor of 3.

After inlining, the following traditional intraprocedural optimizations are performed to maximize the benefit from inlining,

- copy propagation
- constant propagation and folding
- conditional and unconditional branch folding
- dead assignment elimination
- unreachable code elimination

These are described in [2] and were implemented in SOOT using the SOOT API.

Figure 4.3 gives the result of performing this optimization. The numbers presented are fractional execution times with respect to the original execution time of the benchmark for a given platform. For the Linux virtual machines, we obtain a significant improvement in speed. In particular, for the Linux Sun JIT, the average ratio is .92 which indicates that the average running time is reduced by 8%. For raytrace, the results are quite significant, as we obtain a ratio of .62, a reduction of 38%.

For the virtual machines under NT, the average is 1.00 or 1.01, but a number of benchmarks experience a significant improvement. For example, under the Sun JIT, *raytrace* yields a ratio of .89, and under HotSpot, *javac*, *jack* and *mpegaudio* yield significant improvements. Given that HotSpot itself performs dynamic inlining, this indicates that our static inlining heuristics sometimes capture opportunities that HotSpot does not. Our heuristics for inlining were also tuned the Linux VMs, and future experimentation could produce values which are better suited for the NT virtual machines.

These results are highly encouraging as they strongly suggest that a significant amount of improvement can be achieved by performing aggressive optimizations which are not performed by the virtual machines.

	Linux			NT	
	Sun Int.	Sun JIT	Bor. JIT	Sun JIT	Sun Hot.
compress	1.01	.78	1.00	1.01	.99
db	.99	1.01	1.00	1.00	1.00
jack	1.00	.98	.99	-	.97
javac	.97	.96	.97	1.11	.93
jess	.93	.93	1.01	.99	1.00
jpat-p	.99	.99	1.00	1.00	1.00
mpegaudio	1.04	.96	-	-	.97
raytrace	.76	.62	.74	.89	1.01
schroeder-s	.97	1.00	.97	1.02	1.06
soot-c	.94	.94	.96	1.03	1.05
average	.96	.92	.96	1.01	1.00
std. dev.	.07	.12	.08	.06	.04

Figure 4.3: The effect of inlining with class hierarchy analysis.

Chapter 5

The API

5.1 Motivation

Much of the effort in designing SOOT was spent defining (and revising) the application programming interface (API). In particular, we wanted to design an API with the following attributes:

Useable The API should be structured in a way which is natural and easy to use. The complexity of the base system should be kept at a minimum, since in compiler work there is already a great deal of complexity.

Extendable The API should be structured such that it is easily extended, in the sense that additional concepts can be added without interfering with the concepts already present.

General The API should allow as much code re-use as possible. In particular, we wanted an API which allows analyses and transformations to be performed on code without knowing the specifics of the intermediate representation or as little as possible.

We believe that we have achieved our goals with the API presented in this chapter. We have created an API which we use at McGill University and which is being used by other institutions as well. We hope it will be adopted by more research groups which will enable the widespread sharing of code and experimental results to further the state of research on optimizing Java bytecode.

This chapter is organized as follows. First, we explain a few fundamental concepts relating to the overall API, and then we describe each concept of the API in detail. Finally, we give five example programs written with SOOT and provides simple walkthroughs of the code.

5.2 Fundamentals

This section describes two concepts in SOOT which are not specific to SOOT per se, but are important to understand to use SOOT.

5.2.1 Value factories

SOOT makes heavy use of the *value factory* design pattern which is defined here. This pattern is similar to the factory pattern [11] except we add the additional restriction that the instances returned are unmodifiable. This makes the values more important then the actual instances.

The standard name of `v()` is used for the factory method which generates the instances. The `v` stands for value. Value factories are used in at least two different ways: to implement types and to implement constants. For example, `RefType.v("java.lang.String")` refers to the reference type for strings, and `IntConstant.v(5)` refers to the integer constant 5.

Note that the singleton pattern is a special case of the value factory pattern that takes no arguments, and which has the property that the same object is guaranteed to be returned each time.

5.2.2 Chain

The `Chain` is probably the most useful basic data structure in SOOT. It combines the functionality of the `List` and the `Set` to provide a natural representation of an ordered collection of unique elements. To argue the necessity of the `Chain`, let us consider representing a list of statements in a Java method with a `List`. The two standard implementations of `List` are `ArrayList` and `LinkedList`. Both of these implementations are inadequate because they provide worst-case linear time `contains(Object)` and `remove(Object)`, the latter being used very frequently in order to delete arbitrary elements, for example, when performing dead code elimination.

To achieve our goals of constant time `add(Object)`, `remove(Object)`, and `contains(Object)` methods, we define the `Chain` to be essentially an ordered collection of elements which are guaranteed to be unique. The `HashChain` is our default implementation of the `Chain`. It is essentially a `LinkedList` augmented with a `HashMap`. By guaranteeing that the elements are unique, the `HashMap` can contain a mapping from element to the link node in the `LinkedList` which allows for a constant time implementation of `contains` and `remove`. See figure 5.1 for an illustration.

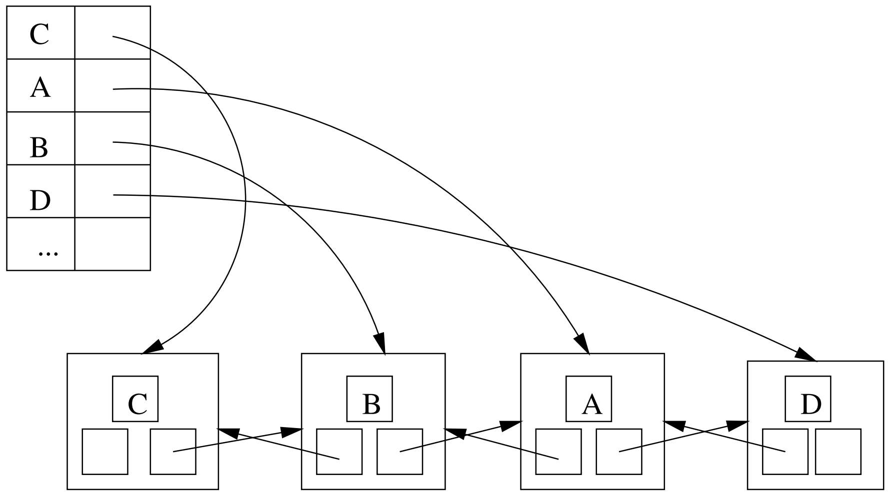

Figure 5.1: An example of a `Chain` and its implementation using a doubly linked list and a hash table.

In addition to being a nice representation for the contents of method bodies in SOOT, `Chains` are also convenient for implementing many algorithms in compilers. A prime example of this is the worklist algorithm for performing data flow analysis. In this algorithm, one must store a list of nodes to visit in some sort of list. When visiting a particular node, however, one must add all the successors of that node to the list of nodes to visit. It is preferable to not add the same node twice, thus one must first check for containment which is a linear time operation on a regular list, or alternatively, one must manually maintain an additional `HashSet` with provides an alternative representation of the nodes to visit with constant time `contains()`.

We have also noticed that one often wants to modify a `Chain` as it is being traversed with an iterator. In general, modifying a collection while iterating through it is bad practice because of the confusing consequences it can have on the current iterators. For this reason, if this occurs, a `ConcurrentModificationException` is thrown. We do provide, however, a method called `snapshotIterator()` which copies the contents of the `Chain` to a buffer, and returns an iterator of the buffer. This allows one to modify the `Chain` while iterating through a static copy of it.

5.3 API Overview

This section gives an overview of the application programming interface (API). Only the subset required to understand the five examples at the end of the chapter is given. The complete API, which is quite extensive, can be found online on our web site[22].

5.3.1 Scene

In SOOT the application which is being analyzed is represented by a `Scene` object. In fact, we have made the simplifying assumption that only one application will be analyzed at a time and so the `Scene` object is represented by a singleton accessed through the static method `Scene.v()`. The `Scene` provides the following functionality.

`SootClass loadClassAndSupport(String className)`
: Loads the given class and resolves all the classes necessary to support that class, that is, the transitive closure of all the references of classes within the class. If the class cannot be found, it returns a phantom class reference (phantom classes are defined in the next subsection.)

`Chain getApplicationClasses(), Chain getLibraryClasses(), Chain getContextClasses(), Chain getPhantomClasses()`
: Returns a backed `Chain` of application, library, context or phantom classes in this scene. These four types of classes are described in the next subsection.

`SootClass getSootClass(String className)`
: Attempts to find and return the given class within the `Scene`. Throws an instance of `RuntimeException` if the class can not be found.

`void addClass(SootClass class)`
: Adds the given class to the `Scene`. Throws an instance of `RuntimeException` if the class is already present.

`SootMethod getMethod(String methodSignature)`
: Returns the `SootMethod` corresponding to the given method signature. Throws a runtime exception if the method can not be found. Method signatures are defined as follows:

$$< className : returnType\ methodName(paramType_1, \ldots, paramType_n) >$$

For example:

```
<java.lang.String: java.lang.String valueOf(char[], int, int)>
```

is the signature for the method `valueOf()` in the class `java.lang.String` which takes an array of characters, an integer, and an integer and returns a `String`.

`SootField getField(String fieldSignature)`
: Returns the `SootField` corresponding to the given field signature. Throws a runtime exception if the field can not be found. Field signatures are defined as follows:

$$< className : \textit{type } fieldName >$$

For example:

$$< java.lang.String : \textit{long } serialVersionUID >$$

is a field named `serialVersionUID` in `String`.

`SootClass getMainClass()`
: Returns the `SootClass` which contains the entry point for the application being analyzed. Throws a null pointer exception if there is no main class set.

`Pack getPack(String packName)`
: Returns the `Pack` of transformations by this name. A `Pack` is list of `Transforms` which encapsulate transformations.

5.3.2 SootClass

Individual classes in the `Scene` are represented by instances of the `SootClass` class. Note that this includes interfaces as well. A `SootClass` contains all the relevant information about a Java class, such as:

`new SootClass(String name, int modifiers)`
: Creates a `SootClass` with the given name and the set of modifiers. Modifiers are described in section 5.3.11.

`int getModifiers()/void setModifiers(int mods)`
: Sets the modifiers for this class.

`int getName()/void setName(String name)`
: Sets the name for this class.

`Chain getInterfaces()`
: Returns a backed [1] `Chain` of interfaces which are implemented by this `SootClass`.

[1]A backed `Chain` is a Chain which, if modified will modify the collection of objects that it represents. Modifying this `Chain` will modify the set of interfaces for this `SootClass`.

`SootClass getSuperclass()/void setSuperclass(SootClass s)`
Returns or sets the superclass of this class.

`Chain getFields()`
Returns a backed `Chain` of `SootFields` which are present in this class.

`Chain getMethods()`
Returns a backed `Chain` of `SootMethods` which are members of this class.

`void addMethod(SootMethod m)`
Adds the given method to this class. Throws a runtime exception if this method already belongs to `SootClass`.

`SootField getFieldByName(String s)`
Returns the field given by name. Throws a runtime exception if there is no field by the given name, or if there are two fields with the same name [2]

`void write()`
Writes this class out to a classfile named `'className.class'`

Types of Classes

There are four different types of classes: application, library, context and phantom. Each type of class has a different role to play with respect to optimizations and transformations. These role of each class is set upon SOOT's start-up.

- Application classes can be fully inspected and modified.
- Library classes can be fully inspected, but not modified. These classes are not regenerated when optimizing the application, but are assumed to be present exactly as is when the application is run.
- Context classes represent classes for which the implementation is unknown. The signatures of all the fields, methods and of the class itself are known, but nothing can assumed about the actual implementation of each method. This restriction is enforced in SOOT; accessing the implementation of a method of a context class will throw an exception.
- Phantom classes are those which are known to exist (because they are referenced in the constant pool of classfiles), but SOOT was unable to load . This occurs with obfuscated code, for example. Phantom classes and regular classes can also contain phantom fields and phantom methods. These are created whenever a class which

[2] Two fields with the name can occur in a `SootClass`, as long as they have different types.

SOOT has loaded refers to fields or methods which do not exist in existing classes or in a phantom class.

5.3.3 SootField

Instances of the `SootField` class represent Java fields. They possess at least the following methods:

`SootField(java.lang.String name, Type type, int modifiers)`
Constructs a `SootField` with the given name, type and modifiers. Types are described in subsection 5.3.10.

`String getName()/void setName(String)`
Gets or sets the name for this `SootField`.

`Type getType()/void setType(Type t)`
Gets or sets the type for this `SootField`.

`int getModifiers()/void setModifiers(int m)`
Gets or sets the modifiers for this `SootField`.

5.3.4 SootMethod

Instances of the `SootMethod` class represent Java methods. There is only one code representation at given time for methods and this is represented by the active `Body` for the `SootMethod`. `Body`s are described in subsection 5.3.6. `SootMethod`s possess the following methods:

`SootMethod(java.lang.String name, java.util.List paramTypes, Type returnType, int modifiers)`
Constructs a `SootMethod` with the given name, type and modifiers.

`String getName()/void setName(String)`
Gets/sets the name for this `SootMethod`.

`Type getReturnType()/void setReturnType(Type t)`
Gets/sets the return type for this `SootMethod`.

`List getParameterTypes()`
Returns a backed list of parameters types for this `SootMethod`.

`int getModifiers()/void setModifiers(int m)`
: Gets/sets the modifiers for this `SootMethod`.

`boolean isConcrete()`
: Returns true if this method can have a body.

`Body getActiveBody()/void setActiveBody(Body b)`
: Returns the current active body for this method, throws an exception if there is none/sets the active body to the given body.

`Body retrieveActiveBody()`
: Returns the current active body for this method, or the default `JimpleBody` if there is none.

5.3.5 Intermediate representations

The three intermediate representations BAF, JIMPLE, GRIMP are referred by the following three singletons: `Baf.v()`, `Jimple.v()` and `Grimp.v()`. These intermediate representation singletons are used to create objects belonging to the intermediate representation, such as `Locals`, `Traps`, or expressions. For example, `Jimple.v().newLocal("a", IntType.v())` creates a `Jimple` local variable of type integer.

5.3.6 Body

The implementation of the method is represented by an implementation of the `Body` interface. There are multiple implementations for `Body`, one for each intermediate representation: `JimpleBody`, `BafBody` and `GrimpBody`. There is also a `StmtBody` interface which is implemented by `JimpleBody` and `GrimpBody` which allows methods to target both JIMPLE code and GRIMP code simultaneously.

`Chain getLocals()`
: Returns a backed chain of the `Locals` in this method.

`Chain getTraps()`
: Returns a backed chain of the `Traps` in this method.

`PatchingChain getUnits()`
: Returns a backed patching chain of the `Units` in this method. `PatchingChains` are described in subsection 5.3.15.

5.3.7 Local

Instances of the `Local` class represent local variables.

`String getName()/ void setName(String s)`
: Gets or sets the name of the local variable.

`Type getType()/ void setType(Type t)`
: Gets or sets the type of the local variable. The types available depend on the particular intermediate representation used.

Locals are created with the method `newLocal(String name, Type t)` called on the intermediate representation singleton. For example, `Baf.v().newLocal("a", WordType.v())` creates a BAF local variable named "a" with type `word`.

5.3.8 Trap

Instances of the `Trap` class represent exception handler traps. At the Java level, exceptions are represented by try-catch blocks. At the bytecode level, exceptions are represented by explicit begin/end catch ranges. `Traps` represent these explicit ranges. See figure 5.2 for an illustration.

`Unit getBeginUnit(), void setBeginUnit(Unit u)`
: Gets or sets the beginning unit of the exception trap.

`Unit getEndUnit(), void setEndUnit(Unit u)`
: Gets or sets the ending unit of the exception trap.

`Unit getHandlerUnit(), void setHandlerUnit(Unit u)`
: Gets or sets the handling unit of the exception trap.

`SootClass getException(), void setException(SootClass c)`
: Gets or sets the exception class to trap.

Traps are created with the method `newTrap(a, b, c, d)` called on the intermediate representation singleton. For example, `Baf.v().newTrap(a, b, c, d)` creates a BAF trap which catches exceptions of type `d` thrown between `a` and `b` with handler `c`.

```
public void f()
{
  try {
    System.out.
        println("trying...");
  } catch(Exception e)
  {
    System.out.
        println("Exception!");
  }
}
```

bytecode

```
public void f()
{
    Test r0;
    java.io.PrintStream $r1, $r3;
    java.lang.Exception $r2;

    r0 := @this;

 label0:
    $r1 = java.lang.System.out;
    $r1.println("trying...");

 label1:
    goto label3;

 label2:
    $r2 := @caughtexception;
    $r3 = java.lang.System.out;
    $r3.println("Exception!");

 label3:
    return;

    catch java.lang.Exception from
      label0 to label1 with label2;
}
```

JIMPLE code

Figure 5.2: An example of an exception trap.

5.3.9 Unit

The `Unit` interface is the most fundamental interface in SOOT as it is used to represent instructions or statements. In BAF these are stack based instructions such as `PushInst` and `AddInst`, and in JIMPLE the legal `Units` are 3-address code statements such as `AssignStmt` and `InvokeStmt`.

These are created through methods such as `newXXX` on the intermediate representation singleton. For example, on the `Baf.v()` singleton the following methods can be called:

`AddInst newAddInst(Type opType)`
  Creates a BAF add instruction which deals with operands of type `opType`.

`DivInst newDivInst(Type opType)`
  Creates a BAF divide instruction which deals with operands of type `opType`.

And on the `Jimple.v()` singleton we can call methods such as:

`IdentityStmt newIdentityStmt(Value local, Value identityRef)`

  Creates an identity statement of the form `local := identityRef`.

`AssignStmt newAssignStmt(Value lvalue, Value rvalue)`
  Creates an assignment statement of the form `lvalue = rvalue`.

`InvokeStmt newInvokeStmt(InvokeExpr e)`
  Creates an invoke statement for the given invoke expression.

`ReturnVoidStmt newReturnVoidStmt()`
  Creates a return void statement.

The `Unit` interface contains the following methods:

`boolean branches()`
  Returns true if this `Unit` has the possibility of branching to another `Unit`, such as is the case with `goto`s.

`Object clone()`
  Returns a clone of this `Unit.`

`boolean fallsThrough()`
  Returns true if this `Unit` has the possibility of falling through to the next `Unit` in the `Chain`, such as is the case with an if-statement or a `nop` statement, as opposed to a `goto` statement which does not fall through.

`List getBoxesPointingToThis()`
: Returns a list of `Unit Boxes` containing pointers to this `Unit`. `Boxes` are defined in subsection 5.3.14.

`List getDefBoxes()`
: Returns a list of `Value Boxes` which contain definitions of values in this `Unit`.

`List getUnitBoxes()`
: Returns the list of `Unit Boxes` which contains `Units` referenced in this `Unit`.

`List getUseAndDefBoxes()`
: Returns a list of `Value Boxes` which contains both definitions and uses of values in this `Unit`.

`List getUseBoxes()`
: Returns a list of `Value Boxes` which contains value uses in this `Unit`.

`List redirectJumpsToThisTo(Unit newLocation)`
: Redirects all jumps to this `Unit` to the given `Unit`. This functionality is possible because backpointers to this `Unit` are kept.

Being able to invoke these methods on `Units` without knowing the particular type of `Unit` is one of the features of SOOT which allows us to write compact analysis and optimization code which is also general.

5.3.10 Type

Types in SOOT are represented with the value factory pattern. Different contexts allow different types to be used. In JIMPLE, we have the following types:

- *Base types:*

  `BooleanType.v(),ByteType.v(),CharType.v(),DoubleType.v(),`
  `FloatType.v(),IntType.v(),LongType.v(),ShortType.v(),`
  `VoidType.v()`

- `ArrayType.v(a,b)` where `a` is the base type of the array and `b` is the number of dimensions.[3]

[3] Unfortunately, when the types were designed `ArrayType` was not made a sub class of `RefType` which it turns out to have been a mistake. However, since there are many users of SOOT, we have chosen not to change the API at this time.

- `RefType.v(a)` where `a` is the name of the class for which this is a reference to. Note this is distinct from a reference to a `SootClass` object.

Parameters of methods and fields can have any of the above types. A local variable in JIMPLE may only be given basic Java Virtual Machine types; booleans, shorts, bytes, and chars are not allowed. But there is an additional type called null (`NullType.v()`). In BAF, local variables can only have one of two types: `WordType.v()` or `DoubleWordType.v()`.

5.3.11 Modifier

Modifiers are represented by final static integer constants:

- Modifier.ABSTRACT
- Modifier.FINAL
- Modifier.INTERFACE
- Modifier.NATIVE
- Modifier.PRIVATE
- Modifier.PROTECTED
- Modifier.PUBLIC
- Modifier.STATIC
- Modifier.SYNCHRONIZED
- Modifier.TRANSIENT
- Modifier.VOLATILE

Modifiers can be merged together by ”or”ing their values.

5.3.12 Value

The `Value` is an interface which is implemented by objects which represent productions or leaves in the grammar of the intermediate representation, and which are not `Units`. In JIMPLE, some examples of `Values` are `Constants` and subclasses of `Expr` such as `AddExpr` which represent the addition of two different `Values`. Instances of the `Value` can be created through the singleton `Jimple.v()`:

`newParameterRef(Type type, int n)`
: Constructs a parameter reference value of the given type with the given argument number, where the arguments are numbered starting at 0.

`newStaticFieldRef(SootField f)`
: Constructs a static field reference value to the given field.

`newVirtualInvokeExpr(Local base, SootMethod method, List args)`
: Constructs a virtual invoke expression on the given receiver to the given method with the given arguments.

`newVirtualInvokeExpr(Local base, SootMethod m, Value arg)`
: Constructs a virtual invoke expression on the given receiver to the given method with a single parameter.

`newAddExpr(Value leftOp, Value rightOp)`
: Constructs an add expression value for the given values.

The following methods are provided for all `Values`:

`Object clone()`
: Returns a clone of this `Value`.

`Type getType()`
: Returns the SOOT type for this `Value`.

`List getUseBoxes()`
: Returns a list of `ValueBoxes` containing the values used in this `Value`.

Note that there is no method named `getDefBoxes()` for `Value`. This is because `Values` never have any definition side effects, unlike `Units`.

5.3.13 Constants

Constants are represented with the value factory pattern. These implement the `Value` interface.

`IntConstant.v(a)`

`FloatConstant.v(a)`

`DoubleConstant.v(a)`

`LongConstant.v(a)`

`NullConstant.v()`

`StringConstant.v(a).`

5.3.14 Box

One of the fundamental concepts in SOOT is the notion of the `Box`. There are two types of `Boxes`: `UnitBox` and `ValueBox`. These contain `Units` and `Values` respectively. Whenever a `Unit`, `Value`, or any other object contains a reference to a `Value` or a `Unit` it is done so indirectly through a `Box` of the appropriate type. Figure 5.3 demonstrates this explicitly.

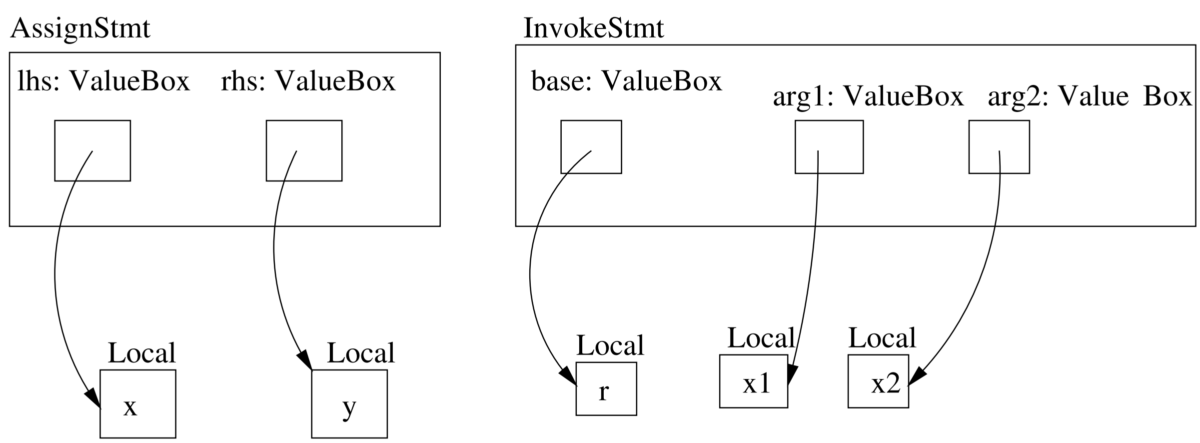

Figure 5.3: A layer of boxes is used between the containers and the `Values`. Boxes provide a de-referencing mechanism.

Boxes serve two fundamental roles.

The first role is to provide a de-referencing mechanism for references to `Values` and `Units`. A `Box` is essentially a pointer. We have noticed during the development of SOOT that it is convenient to be able to inspect and change references to other `Values` and `Units` without knowing where they occur in a `Body`, or in a `Unit`. Perhaps the simplest example occurs when performing constant propagation in JIMPLE. In constant propagation, if there is a use of a local variable for which only one definition exists and that definition is a constant, then we can replace the use of the local with a reference to the constant. Clearly, we do not care where the use occurs: it can occur in a `InvokeExpr` or a `AssignStmt` and in both cases the constant propagation should take place. To facilitate these construct independent transformations or inspections we added methods to the `Body`, `Value`, and `Unit` classes such as `getUseBoxes()` which returns a list of `ValueBoxes` which contain `Values` which are considered to be used as opposed to defined. Similarly, `Body` and `Unit` have a method named `getUnitBoxes()` which returns a list of `Boxes` which contains references to other `Units`. These are extremely useful when performing code migration from one method to another, for example, in order to preserve the consistency of the branches amongst the `Units`.

The second role is to enforce the grammar on the actual object representations. For example, in JIMPLE the arguments of an `InvokeExpr` can only be `Locals` or `Constants`. Changing the arguments is done through a method on the `InvokeExpr` which in turn attempts to change the contents of the `ValueBoxes` which represent the arguments. Since the `ValueBoxes` in this case are `ImmediateBoxes`, they will only accept `Constants` or `Locals` (and throw an exception if an object of a different type is inserted.) Note that by changing the `ValueBox` to something more flexible which accepts `Exprs` as well is how we implement the `InvokeExpr` for `Grimp`.

5.3.15 Patching Chains

The `Units` in a `Body` are contained in a `PatchingChain`, a subclass of `Chain`. The `PatchingChain` has some special properties which are desired when implementing compiler transformations. In particular, a common operation that one performs on a collection of `Units` is to delete a particular `Unit`. But what happens to all the `Units` which are referring to this `Unit`? They must somehow be patched in order to refer to the element which comes after the `Unit` removed. Similarly, one often wishes to insert a `Unit` *a* before another `Unit` *b*, and that the newly inserted `Unit` is in the same basic block. This is achieved by changing all references of *b* to *a*. The `PatchingChain` takes care of all these details automatically.

5.3.16 Packages and Toolkits

To structure SOOT, the API has been split into several packages. There is at least one package per intermediate representation, and code containing transformations or optimizations is usually stored in a toolkit package. Here is a list of the current packages available:

`soot`
: Base SOOT classes, shared by different intermediate representations.

`soot.baf`
: Public classes for the BAF intermediate representation.

`soot.baf.internal`
: Internal implementation-specific classes for the BAF intermediate representation.

`soot.baf.toolkits.base`
: A toolkit to optimize the BAF IR.

`soot.coffi`
: Contains classes from the Coffi tool, by Clark Verbrugge.

`soot.grimp`
: Public classes for the GRIMP intermediate representation.

`soot.grimp.internal`
: Internal implementation-specific classes for the GRIMP intermediate representation.

`soot.grimp.toolkits.base`
: A toolkit to optimize the GRIMP IR.

`soot.jimple`
: Public classes for the JIMPLE intermediate representation.

`soot.jimple.internal`
: Internal implementation-specific classes for the JIMPLE intermediate representation.

`soot.jimple.parser`
: An interface to the JIMPLE parser.

`soot.jimple.toolkits.base`
: A toolkit to optimize the JIMPLE IR.

`soot.jimple.toolkits.invoke`
: A toolkit to deal with JIMPLE and invoke statements.

`soot.jimple.toolkits.scalar`
A toolkit for scalar optimization of JIMPLE .

`soot.jimple.toolkits.typing`
Implements a typing algorithm for JIMPLE .

`soot.toolkits.graph`
Toolkit to produce and manipulate various types of control flow graphs.

`soot.toolkits.scalar`
A number of scalar optimizations which are not intermediate representation specific and the flow analysis framework.

`soot.util`
Generally useful utility classes for SOOT.

The complete API for these can be found online on the SOOT website[22].

5.3.17 Analyses and Transformations

As much as possible analyses and transformations are represented by singleton classes. This occurs more frequently for transformations than for analyses. In particular, transformations for the `Body` must extend the `BodyTransformer` class. And transformations of the entire `Scene` (all the classes, fields and methods) extend the `SceneTransformer` class. Representing transformations in this way allows them to be referred to in a way which is useful for enabling and disabling specific optimizations.

5.3.18 Graph representation of Body

SOOT provides a basic infrastructure for inspecting and modifying `Chains` of `Units` in the form of control flow graphs. There are two types of graphs, the `UnitGraph` and the `BlockGraph`. The latter represents control flow graphs as they are usually represented in traditional compiler textbooks[2]. The `BlockGraph` contains nodes called `Blocks` which are basic blocks. The `UnitGraph`, however, is a control flow graph in which individual nodes are `Units`. This has advantages and disadvantages. One advantage is that it simplifies the control flow graphs; there is no notion of a basic block. This simplifies implementing traditional and nontraditional fixed point data flow analyses. The main disadvantage, however, is that without basic blocks these data flow analyses take longer to run because basic blocks normally provide short cuts by summing the effect of multiple instructions.

Both the `UnitGraph` and the `BlockGraph` implement the `DirectedGraph` interface. This allows the development of methods which can operate on arbitrary `DirectedGraphs`, be they `Unit` or `Block` based. For example, there are methods which return pseudo-topological orderings on `DirectedGraphs` which are extremely useful for iterating on the `Blocks` or `Units` in an efficient manner.

`UnitGraph` and `BlockGraph` are in fact abstract classes. The concrete subclasses that may be instantiated are `CompleteUnitGraph`, `BriefUnitGraph`, `CompleteBlockGraph` and `BriefBlockGraph`. The words complete and brief relate to how the exception edges are handled. In the brief forms of the graphs, exceptions are practically ignored, as there are no edges between any `Units` and exception handlers. The exception handler `Units` are made to be heads of the graphs in this form. `Complete` graphs on the other hand include these edges. For most analyses the complete form is required. For example, computing ud/du chain information with a brief graph will yield incorrect information, as the uses of local variables within exception handlers will be ignored.

Note that most of the implementations of these graphs provide only static, snapshot views of the `Units`. It is not possible to modify the graph directly, or make a modification in the `Chain` of `Units` and have that change be reflected in the graph. This means that modifications in the `Chain` requires a regeneration of the active graphs for them to be correct. The only exception to this is the `BlockGraph`. If the `Block` objects are modified directly, the modifications will trickle down to the `Chain` of `Units`. So it is possible in this case to modify the basic blocks of the graph and maintain a correct view.

5.4 Usage examples

5.4.1 Creating a hello world program

This first example on how to use SOOT consists of creating a hello world program from scratch. Executing the program will create a class called ”HelloWorld”, which in turn, when executed, will print ”Hello world!”.

The code is first given, and then each numbered step is explained. The basic idea is to create an empty class, create an empty body, and then create the JIMPLE code for that body.

```
public class Main
{
    public static void main(String[] args)
    {
        SootClass sClass;
        SootMethod method;

        // Create the class
(1)|       Scene.v().loadClassAndSupport("java.lang.Object");

        // Declare 'public class HelloWorld'
(2)|       sClass = new SootClass("HelloWorld", Modifier.PUBLIC);
```

```
        // 'extends Object'
(3)|       sClass.setSuperclass(Scene.v().getSootClass("java.lang.Object"));
   |       Scene.v().addClass(sClass);

        // Create the method, public static void main(String[])
   |        method = new SootMethod("main",
(4)|             Arrays.asList(new Type[] {ArrayType.v(RefType.v("java.lang.String"), 1)}),
   |             VoidType.v(), Modifier.PUBLIC | Modifier.STATIC);
   |
   |       sClass.addMethod(method);

        // Create the method body
        {
            // create empty body
(5)|        JimpleBody body = Jimple.v().newBody(method);

(6)|        method.setActiveBody(body);
            Chain units = body.getUnits();
            Local arg, tmpRef;

            // Add some locals, java.lang.String[] l0
(7)|            arg = Jimple.v().newLocal("l0", ArrayType.v(RefType.v("java.lang.String"), 1));
                body.getLocals().add(arg);

(8)|        // Add locals, java.io.printStream tmpRef
                tmpRef = Jimple.v().newLocal("tmpRef", RefType.v("java.io.PrintStream"));
                body.getLocals().add(tmpRef);

(9)|        // add "l0 = @parameter0"
                units.add(Jimple.v().newIdentityStmt(arg,
                     Jimple.v().newParameterRef(ArrayType.v(RefType.v("java.lang.String"), 1),
                        0)));

(10)        // add "tmpRef = java.lang.System.out"
                units.add(Jimple.v().newAssignStmt(tmpRef, Jimple.v().newStaticFieldRef(
                    Scene.v().getField("<java.lang.System: java.io.PrintStream out>"))));

            // insert "tmpRef.println("Hello world!")"
            {
(11)|           SootMethod toCall = Scene.v().getMethod(
    |               "<java.io.PrintStream: void println(java.lang.String)>");
(12)|           units.add(Jimple.v().newInvokeStmt(Jimple.v().newVirtualInvokeExpr(tmpRef,
    |                toCall, StringConstant.v("Hello world!"))));
            }

            // insert "return"
(13)|           units.add(Jimple.v().newReturnVoidStmt());

        }

(14)|   sClass.write();
    }

}
```

Walkthrough of the code

1. The first step is load the class `java.lang.Object` into the `Scene`. This step loads all the fields and methods of the class, as well as all the objects needed to load this class, transitively. This is done in order to reference the `java.lang.Object` class later on.

2. Creates a new public class with the name "HelloWorld".

3. Makes the class `HelloWorld` extend `java.lang.Object`, and adds the class to the `Scene`. All classes should be added to the `Scene` once created. Note how we retrieve the class `java.lang.Object` with a call to `Scene.v().getSootClass ("java.lang.Object")`.

4. Create a method `void main(String[] args)` for this class. Note how the `ArrayType` is composed of a `RefType` and a dimension count, and that the modifiers are combined together by ORing them together.

5. Creates a new empty `JimpleBody`. There are no locals, no units and no traps at the moment.

6. Sets the body that was created to be the active body of this method. There can only be one active body.

7. Adds a local of type `java.lang.String[]` named `l0`. Note how the local created must be explicitly added to the `Chain` of locals returned by `body.getLocals()`.

8. Adds a temporary local `java.io.PrintStream tmpRef`.

9. Creates and adds an identity statement of the form

   ```
   l0 := @parameter0: java.lang.String[]
   ```

   This indicates that `l0` is the local variable which corresponds to the first argument passed to this method.

10. Creates an assignment statement of `java.lang.System.out` to the local variable `tmpRef`. Note how the field is accessed through its absolute signature in the `Scene.` The signature format is straightforward: the class name containing the field, followed by the type of the field and then the name of the field. This assignment statement is then added to the chain of units.

11. Gets the method class corresponding to the given method signature. Note how it is accessed through its absolute signature in the `Scene.`

12. Creates a call to the above method, with as argument the string constant ”Hello World!”.

13. Creates and adds a return void statement to the `Chain` of `Units`.

14. Writes the classfile to a file called ”HelloWorld.class”.

5.4.2 Implementing live variables analysis

This second usage example consists of implementing the standard *live variables analysis* using the SOOT framework.

To implement this analysis, we extend the `BackwardFlowAnalysis` class which is provided with SOOT. Then it suffices to provide a constructor for the analysis and to override four methods: `newInitialFlow(), flowThrough(), merge()` and `copy()`. There is also a fifth method which can generally be overriden: `customizeInitialFlowGraph()`. Although we do not use this method here, this method can be used to customize the initial flow graph so that, for example, some nodes start with different initial flow nodes.

We first provide the code, then we provide a walkthrough describing each step in detail.

```
class SimpleLiveLocalsAnalysis extends BackwardFlowAnalysis
{
    FlowSet emptySet;
    Map unitToGenerateSet;
    Map unitToPreserveSet;

    SimpleLiveLocalsAnalysis(UnitGraph g)
    {
        super(g);

        // Generate list of locals and empty set
        {
            Chain locals = g.getBody().getLocals();
(1)|        FlowUniverse localUniverse = new FlowUniverse(locals.toArray());

(2)|        emptySet = new ArrayPackedSet(localUniverse);
        }

   |     // Create preserve sets.
   |     {
   |        unitToPreserveSet = new HashMap(g.size() * 2 + 1, 0.7f);
   |
   |        Iterator unitIt = g.iterator();
   |
   |        while(unitIt.hasNext())
   |        {
   |            Unit s = (Unit) unitIt.next();
   |
(3)|  (a)|      BoundedFlowSet killSet = (BoundedFlowSet) emptySet.clone();
   |     |
   |     |      Iterator boxIt = s.getDefBoxes().iterator();
   |     |
   |     |      while(boxIt.hasNext())
   |     |      {
   |  (b)|          ValueBox box = (ValueBox) boxIt.next();
   |     |
   |     |          if(box.getValue() instanceof Local)
   |     |              killSet.add(box.getValue(), killSet);
   |     |      }
   |
   |     |      // Store complement
   |  (c)|         killSet.complement(killSet);
   |     |         unitToPreserveSet.put(s, killSet);
   |        }
   |    }
```

```
          // Create generate sets
          {
              unitToGenerateSet = new HashMap(g.size() * 2 + 1, 0.7f);

              Iterator unitIt = g.iterator();

              while(unitIt.hasNext())
              {
                 Unit s = (Unit) unitIt.next();

      (a)        FlowSet genSet = (FlowSet) emptySet.clone();

(4)              Iterator boxIt = s.getUseBoxes().iterator();

                 while(boxIt.hasNext())
                 {
      (b)            ValueBox box = (ValueBox) boxIt.next();

                     if(box.getValue() instanceof Local)
                         genSet.add(box.getValue(), genSet);
                 }

      (c)        unitToGenerateSet.put(s, genSet);
              }
         }

(5)      doAnalysis();
     }

     protected Object newInitialFlow()
     {
(6)      return emptySet.clone();
     }

     protected void flowThrough(Object inValue, Directed unit, Object outValue)
     {
         FlowSet in = (FlowSet) inValue, out = (FlowSet) outValue;

         // Perform kill
(7)           in.intersection((FlowSet) unitToPreserveSet.get(unit), out);

         // Perform generation
             out.union((FlowSet) unitToGenerateSet.get(unit), out);
     }

     protected void merge(Object in1, Object in2, Object out)
     {
         FlowSet inSet1 = (FlowSet) in1,
              inSet2 = (FlowSet) in2;
(8)
         FlowSet outSet = (FlowSet) out;

         inSet1.union(inSet2, outSet);
     }

     protected void copy(Object source, Object dest)
     {
         FlowSet sourceSet = (FlowSet) source,
(9)          destSet = (FlowSet) dest;

         sourceSet.copy(destSet);
     }
}
```

New classes

This subsection describes the classes which are specific to this example and are not described elsewhere in this chapter.

`FlowSet`
: The data which flows around the flow analysis framework should (although is not required to) implement the `FlowSet` interface. The `FlowSet` interface implements methods such as `union()`, `intersection()`, `isEmpty()` and so forth. There are two standard implementations of `FlowSet`: `ArrayPackedSet` and `ArraySparseSet.` The latter implements the set using a list, whereas the former implements the set using a bit-data vector.

`BoundedFlowSet`
: A `BoundedFlowSet` is a special `FlowSet` which is bounded, in the sense that the flow set's domain is restricted to a `FlowUniverse` set which is given upon the creation of the set. `BoundedFlowSets` are useful because the complement method can be used.

`FlowUniverse`
: Objects of this class are arrays of elements which are given to `BoundedFlowSets` upon their creation. They are used to specify the domain of the `BoundedFlowSets`.

Walkthrough of the code

1. The first step is to define the `FlowUniverse` to be the set of local variables in this `Body`.

2. Create an empty set for the given `FlowUniverse` using the `ArrayPackedSet` representation which is essentially a regular bit-vector representation. Note that `ArrayPackedSet` is a `BoundedFlowSet`. `ArraySparseSet` is the other representation for the bit-vector data, but it is unsuitable because it is unbounded.

3. Note that normally for the live variable flow analysis, we compute in's and out's as follows:

   $$out(s) = (in(s) \backslash kill(s)) \cup gen(s)$$

   We use the following equation instead:

   $$out(s) = (in(s) \cap preserve(s)) \cup gen(s)$$

   which simply uses a *preserve set* instead of a *kill* set. This block of code computes the preserve set for each statement in advance.

(a) Generate a new empty set.

(b) Iterate over the list of definitions for this unit. For each definition, add that local to the kill set. This is done by retrieving the `defBoxes` and then inspecting each `defBox` for a local.

(c) Store the complement of the kill set as the preserve set for this unit.

4. The next step is to create the generate set for each statement.

   (a) Create an empty set.

   (b) Iterate over the use boxes in this statement. If the use box contains a local, then add the local to the generate set.

   (c) Store the resulting set as the generate set for this statement.

5. Do the flow analysis which iterates until a fixed point is achieved. This calls the four following methods.

6. This method returns a brand new empty set. This is used to decorate each program point with an initial flow set. The method `customizeInitialFlowGraph` can be used if different initial flow sets must be put on each program point.

7. Perform the effect of flowing an `inValue` through `Unit`. Store the result in `out-Value`. In our case this means performing an intersection of the `in` set with the preserve set and then a union with the generate set.

8. Merge the flowsets `in1` with `in2` and store the result in `out`. In our case, for live variable analysis, we use union.

9. Implement a simple copy from `source` to `dest`.

5.4.3 Implementing constant propagation

This example implements a simple constant propagator for SOOT using the the results of the `SimpleLocalDefs` flow analysis provided with SOOT. Note that this example is self-contained and can be invoked to perform the transformation through its `main` method.

```
public class Main
{
     public static void main(String[] args)
     {
        if(args.length == 0)
        {
            System.out.println("Syntax: java Main <classfile> [soot options]");
            System.exit(0);
        }
```

```
(1)     Scene.v().getPack("jtp").add(new Transform("jtp.propagator", Propagator.v()));
        soot.Main.main(args);
    }
}

class Propagator extends BodyTransformer
{
    private static Propagator instance = new Propagator();
    private Propagator() {}
(2)
    public static Propagator v() { return instance; }

    static String oldPath;
(3) protected void internalTransform(Body b, String phaseName, Map options)
    {
        if (soot.Main.isVerbose)
            System.out.println("[" + b.getMethod().getName() + "] Propagating constants...");

        JimpleBody body = (JimpleBody) b;

        Chain units = body.getUnits();
(4)     CompleteUnitGraph stmtGraph = new CompleteUnitGraph(body);

        LocalDefs localDefs = new SimpleLocalDefs(stmtGraph);
        Iterator stmtIt = units.iterator();

        while(stmtIt.hasNext())
        {
            Stmt stmt = (Stmt) stmtIt.next();
(5)         Iterator useBoxIt = stmt.getUseBoxes().iterator();

            while(useBoxIt.hasNext())
            {
                ValueBox useBox = (ValueBox) useBoxIt.next();

                if(useBox.getValue() instanceof Local)
                {
                    Local l = (Local) useBox.getValue();
                    List defsOfUse = localDefs.getDefsOfAt(l, stmt);
(6)
                    if(defsOfUse.size() == 1)
                    {
                        DefinitionStmt def = (DefinitionStmt)
                            defsOfUse.get(0);

(7)                     if(def.getRightOp() instanceof Constant)
                        {
(8)                         if(useBox.canContainValue(def.getRightOp()))
(9)                             useBox.setValue(def.getRightOp());
                        }
                    }
                }
            }
        }
    }
}
```

Walkthrough of the code

1. This hooks a call to the propagator into SOOT's list of transformations. The call `Scene.v().getPack("jtp")` returns the `Pack` which corresponds to the JIMPLE transformation pack (which is a list of all the transformations performed on JIMPLE). Other packs are `jop` and wjtp which stand for JIMPLE optimization pack and the whole JIMPLE transformation pack. Then we add a `Transform` instance to this

pack which is essentially a name for the transformation and the transformer class implementing the `BodyTransformer` interface.

2. This code implements the singleton functionality for the class.

3. This is the entry point method which contains the body of the transformation.

4. Retrieve the list of units for this body, build the `CompleteUnitGraph` and the `SimpleLocalDefs` object which contains use-def and def-use information.

5. Iterate over all the uses in the program, looking for uses of local variables.

6. Get the definition for that local variable. If only one definition exists, then do step 7.

7. If the right hand side of the definition is a constant.

8. If the use box can contain a constant.

9. Set the contents of the use box to the right hand side of the definition, that is, the constant.

5.4.4 Instrumenting a classfile

This example instruments the given application so that when it is executed, it outputs the number of gotos that were dynamically executed. The general idea is to:

1. Insert a counter in the main class called `gotoCount`.

2. Insert counter incrementors before each goto statement present in the JIMPLE code.

3. Insert a print of the counter at the end of the `mainClass`, or before any call to `System.exit()`.

```
public class Main
{
    public static void main(String[] args)
    {
        if(args.length == 0)
        {
            System.out.println("Syntax: java Main --app <main_classfile> [soot "
                + "options]");
            System.exit(0);
        }

(1) |   Scene.v().getPack("jtp").add(new Transform("jtp.instrumenter",
    |      GotoInstrumenter.v()));
        soot.Main.main(args);
    }
}

class GotoInstrumenter extends BodyTransformer
```

```
{
    private static GotoInstrumenter instance = new GotoInstrumenter();
(2) private GotoInstrumenter() {}
    public static GotoInstrumenter v() { return instance; }

    public String getDeclaredOptions() { return super.getDeclaredOptions(); }

    private boolean addedFieldToMainClassAndLoadedPrintStream = false;
    private SootClass javaIoPrintStream;

    private Local addTmpRef(Body body)
    {
(3)     Local tmpRef = Jimple.v().newLocal("tmpRef", RefType.v("java.io.PrintStream"));
        body.getLocals().add(tmpRef);
        return tmpRef;
    }

    private Local addTmpLong(Body body)
    {
(4)     Local tmpLong = Jimple.v().newLocal("tmpLong", LongType.v());
        body.getLocals().add(tmpLong);
        return tmpLong;
    }

    private void addStmtsToBefore(Chain units, Stmt s, SootField gotoCounter, Local tmpRef,
        Local tmpLong)
    {
        // insert "tmpRef = java.lang.System.out;"
        units.insertBefore(Jimple.v().newAssignStmt(
                    tmpRef, Jimple.v().newStaticFieldRef(
                    Scene.v().getField("<java.lang.System: java.io.PrintStream out>"))), s);

        // insert "tmpLong = gotoCounter;"
(5)     units.insertBefore(Jimple.v().newAssignStmt(tmpLong,
                    Jimple.v().newStaticFieldRef(gotoCounter)), s);

        // insert "tmpRef.println(tmpLong);"
        SootMethod toCall = javaIoPrintStream.getMethod("void println(long)");
        units.insertBefore(Jimple.v().newInvokeStmt(
                       Jimple.v().newVirtualInvokeExpr(tmpRef, toCall, tmpLong)), s);
    }

(6) protected void internalTransform(Body body, String phaseName, Map options)
    {
        SootClass sClass = body.getMethod().getDeclaringClass();
        SootField gotoCounter = null;
        boolean addedLocals = false;
        Local tmpRef = null, tmpLong = null;
        Chain units = body.getUnits();

        if (!Scene.v().getMainClass().
            declaresMethod("void main(java.lang.String[])"))
                throw new RuntimeException("couldn't find main() in mainClass");

(7)     if (addedFieldToMainClassAndLoadedPrintStream)
            gotoCounter = Scene.v().getMainClass().getFieldByName("gotoCount");
        else
        {
            // Add gotoCounter field
            gotoCounter = new SootField("gotoCount", LongType.v(),
                                        Modifier.STATIC);
            Scene.v().getMainClass().addField(gotoCounter);
(8)
            javaIoPrintStream = Scene.v().getSootClass("java.io.PrintStream");
            addedFieldToMainClassAndLoadedPrintStream = true;
        }

        // Add code to increase goto counter each time a goto is encountered
        {
            boolean isMainMethod = body.getMethod().getSubSignature().equals("void " +
```

```
                "main(java.lang.String[])");

            Local tmpLocal = Jimple.v().newLocal("tmp", LongType.v());
            body.getLocals().add(tmpLocal);

            Iterator stmtIt = units.snapshotIterator();

             while(stmtIt.hasNext())
(9)          {
                 Stmt s = (Stmt) stmtIt.next();

                if(s instanceof GotoStmt)
                {
                    AssignStmt toAdd1 = Jimple.v().newAssignStmt(tmpLocal,
                       Jimple.v().newStaticFieldRef(gotoCounter));
                    AssignStmt toAdd2 = Jimple.v().newAssignStmt(tmpLocal,
                        Jimple.v().newAddExpr(tmpLocal, LongConstant.v(1L)));
                    AssignStmt toAdd3 = Jimple.v().newAssignStmt(Jimple.v().
                        newStaticFieldRef(gotoCounter), tmpLocal);
(9a)
                    // insert "tmpLocal = gotoCounter;"
                    units.insertBefore(toAdd1, s);

                    // insert "tmpLocal = tmpLocal + 1L;"
                    units.insertBefore(toAdd2, s);

                    // insert "gotoCounter = tmpLocal;"
                    units.insertBefore(toAdd3, s);
                }
                else if (s instanceof InvokeStmt)
                 {
                    InvokeExpr iexpr = (InvokeExpr) ((InvokeStmt)s).getInvokeExpr();
                    if (iexpr instanceof StaticInvokeExpr)
                    {
                        SootMethod target = ((StaticInvokeExpr)iexpr).getMethod();

                        if (target.getSignature().equals("<java.lang.System: void " +
                            "exit(int)>"))
(9b)                    {
                             if (!addedLocals)
                             {
                                 tmpRef = addTmpRef(body); tmpLong = addTmpLong(body);
                                 addedLocals = true;
                             }
                             addStmtsToBefore(units, s, gotoCounter, tmpRef, tmpLong);
                         }
                     }
                }
                else if (isMainMethod && (s instanceof ReturnStmt || s instanceof
                    ReturnVoidStmt))
                {
                    if (!addedLocals)
                    {
(9c)                    tmpRef = addTmpRef(body); tmpLong = addTmpLong(body);
                        addedLocals = true;
                    }
                    addStmtsToBefore(units, s, gotoCounter, tmpRef, tmpLong);
                }
            }
        }
    }
}
```

Walkthrough of the code

1. This hooks a call to the propagator into SOOT's list of transformations. The call `Scene.v().getPack("jtp")` returns the `Pack` which corresponds to the JIMPLE transformation pack (which is a list of all the transformations performed on JIMPLE). Then we add a `Transform` instance to this pack which is essentially a name for the transformation and the transformer class implementing the `BodyTransformer` interface.

2. These fields and methods implement the singleton functionality.

3. Adds a temporary reference local variable of type `java.io.PrintStream` to the given body.

4. Adds a temporary long local variable of type `LongType` to the given body.

5. Inserts the following section of code before `s`:

   ```
   tmpRef = java.lang.System.out;
   tmpLong = gotoCount;
   tmpRef.println(tmpLong);
   ```

   This is used to just print out the contents of the goto counter.

6. This is the entry method for the transformation.

7. If the goto counter has been already added to the PrintStream class, then retrieve it.

8. Else, create a field called `gotoCount` of type long, and add it to the main class.

9. Iterate over all the statements in the class. Note that a snapshot iterator is used to allow modifications to the `Chain` of units while iterating.

   (a) If the statement is a goto statement, then insert the following code before the goto, to increment the counter:

   ```
   tmpLocal = gotoCounter;
   tmpLocal = tmpLocal + 1L;
   gotoCounter = tmpLocal;
   ```

   (b) If the statement is a static invoke to the method `System.exit(int)` then call `addStmtsToBefore` which inserts a print of the goto counter.

   (c) If the statement is the return void of the main method then call `addStmtsToBefore` which inserts a print of the goto counter.

5.4.5 Evaluating a Scene

This is a simple example which counts the number of classfiles, methods and fields in the active `Scene`. The idea is to iterate over all of these objects and increment counts whenever appropriate.

```
public class Main
{
    public static void main(String[] args)
    {
        if(args.length == 0)
        {
            System.out.println("Syntax: java Main --app <main_classfile> [soot options]");
            System.exit(0);
        }

(1)|    Scene.v().getPack("wjtp").add(new Transform("wjtp.profiler", Evaluator.v()));
        soot.Main.main(args);
    }
}

class Evaluator extends SceneTransformer
{
   |private static Evaluator instance = new Evaluator();
(2)|private Evaluator() {}
   |static String oldPath;

    public static Evaluator v() { return instance; }

(3)|protected void internalTransform(String phaseName, Map options)
    {
        long classCount = 0;
        long stmtCount = 0;
        long methodCount = 0;

        // Pre-process each class, constructing the invokeToNumberMap
        {
   |        Iterator classIt = Scene.v().getApplicationClasses().iterator();
   |
   |        while(classIt.hasNext())
(4)|        {
   |           SootClass sClass = (SootClass) classIt.next();
   |           classCount++;

   |            Iterator methodIt = sClass.getMethods().iterator();
   |
(5)|            while(methodIt.hasNext())
   |            {
   |                SootMethod m = (SootMethod) methodIt.next();
   |                methodCount++;

(6)|                if(!m.isConcrete())
   |                    continue;

(7)|                JimpleBody body = (JimpleBody) m.retrieveActiveBody();
   |                stmtCount += body.getUnits().size();
                }
            }
        }

        DecimalFormat format = new DecimalFormat("0.0");

        System.out.println("Classes: \t" + classCount);
        System.out.println("Methods: \t" + methodCount + " (" +
            format.format((double) methodCount / classCount) + " methods/class)" );
        System.out.println("Stmts:   \t" + stmtCount +  " (" +
            format.format((double) stmtCount / methodCount) + " units/methods)");
        System.exit(0);
```

```
    }
}
```

Walkthrough of the code

1. This hooks a call to the propagator into SOOT's list of transformations. The call `Scene.v().getPack("wjtp")` returns the `Pack` which corresponds to the JIMPLE transformation pack (which is a list of all the whole program transformations performed on JIMPLE). Then we add a `Transform` instance to this pack which is essentially a name for the transformation and the transformer class implementing the `BodyTransformer` interface.

2. Implements the singleton functionality for this interface.

3. This is the entry point method which contains the body of the analysis.

4. Iterate over the application classes in the `Scene`.

5. Iterate over the methods in each class.

6. If the method has no body, skip this method.

7. Retrieve the active body for this method, and increase the statement count by the size of the `Unit Chain` in the `Body`.

5.4.6 Summary

This chapter presented a complete overview of the basic SOOT Application Programming Interface (API). Five example programs built using the SOOT framework were presented and walkthroughs were given to describe them.

The current and complete API can be found online on our web site[22].

Chapter 6

Experiences

6.1 The Curse of Non-Determinism

While developing SOOT, we encountered the interesting phenomenon of nondeterministic optimization. In the first version of SOOT, the same program would be fed into SOOT twice, and two different optimized versions would be produced. This is undesirable for several reasons. First, it makes it very difficult to debug SOOT because each time you run it on a test program different results are produced. Second, because the optimized programs produced by SOOT are not always the same, it makes it difficult to reproduce results and to understand the effect of the optimizations. For example, one technique which is commonly used to isolate the effect of an optimization is to turn it off and note the slowdown. Having any of the optimizations behave nondeterministicly invalidates this common technique. Thus it is quite clear that nondeterminism must be avoided.

So where is the nondeterminism being introduced? Although we are not using random numbers explicitly, they are being used implicitly when using a `java.util.Hashtable` without overriding the `hashCode()` method. The default implementation for this method is to provide a hash code based on the actual memory location of the `Object`. Although this provides a great hash code, it is clear that on every execution of SOOT the hash tables will be different. This difference has a noticeable effect when one iterates through the hash table, because the elements are returned following the natural order of the table.

To avoid nondeterminism one must simply avoid iterating on a hash table, or provide a deterministic `hashCode()` method for the objects being inserted into the hash table. Sometimes the latter is impossible without producing a hash method with several collisions. In this case, if you absolutely need some sort of set which has constant time insertion, removal, and queries, and produces deterministic enumerations then your best bet is the `Chain` described in subsection 5.2.2. These techniques were used in the more recent versions of SOOT and the framework is deterministic.

6.2 Sentinel Test Suite

We have coined the term *sentinel test suite* to denote a suite of programs which are used to validate the correctness of our compiler framework. Due to the complexity of compiler work, it is essential to perform regular tests on this test suite to ensure that the compiler operates properly, and that we have not introduced bugs with the latest modification to SOOT. Currently the sentinel test suite contains 266 different programs of varying complexity. Each time a substantial bug is found in our framework, the program which produces the bug is isolated and shrunk to its smallest size and added to the sentinel test suite.

Chapter 7

Conclusions and Future Work

We presented SOOT, a framework which simplifies the task of optimizing Java bytecode. The contributions of this thesis are the design, implementation and experimental validation of this framework. The implementation of the framework consists of roughly 80,000 lines of code.

Java bytecode is a poor choice of intermediate representation for the implementation of optimizations because it is stack based. The stack implicitly participates in every computation, expressions are not explicit and they can be arbitrarily large. Even simple optimizations and transformations become difficult to design and implement in this form. SOOT rectifies this situation by providing three intermediate representations:

1. BAF, a streamlined representation of bytecode which is simple to manipulate. This is used to simplify the development of analyses and transformations which absolutely must be performed on stack code. Unlike bytecode, BAF does not require a JSR-equivalent instruction or a constant pool. Further, BAF has explicit local variables, exception ranges and typed instructions.

2. JIMPLE, a typed 3-address code intermediate representation suitable for optimization. It is our ideal form for optimization because it is compact, stackless, consists of 3-address code, and the local variables are typed and named.

3. GRIMP, an aggregated version of JIMPLE suitable for decompilation and for reading. It allows trees to be constructed as opposed to the flat expressions present in JIMPLE.

SOOT also provides a set of transformations between these intermediate representations: from bytecode to JIMPLE via one path and JIMPLE to bytecode via two different paths (one via BAF, and the other via GRIMP).

Extensive results of two experiments were given. The first experiment validated that Java bytecode can be converted to JIMPLE code and back to bytecode without a loss of performance. Two methods of producing bytecode from JIMPLE code were examined, with the BAF method being the most effective. The second experiment showed that the effect of applying optimizations on JIMPLE code can in fact yield a speed-up, and that the framework is a valid approach to optimizing bytecode. We are encouraged by our results so far, and we have found that the SOOT API has been effective for a variety of tasks including the devirtualization of methods using variable type analysis, decompilation, and the optimizations presented in this thesis, as up to 38% speed-up is achieved.

SOOT provides an API which we believe is useable, extendable and general for implementing analyses and optimizations. It is widely being used at McGill University for most of our ongoing compiler projects. It was released as publicly available code in March 2000 and it has also been adopted by several research groups at other institutions. We hope that it will be adopted by more research groups which will enable the widespread sharing of code and experimental results to further the state of research on optimizing Java bytecode.

We are actively engaged in further work on SOOT on many fronts. We have been exploring additional optimizations, such as loop invariant removal and common sub-expression elimination with side effect information. We have also begun researching the use of attributes with stack allocation and array bounds check elimination, as well as investigating the optimization of SOOT itself.

Bibliography

[1] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast and effective code generation in a just-in-time Java compiler. *ACM SIGPLAN Notices*, 33(5):280–290, May 1998.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178, Berkeley, USA, June 15–19 1998. USENIX Association.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[5] DashOPro.
. http://www.preemptive.com/products.html.

[6] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. VORTEX: An optimizing compiler for object-oriented languages. In *Proceedings OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31 of *ACM SIGPLAN Notices*, pages 83–100. ACM, October 1996.

[7] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, 7–11 August 1995. Springer.

[8] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an Optimizing Compiler for Java. Microsoft technical report, Microsoft Research, October 1998.

[9] Flex
. http://www.flex-compiler.lcs.mit.edu/.

[10] Etienne M. Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for java bytecode. In *Static Analysis Symposium 2000*, Lecture Notes in Computer Science, Santa Barbara, June 2000.

[11] Mark Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, volume Volume 2. Wiley, 1998.

[12] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(4):135–150, March 1993.

[13] JavaClass.
. http://www.inf.fu-berlin.de/ dahm/JavaClass/ .

[14] Compaq JTrek.
. http://www.digital.com/java/download/jtrek .

[15] Han Bok Lee and Benjamin G. Zorn. A Tool for Instrumenting Java Bytecodes. In *The USENIX Symposium on Internet Technologies and Systems*, pages 73–82, 1997.

[16] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[17] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[18] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Berkeley, June 16–20 1997. Usenix Association.

[19] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A way ahead of time (WAT) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 41–54, Berkeley, June 16–20 1997. Usenix Association.

[20] Ven Seshadri. IBM High Performance Compiler for Java. In *AIXPert Magazine*, sep 1997.

[21] Tatiana Shpeisman and Mustafa Tikir. Generating Efficient Stack Code for Java. Technical report, University of Maryland, 1999.

[22] Soot - a Java Optimization Framework.
. http://www.sable.mcgill.ca/soot/.

[23] 4thpass SourceGuard.
. http://www.4thpass.com/sourceguard/.

[24] Suif. http://www.suif.stanford.edu/.

[25] SuperCede, Inc. SuperCede for Java.
. http://www.supercede.com/.

[26] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical Experience with an Application Extractor for Java . IBM Research Report RC 21451, IBM Research, 1999.

[27] Tower Technology. Tower J.
. http://www.twr.com/.

[28] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In David A. Watt, editor, *Compiler Construction, 9th International Conference*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, Berlin, Germany, March 2000. Springer.