# Jimple: Simplifying Java Bytecode for Analyses and Transformations*

Raja Vallée-Rai    Laurie J. Hendren

Sable Research Group
McGill University
{kor,hendren}@sable.mcgill.ca

## Abstract

In this paper we present Jimple, a 3-address intermediate representation that has been designed to simplify analysis and transformation of Java bytecode. We motivate the need for a new intermediate representation by illustrating several difficulties with optimizing the stack-based Java bytecode directly. In general, these difficulties are due to the fact that bytecode instructions affect an expression stack, and thus have implicit uses and definitions of stack locations. We propose Jimple as an alternative representation, in which each statement refers explicitly to the variables it uses. We provide both the definition of Jimple and a complete procedure for translating from Java bytecode to Jimple. This definition and translation have been implemented using Java, and finally we show how this implementation forms the heart of the Sable research projects.

## 1 Introduction

The Java programming language is gaining a lot of popularity, both as a language for programming small applets on the web, and as a language for developing larger application systems. Although the design of the source language and target bytecode are quite clean, there remains significant challenges in providing Java compilers and run-time environments that can exhibit run-time performance comparable to what is currently possible for applications written in C/C++. Improving the performance of Java needs to be attacked at many levels, including: static bytecode optimization, Just-In-Time (JIT) compiler technology and improved run-time environments. In all cases, static program analysis can provide important information. This paper reports on the design and implementation of the Jimple intermediate representation that has been designed to simplify the task of developing static analyses and transformations.

When considering the correct intermediate representation for optimization, one might think that analyzing Java bytecode directly would be a good idea. Certainly the bytecode is not too low-level, since all important program information like types is directly available. Further, since bytecode is the "binary" of the Java world, one would avoid spurious translations to a separate intermediate code. However, Java bytecode is a stack-based intermediate representation where instructions have an implicit effect on an evaluation stack (for example, the bytecode instruction `iadd` pops two integer values and pushes their sum). Many existing analysis and transformation techniques rely on a more traditional intermediate representation such as 3-address instructions, where each instruction has explicit named operands (for example, an addition would be represented as something like `x=a+b`). Jimple provides such a 3-address representation.

The remainder of the paper is structured as follows. In Section 2 we explore the difficulties of analyzing and transforming a stack-based representation in more depth, and we motivate the need for a Jimple-like representation. Section 3 gives

*This research was supported, in part, by NSERC and IBM's Centre for Advanced Studies (CAS). Java is a trademark of Sun Microsystems Inc.

the definition of Jimple, and Section 4 shows how we translate Java bytecode to Jimple. In doing this translation we concentrate on producing correct Jimple code for any verifiable bytecode (even for bytecode that does not come directly from a Java compiler), and we try to mimimize the number of extra statements and variable names that are introduced. Both the Jimple API and the algorithms to translate Java bytecode to Jimple has been implemented in Java. In Section 5 we give a brief overview of how we are using Jimple with our Sable Java projects at McGill and, finally, in section 6 we give some conclusions and a brief description of future work.

## 2 Motivation

A central theme of this paper is that we prefer to develop analyses and optimizing transformations on Jimple, a 3-address intermediate representation, rather than optimizing and transforming the stack-based bytecode directly. In this section we discuss the benefits of stack-based code in general (Section 2.1), and then examine some disadvantages of analyzing and optimizing the stack-based code directly (Section 2.2).

### 2.1 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 correct 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.

### 2.2 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 assignments (such as `x=a+b`) and branch statements (such as an `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 instruction 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 these 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. 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. Another way of looking at this problem is that some bytecode expressions, like ineg, are implicitly defining a variable, but the name of the variable is not explicit, it is simply some stack location. By transforming to Jimple 3-address code all definitions become explicit definitions of named variables.

**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 also must 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 `pop`s. Note that a more sophisticated analyses would hoist the `imul, istore 5` sequence to one basic block higher to achieve a better result, but this shows exactly how simple transformations have become complicated. In Jimple this problem does not occur, because a dead assignment can be eliminated without any fear of affecting other areas of the program.

In general, transforming Jimple code is simpler because the fundamental units (instances of 3-address code) are self contained and do not depend on other nearby units for their meaning.

For developing analyses and transformations, it should be clear that working with 3-address code, like Jimple, is much simpler and more efficient than dealing with stack code. Thus the need for a correct and efficient way of transforming Java bytecode to and from Jimple in to make it profitable to optimize the code in Jimple.

The remainder of this paper gives a correct transformation from Jimple to Java bytecode. Future work will optimize the transformations, while retaining correctness.

## 3 Definition of Jimple

The Jimple intermediate representation comes in two flavors: *external* and *internal*. The external representation is a regular ASCII text file which represents a classfile in its Jimple form. This is important in order to store Jimple code and provides compiler tools with a new language to target. In general, however, manipulating text files to perform optimizations is cumbersome and so an internal representation to use in programs is needed. We supply such an internal representation in the form of a Jimple API (written in Java, of course) and this API enables us to represent and manipulate Jimple code in a straightforward and elegant manner.

Thus the Jimple intermediate representation can be defined concretely in two different ways. We present a grammar for the external version in figures 1 and 2, and the latest specification of the Jimple API can be found on the web at `http://www.sable.mcgill.ca/jimple`.

Both forms have the following essential characteristics:

- The code is stackless.
- Expressions are restricted to the least number of operands (2 in most cases, such as for arithmetic expressions, but possible more for some method calls), and these operands must either be constants or locals.
- All local variables must be explicitly declared and typed.
- Some local variables can be assigned initial special roles (such as the containers for the passed parameters, or a caught exception.)

See figure 10 for a simple example of a complete Jimple method.

Finally, we would like to note that Jimple's design was inspired by SIMPLE [HS92], an AST to represent C statements in a simple manner also to facilitate optimizations. Furthermore, our current design of Jimple is in its third generation, and notes on the previous incarnations and their problems can be found in [VR98].

## 4 Translating Bytecode to Jimple

Generating correct Jimple code from bytecode has two fundamental requirements. The generated code must be *stackless*, and all local variables must be *explicitly typed.* Since we are interested in optimizing this Jimple code, however, we also require that code be *compact.* To illustrate these three requirements, consider the transformation of the following bytecode.

```
  iload 0             int x, y, z;
  iload 1
  iadd        -->     z = x + y;
  istore 2

Original Java         Good Jimple code
  bytecode
```

In the produced Jimple code, the stack has disappeared, all local variables are typed and the statement is as compact as possible. Thus this is a good of example of the type of code we want to generate. To satisfy these requirements and generate good Jimple code, we propose a five step transformation.

### 4.1 Five Steps to Jimple

1. **Producing Verbose Typeless Jimple:** The first step to produce Jimple code is to produce verbose typeless Jimple. This is performed by mapping the operand stack onto a set of untyped Jimple local variables, and transforming the implicit references to the stack operands in the stack code to explicit references to the local variables in Jimple. An illustration:

```
.numlocals 2        unknown op0, op1;
.maxdepth 2         unknown a, b;

iload 0             op0 = a;
iload 0             op1 = a;
imul                op0 = op0 * op1;
istore 1            b = op0;

Java bytecode       Jimple code
```

   Note that before translating a given bytecode instruction to its Jimple counterpart, the height of the operand stack must be determined before that bytecode.. This is necessary in order to generate the correct references to the Jimple local variables representing the stack. (This is illustrated in the previous example by noting that the same Java bytecode instruction generates different Jimple bytecode based on its position.) Determining the height of the operand stack is achieved through a simple traversal of the code which models the stack effect of each instruction. This modeling process (as well as the translation process) is tedious to implement due to the sheer number different Java bytecode instructions to handle.

   The following table summarizes the effect on the stack and the Jimple code generated for a few Java bytecode instructions.

| bytecode | stack effect | code generated |
|---|---|---|
| iadd | nt=t-1 | op(nt)=op(t)+op(t-1) |
| iload x | nt=t+1 | op(t) = x |
| ladd | nt=t-2 | op(nt)=op(t)+op(t-2) |
| nop | nt=t | <nothing> |
| pop | nt=t-1 | <nothing> |

```
t is the top index of the stack
nt is the new top index
```

   There are two interesting points to note:

   - Generated typeless Jimple at this point is self-contained. We could stop after step one and have a valid intermediate representation on which to operate. We are interested, however, in typing the local variables since this is essential for higher level optimizations such as type based alias analysis[DMM98].
   - Producing correct typeless Jimple is possible because the Java Virtual Machine guarantees that the depth of the operand stack is fixed for every program point throughout the execution of the program[LY96].

2. **Compacting the Code:** The previous stage produces correct, but extremely verbose code. A simple Java assignment statement such as `x = a * a` is converted to four Jimple instructions, as illustrated by the example in step 1. The code can be compacted by simply performing copy/constant propagation and dead code elimination. More precisely, the algorithm we use is:

```
jimple_method = local_declarations identity_stmts
    stmts exception_ranges;
local_declarations = local_declaration local_declarations | ;
stmts = label ":" stmt ";" stmts |
    stmt ";" smts | ;
exception_ranges = exception_range exception_ranges | ;
local_declaration = type name ";";
identity_stmts = identity_stmt identity_stmts | ;

stmt = breakpoint_stmt | assign_stmt | enter_monitor_stmt |
    goto_stmt | if_stmt | invoke_stmt | lookup_switch_stmt | nop_stmt | ret_stmt |
    return_stmt | return_void_stmt | table_switch_stmt | throw_stmt;
breakpoint_stmt = "breakpoint";
assign_stmt = variable "=" rvalue;
identity_stmt = local ":=" identity_value;
enter_monitor_stmt = "entermonitor" immediate;
exit_monitor_stmt = "exitmonitor" immediate;
goto_stmt = "goto" label;
if_stmt = "if" condition "then" label;
invoke_stmt = invoke_expr;
lookup_switch_stmt = "lookupswitch(" immediate ") {"  cases " default: goto " label "}"
nop_stmt = "nop";
ret_stmt = "ret" local;
return_stmt = "return" immediate;
return_void_stmt = "return" ;
table_switch_stmt = "tableswitch(" immediate ") {" cases " default: goto " label "}"
throw_stmt = "throw" immediate;

cases = case cases | ;
case = "case " int_constant ": goto " label;

condition = eq_expr | ge_expr | le_expr | lt_expr | ne_expr | gt_expr;
rvalue = array_ref | constant | expr | instance_field_ref | local |
    next_next_stmt_address | static_field_ref;
identity_value = caught_exception_ref | parameter_ref | this_ref;
variable = array_ref | instance_field_ref | static_field_ref | local;
immediate = constant | local;

expr = binop_expr | cast_expr | instance_of_expr | invoke_expr | new_array_expr |
    new_expr | new_multi_array_expr | unop_expr;
binop_expr = add_expr | and_expr | cmp_expr | cmpg_expr | cmpl_expr | div_expr |
    eq_expr | ge_expr | gt_expr | le_expr | lt_expr | mul_expr | ne_expr | or_expr |
    rem_expr | shl_expr | shr_expr | sub_expr | ushr_expr | xor_expr;
add_expr = immediate "+" immediate;
and_expr = immediate "&" immediate;
cmp_expr = immediate "cmp" immediate;
cmpg_expr = immediate "cmpg" immediate;
cmpl_expr = immediate "cmpl" immediate;
div_expr = immediate "/" immediate;
eq_expr = immediate "==" immediate;
ge_expr = immediate ">=" immediate;
gt_expr = immediate ">" immediate;
le_expr = immediate "<=" immediate;
lt_expr = immediate "<" immediate;
mul_expr = immediate "*" immediate;
```

Figure 1: Grammar for external Jimple. (part 1 of 2)

```
ne_expr = immediate "!=" immediate;
or_expr = immediate "|" immediate;
rem_expr = immediate "%" immediate;
shl_expr = immediate "<<" immediate;
shr_expr = immediate ">>" immediate;
sub_expr = immediate "-" immediate;
ushr_expr = immediate "ushr" immediate;
xor_expr = immediate "xor" immediate;
cast_expr = "(" type ")" immediate;

instance_of_expr = immediate "instanceof" ref_type;
invoke_expr = interface_invoke_expr | special_invoke_expr | static_invoke_expr |
    virtual_invoke_expr;
static_invoke_expr = "staticinvoke" "[" method_signature "](" immediate_list ")";
interface_invoke_expr = "interfaceinvoke" immediate ".[" + method_signature "]"
    "(" immediate_list ")";
special_invoke_expr = "specialinvoke" immediate ".[" method_signature "]"
    "(" immediate_list ")";
virtual_invoke_expr = "virtualinvoke" immediate ".[" method_signature "]"
    "(" immediate_list ")";

immediate_list = immediate | immediate immediate_next_list | ;
immediate_next_list = "," immediate immediate_next_list | ;
new_array_expr = "new" type "[" immediate "]";
new_expr = "new" ref_type "()";
new_multi_array_expr = "new multiarray " type sized_dims empty_dims;

sized_dims = "[" immediate "]" next_sized_dims;
next_sized_dims = "[" immediate "]" next_sized_dims | ;
empty_dims = "[]" empty_dims | ;
unop_expr = length_expr | neg_expr;
length_expr = "length" immediate;
neg_expr = "-" immediate;
array_ref = immediate "[" immediate "]";
instance_field_ref = immediate ".[" field_signature "]";
static_field_ref = "[" field_signature "]";

method_signature = identifier "(" type_list "):" type;
field_signature = identifier ":" type;
type_list = type type_next_list | ;
type_next_list = "," type type_next_list | ;
caught_exception_ref = "@caughtexception";
parameter_ref = "@parameter" int_constant;
this_ref = "@this";

label = identifier;
local = identifier;
constant = double_constant | float_constant | int_constant | long_constant |
    string_constant | null_constant;
type = int_type | long_type | float_type | double_type | ref_type |
    stmt_address_type | void_type;
exception_range = ".catch" ref_type "from" label "to" label "using" label;
```

Figure 2: Grammar for external Jimple (part 2 of 2.)

```
until the code stops changing
  perform constant and copy propagation
  perform dead code elimination
```

Jimple is to be used as an intermediate representation on which to perform optimizations. This step, however, made us develop optimizations right away just in order to produce Jimple. In doing so, we partially tested our framework's usability, and the results are encouraging. The analyses (*reaching definitions*, *using definitions*, and *available copies*) and transformations were extremely straightforward to implement. (Definitions for these basic analyses can be found in [Muc97] and [ASU86]).

Here are some examples of compacting some code produced from step 1:

```
  Before              After

op0 = a
op1 = b             x = a + b
op0 = a + b
x = op0

op0 = a             call f(a, b)
op1 = b
call f(op0, op1)
```

Our cleanup algorithm is somewhat time consuming because data flow analysis is time consuming. One might argue that no such analyses are needed because the Jimple code could be collapsed into aggregated expressions as it is produced, avoiding all these writes and reads from temporary locals. This proposed algorithm would fail, however, to deal with the special case of expression computations spanning basic blocks. Spanning expressions depend on values to be stored in specific stack positions, and so explicit reads and writes to the locals representing these stack positions are required to produce equivalent Jimple.

Consider the following example:

```
 iload 0    iload 2
 iload 1    iload 3
        \/
        /\
 iadd        imul
 istore 4    istore 4
 ...         ...
```

In this case it is impossible to produce more compact Jimple statements to represent this code, since the arguments of iadd and imul are variable. Our proposed algorithm, however, correctly produces equivalent typeless Jimple code:

```
 op0 = a        op0 = c
 op1 = b        op1 = d
           \/
           /\
 op0=op0+op1   op0=op0*op1
 e=op0         e=op0
 ...         ...
```

Another issue worth considering is the placement of this compaction step. Clearly it could be performed at the end, when the typed Jimple is produced, and possibly yield better results since no further transformations occur. According to our experiments, however, we determined that placing the compaction at the second step yields the best execution speed and that the increase in code size or number of locals is negligeable in comparison (see figure 3.) If the size of the code is critical, then the few extra statements produced in the jimplification process could be eliminated in an optimizer based on Jimple anyway.

Note that a significant speed-up is achieved for the latter two benchmarks (61% and 64% faster, respectively). This occurs because the algorithms used in subsequent steps execute faster by dealing with a smaller set of statements and local variables.

3. **Splitting the Local Variables** Having generated typeless compact Jimple code, all that remains is to type it. Unfortunately, typing can not be performed directly on the code at this point. In the example[1]

   ```
   op0 = "hello world";
   string = op0;
   op0 = 1;
   unit = op0;
   ```

   the variable `op0` is used as both a `String` and an `int`. Obviously one can not assign a single type to this variable without causing type conflicts. This type re-use occurs in Jimple because it is allowed in the Java bytecode. The Java Virtual Machine specification allows local variables and stack positions to be used as different types, as long as it is done consistently so that no type conflicts occur. This

[1] Note that this code can not actually be generated by step 2 since it is not compact.

| | baf.Method | | | jimple.StmtGraph | | | DustV2.Playfield | | |
|---|---|---|---|---|---|---|---|---|---|
| | locals | stmts | time ratio | locals | stmts | time ratio | locals | stmts | time ratio |
| compacting:step 2 | 103 | 216 | 1.00 | 52 | 261 | 1.00 | 234 | 1958 | 1.00 |
| compacting:step 5 | 107 | 216 | 1.17 | 58 | 252 | 1.61 | 257 | 1940 | 1.64 |

Figure 3: The effect of compacting the code at different steps for three different benchmarks. The counts reported above indicate the final number of locals and statements in the jimplified code. The *time ratio* compares the execution of time to execution time for the jimplification when the clean-up is in step two.

property is enforced by the type analysis stage of the classfile verification[LY96].

We can transform the preceeding code into typable code by simply renaming the last two references of `op0` to `op1`:

```
op0 = "hello world";
string = op0;
op1 = 1;
unit = op1;
```

This idea of renaming references can be generalized, so that local variables are prevented from being re-used at all. Avoiding all re-uses would cause a split of `x` in the following example, even if it is used as the same type.

```
x = 5;          x1=5;
use(x);         use(x1);
x = 10;         x2=10;
use(x);         use(x2);

before          after
```

Although this might seem wasteful, splitting local variables in this manner guarantees, by definition, that they are not re-used. Hence, they are not re-used as different types and are now eligible to be typed.

Our intuitive notion of a local variable's multiple uses is captured by the notion of *webs*[Muc97]. A web is a set $S$ of references (both definitions and uses in the regular sense[2]) which is minimal and closed. Closed in the sense that for every definition $d$, the regular uses of $d$ are in $S$, and for all uses $u$, the possible definitions are also in $S$. Minimal in the sense that it should not have a strict subset which is closed.

Given that we have the analyses of *reaching definitions* and using definitions available, building the set of webs for a program is straightforward:

```
for every definition d
  if d is not in a web
    W = new web
    S = new set
    S.add(d)

    iterate over x in S while S is not empty
      W.add(x)
      x is def: add all uses of x to S
      x is use: add all defs of x to S
```

In plain words, the action of this algorithm is as follows. Take a particular definition, and put it in a web. Take all of the uses (in the *using definitions* sense) of this definition, and add them to the web. For every use added to the web, add all the possible definitions to the web. Iterate these last two sentences until a fixed point is reached, at which point all of these uses and definitions define a particular *use* of the variable, as we are interested in finding.

With these webs at our disposal, the splitting algorithm is simply:

```
construct the webs of uses and definitions
split the locals so that every web has its
  own associated local
```

Here is the result of applying this splitting algorithm on an example with multiple uses and definitions.

```
      Before              After

x = 1      x = 2     x1 = 1   x1 = 2
     \/                   \/
     y=x                y = x1
     x = 3              x2 = 3
     y = y + x          y = y + x2
     print(x)           print(x2)
```

4. **Typing the Local Variables** Now that we have Jimple code which *can* be typed, we can

[2]Like on the right hand side of an assignment statement, when the variable is read.

worry about *how* to type it. A straightforward approach to this problem is to build a set of type restrictions for every local variable, based on how the local variables are used in the code. Since all local variables must be defined before they are used (this is guaranteed by [LY96]) we decided to restrict the set of references to consider to only the definitions. In the case of primitive types (e.g. `int`, `double`), the type restrictions are in fact type equalities. These equalities will always be consistent, because code such as

```
x = 1        x = 2.5
      \/
     print(x)
```

can not be generated from the previous step, since the verifier would reject it before step 1 is reached.

Typing reference local variables is a more interesting problem. An assignment to a reference local variable set the type restriction to be that *the left hand side is a superclass of the right hand side.* Given a set of these restrictions, a simple solution is the *least common superclass.* For example, if `B` and `C` extend `A`, the variable x in

```
x = new B     x = new C
        \/
     print(x.toString())
```

could be directly typed as `A`, the least common superclass of `B` and `C`. Note that collecting type restrictions and assigning types must be performed iteratively. This requirement is illustrated in the following example.

```
x = new C;
y = null;

label1:
    x = y;
    y = new B;
    goto label1;
```

The type of `x` depends on the type of `y` which is unknown when the first assignment is visited. Only after having determined the type of `y` to be `B`, can `x = y` be visited and `x` be properly assigned a type. Here is a formal description of the algorithm we have just implicitly described.

```
assign the type unknown to all variables

until there are no more type changes
  for every definition d of a local l
    let rt be the type of the right hand side
    type(l) = type(l).merge(rt)

definition of merge(u, v):
  if u and v are primitives and equal: u
  if u and v are object references: least
    common superclass
  otherwise: type error
```

Although this algorithm works on most classfiles we have encountered, it fails in two specific situations.

- When dealing with `nulls`. The following code only throws an null pointer exception, but it should still be converted to correct Jimple.

  ```
  a = null;

  print(a[i]);
  ```

  Our algorithm however types `a` as a `java.lang.Object` since it only uses information about definitions. This causes a type conflict for the array reference `a[i]` since `a` is not an array.

- The second problem occurs when dealing with interfaces.

  ```
  class TypeFail
  {
      boolean condition;
      A a; B b; C c; D d;

      void f()
      {
          unknown x;

          if(condition)
              x = c;
          else
              x = d;

          x.a();
      }
  }

  interface A { void a(); }
  interface B { void b(); }
  interface C extends A, B {}
  interface D extends A, B {}
  ```

  Once again, just looking at the definitions is not sufficient, because the classes `C` and `D` here have two different common superclasses. Choosing the wrong one produces a type conflict since the method `a()` cannot be called on an instance of interface `B`.

These problems with our algorithm indicate that we incorrectly simplified the solution when we decided to only consider definitions. Instead, we should take into account every reference to a local variable. In the above examples, using a as an array would require that a be an array, and calling a() on x would require that x be an instance of a subclass of A.

This new algorithm is currently being developed by a member of our research group, and is turning out to be more difficult than expected. A future paper will present the complete typing solution when it is reached.[3]

5. **Packing the Local Variables**

   At this point, we have compacted and typed our stackless 3-address code, and it seems that we might be done. Unfortunately, we are not, for step 3 has uncompacted some aspects of the code. In addition to having the least number of statements possible, we require that the Jimple code produced we have the smallest number of locals too. And step 3 has potentially resulted in an explosion of locals (see figure 11.) The following example illustrates this problem.

```
x = y + z;            x1 = y + z;
print(x);             print(x1);
x = a + b;            x2 = a + b;
print(x);             print(x2);

before splitting      after splitting
```

   The final step for our transformation process requires that we reorganize the local variables so that they are re-used whenever possible. This is the opposite problem of step 3. Packing local variables in this manner is similar to the problem of register allocation. The only difference is that we start off with a fixed set of locals which can be arbitrarily large and we are attempting to minimize their use, as opposed to attempting to map the local variables onto a fixed set of registers.

   The algorithm we use is a tweaked register allocation algorithm based on Chaitin's graph coloring scheme as described in [ASU86]:

```
construct an interference graph G for the locals

while there is a local to color
  let x be the local with the most interferences
  remove x from the graph and assign it a color
    such that its neighbouring locals have a
    different color, using an already assigned
    color if possible

assign a local variable to each color, and
  rename all the references appropriately.
```

   After having performed this algorithm, we finally have transformed our Java bytecode into correct Jimple code which is *stackless, typed* and *compact*. Hence our initial requirements for the transformation to this 3-address code intermediate representation are sastified.

The next two sections illustrate the jimplification process with a complete example, and give concrete results on its performance.

### 4.2 An Example

Here is an example of the entire jimplification process on a method called stepPoly which computes a simple piece-wise defined function. The original Java code is in figure 4, and the bytecode on which the jimplification process is performed is in figure 5. Figures 6 through 10 are snapshots of the code which is internally produced after every step of the transformation.

### 4.3 Preliminary Results

Figure 11 presents some results of the jimplification process on three different benchmarks. baf.Method and jimple.StmtGraph come from the Jimple framework, whereas DustV2.Playfield comes from a numerical application which simulates electric particle motion.

The numbers presented are the local variable and statements counts of the entire classfile, after a given step of the transformation has been performed. The time percentage indicates the amount of execution time spent for this stage.

There are two interesting facts to note:

- After step 2 is performed, the number of statements consistently drops by roughly 40%.
- Local packing is essential for good code generation. For the last benchmark, it reduces the local variable count by 81%!

[3] In the mean time, you can contact Etienne Gagnon at gagnon@sable.mcgill.ca for more details.

| | baf.Method | | | jimple.StmtGraph | | | DustV2.Playfield | | |
|---|---|---|---|---|---|---|---|---|---|
| | locals | stmts | time(%) | locals | stmts | time(%) | locals | stmts | time(%) |
| step 1: verbose typeless | 110 | 357 | 34 | 44 | 430 | 29 | 217 | 3261 | 24 |
| step 2: compaction | 87 | 216 | 34 | 34 | 261 | 31 | 178 | 1958 | 34 |
| step 3: local splitting | 134 | 216 | 10 | 181 | 261 | 11 | 1262 | 1958 | 11 |
| step 4: type inference | 134 | 216 | <1 | 181 | 261 | <1 | 1262 | 1958 | <1 |
| step 5: local packing | 103 | 216 | 15 | 54 | 261 | 24 | 234 | 1958 | 27 |

Figure 11: Statistics on the local and statement count after each step in jimplification process.

```
public int stepPoly(int x)
{
    if(x < 0)
    {
        System.out.println("error");
        return -1;
    }
    else if(x <= 5)
        return x * x;
    else
        return x * 5 + 16;
}
```

Figure 4: `stepPoly` in its original Java form.

```
public int stepPoly(int)
{
    .maxstack 2
    .maxlocals 2

    iload 1;
    iconst 0;
    if_icmpge label0;

    getstatic [java.lang.System.out:
      java.io.PrintStream];
    sconst "error";
    virtualinvoke [java.io.PrintStream.
      println(java.lang.String):void];
    iconst -1;
    ireturn;

 label0:
    iload 1;
    iconst 5;
    if_icmpgt label1;

    iload 1;
    iload 1;
    imul;
    ireturn;

 label1:
    iload 1;
    iconst 5;
    imul;
    iconst 16;
    iadd;
    ireturn;
}
```

Figure 5: `stepPoly` in its bytecode form.

```
public int stepPoly(int)
{
    unknown op0, l0, op1, l1;

    l0 := @this;
    l1 := @parameter0;
    op0 = l1;
    op1 = 0;
    if op0 >= op1 goto label0;

    op0 = java.lang.System.out;
    op1 = "error";
    op0.println(op1);
    op0 = -1;
    return op0;

 label0:
    op0 = l1;
    op1 = 5;
    if op0 > op1 goto label1;

    op0 = l1;
    op1 = l1;
    op0 = op0 * op1;
    return op0;

 label1:
    op0 = l1;
    op1 = 5;
    op0 = op0 * op1;
    op1 = 16;
    op0 = op0 + op1;
    return op0;
}
```

Figure 6: `stepPoly` after step 1. Verbose and typeless.

```
public int stepPoly(int)
{
    unknown op0, l0, l1;

    l0 := @this;
    l1 := @parameter0;
    if l1 >= 0 goto label0;

    op0 = java.lang.System.out;
    op0.println("error");
    return -1;

 label0:
    if l1 > 5 goto label1;

    op0 = l1 * l1;
    return op0;

 label1:
    op0 = l1 * 5;
    op0 = op0 + 16;
    return op0;
}
```

Figure 7: `stepPoly` after step 2. It has been compacted.

```
public int stepPoly(int)
{
    unknown r1, r2, r0, r3, r4, r5;

    r0 := @this;
    r1 := @parameter0;
    if r1 >= 0 goto label0;

    r4 = java.lang.System.out;
    r4.println("error");
    return -1;

 label0:
    if r1 > 5 goto label1;

    r2 = r1 * r1;
    return r2;

 label1:
    r5 = r1 * 5;
    r3 = r5 + 16;
    return r3;
}
```

Figure 8: `stepPoly` after step 3. The locals have been split to match the webs.

```
public int stepPoly(int)
{
    java.io.PrintStream r1;
    Example r0;
    int i0, i1, i2, i3;

    r0 := @this;
    i0 := @parameter0;
    if i0 >= 0 goto label0;

    r1 = java.lang.System.out;
    r1.println("error");
    return -1;

 label0:
    if i0 > 5 goto label1;

    i1 = i0 * i0;
    return i1;

 label1:
    i3 = i0 * 5;
    i2 = i3 + 16;
    return i2;
}
```

Figure 9: `stepPoly` after step 4. It is now typed.

```
public int stepPoly(int)
{
    java.io.PrintStream r1;
    Example r0;
    int i0;

    r0 := @this;
    i0 := @parameter0;
    if i0 >= 0 goto label0;

    r1 = java.lang.System.out;
    r1.println("error");
    return -1;

 label0:
    if i0 > 5 goto label1;

    i0 = i0 * i0;
    return i0;

 label1:
    i0 = i0 * 5;
    i0 = i0 + 16;
    return i0;
}
```

Figure 10: `stepPoly` after step 5. The locals have been packed and this code is ready for optimizations.

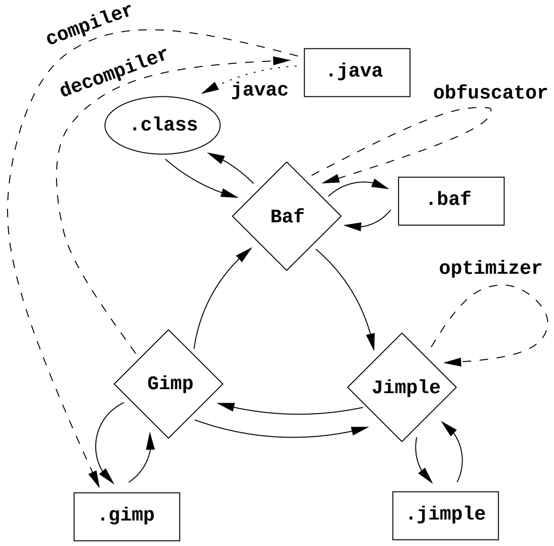


Figure 12: The Jimple intermediate representation is at the heart of the Sable research projects at McGill. The diamonds indicate internal API representations, whereas the rectangles indicate external ASCII text representations.

## 5 Current Uses

The Jimple intermediate representation is currently at the heart of the Sable research projects at McGill. As shown in Figure 12, we also have two other representations, *Gimp* and *Baf*. Gimp is used when we wish to form large aggregated expressions which can be used in the conversion back to bytecode and in the decompiler. Baf is a direct abstraction of Java bytecode, and is used to shield us from the encoding issues of dealing with Java classfiles directly.

Also note that Jimple is the central representation on which we perform analysis and transformation. We are currently working on a wide variety of pointer analyses, side-effect analyses, and type approximation algorithms. The results of these analyses will be used to optimize Jimple itself by applying traditional scalar optimizations like code motion, redundant and dead code elimination, and optimization of virtual method calls. We also feel that it will be useful to encode the analyses results as attributes in the resulting class files. In the case of JIT compilers, pre-computed dataflow information can be used to aid in register allocation, the elimination of extra array bounds checks, and the optimization of checks for exceptions[HAKN97]. In the case of ahead-of-time compilers, the attributes can be used instead of re-evaluating the dataflow information (in fact the ahead-of-time compiler would not even need to implement the analysis). Finally, these attributes may also be used to provide hints to a run-time system that can lead to dynamic optimizations.

## 6 Conclusions and Future Work

In this paper we presented Jimple, a 3-address representation for Java bytecode. We motivated the need for such a representation, and showed how to convert Java bytecode to Jimple.

It should be noted that many other Java translators probably also have to solve similar problems because of the stack-oriented nature of Java bytecode. This is particularly true when the target language is traditional native code, or other representations that do not use an expression stack. Further, other intermediate forms for Java have also been proposed. One such proposal is *Slim Binaries* which are based on abstract syntax trees that can be compressed into a very dense representation leading to smaller binaries than class files [KF97]. The authors argue that Slim Binaries are also well suited to all levels of optimization. The Jimple representation is not aimed at minimizing code size, but is designed for ease of use (and familiarity) for compiler writers who like to work with 3-address type representations.

In designing Jimple we have concentrated on making a clean API that is easy to use for both analysis and transformations. Jimple has undergone several revisions based on the experience of building analysis and transformations, including those analyses and transformations outlined in Section 4. Further, in translating bytecode to Jimple, we have included techniques that minimize extra names and statements that are introduced by a naive translation from bytecode to 3-address code. By making this code publicly available we hope to provide an alternative to optimizing bytecode directly, and to encourage others to build on the Jimple framework.

Our next goals are to provide automatic translation of Jimple (plus analysis information) back to class files with attributes. We expect that some additional work will be required to ensure that we generate efficient stack code from Jimple. This will involve good generation for aggregated expressions, and effective packing of Jimple variables into bytecode variables and stack locations.

### Acknowledgements

We would like to thank Clark Verbrugge for starting the Jimple project and providing us with initial implementations of Jimple and Coffi which helped us get off the ground. Also, the elegance of the Jimple API's is due in part to Etienne Gagnon. He advised us on its design and helped us see the light of object oriented programming.

### References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI '98)*, pages 106–115, Montreal, Canada, June 1998.

[HAKN97] Joe Hummel, Ana Azevedo, David Kolson, and Alex Nicolau. Annotating and optimizing the java bytecodes. In *Proceedings of the Intenational Workshop of Security and Efficiency Aspects of Java (MASCOTS '97)*, pages 9–10, Eilat, Israel, January 1997.

[HS92] Laurie J. Hendren and Bhama Sridharan. The SIMPLE AST - McCAT compiler. ACAPS Technical Note 36, ACAPS Research Group, McGill University, October 1992.

[KF97] Thomas Kistler and Michael Frannz. A tree-based alternative to java bytecodes. In *Proceedings of the Intenational Workshop of Security and Efficiency Aspects of Java (MASCOTS '97)*, pages 9–10, Eilat, Israel, January 1997.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[VR98] Raja Vallée-Rai. The Jimple Framework. Sable Technical Report 1, Sable Research Group, McGill University, February 1998.