McGill University
School of Computer Science
Sable Research Group

The Jimple Framework

Sable Technical Report No. 1

Raja Vallée-Rai

February 11, 1998

Abstract

The Java Virtual Machine is a stack based architecture. This makes compiler analyses and transformations more complex because of the implicit participation of the operand stack in instructions. To simplify these tasks, we developed an alternative representation for Java bytecode based on 3-address code and a framework to manipulate Java classfiles in this form. This report presents an overview of the Jimple framework and the algorithms used in the jimplification process.

1 Introduction

The Java programming language is an appealing language to work with because it embodies many desirable features that current standard programming languages lack. Perhaps the most hyped of these features is its platform neutrality. This feature is enabled by compiling Java programs only to Java bytecode, which is the machine code for the Java Virtual Machine (JVM), a stack based architecture. The program can then be distributed under this form to all platforms, and is interpreted or compiled to native code when it is run.

A stack-based architecture for the JVM is suitable for several reasons:

1. the code can be compactly represented and this reduces the transfer time of applications over the Internet,
2. the JVM is highly portable because every platform can simulate a stack, and
3. code generation is simple.

One big disadvantage of stack architectures, however, is that analyses and optimizations are more complex to design and implement. A simple instruction such as adding two variables can be spread over four bytecode instructions, and the effect of these four instructions must be tracked through their operations on the stack. It is not impossible to perform flow analysis directly on stack architectures, but it adds an unnecessary layer of complexity for each analysis and optimization. Jimple is a framework to manipulate Java classfiles and simplify the building of analyses on Java bytecode. It removes the extra layer of complexity due to the stack architecture by transforming Java bytecode to Jimple code, a form of 3-address code.

The Jimple intermediate representation was originally developed by Clark Verbrugge and Laurie Hendren [1], and is heavily based on SIMPLE [2]. The Jimple framework was developed as a general tool to be used by many applications, and in fact it is already being used to build a decompiler and gather statistics about classfiles. We will be using it to build a Java bytecode optimizer, one of the main goals of the Sable project. See figure 1 for an overview of the Sable project and how Jimple fits in.

The remainder of this documents presents overview of the Jimple framework, and the algorithms involved in the jimplification process.

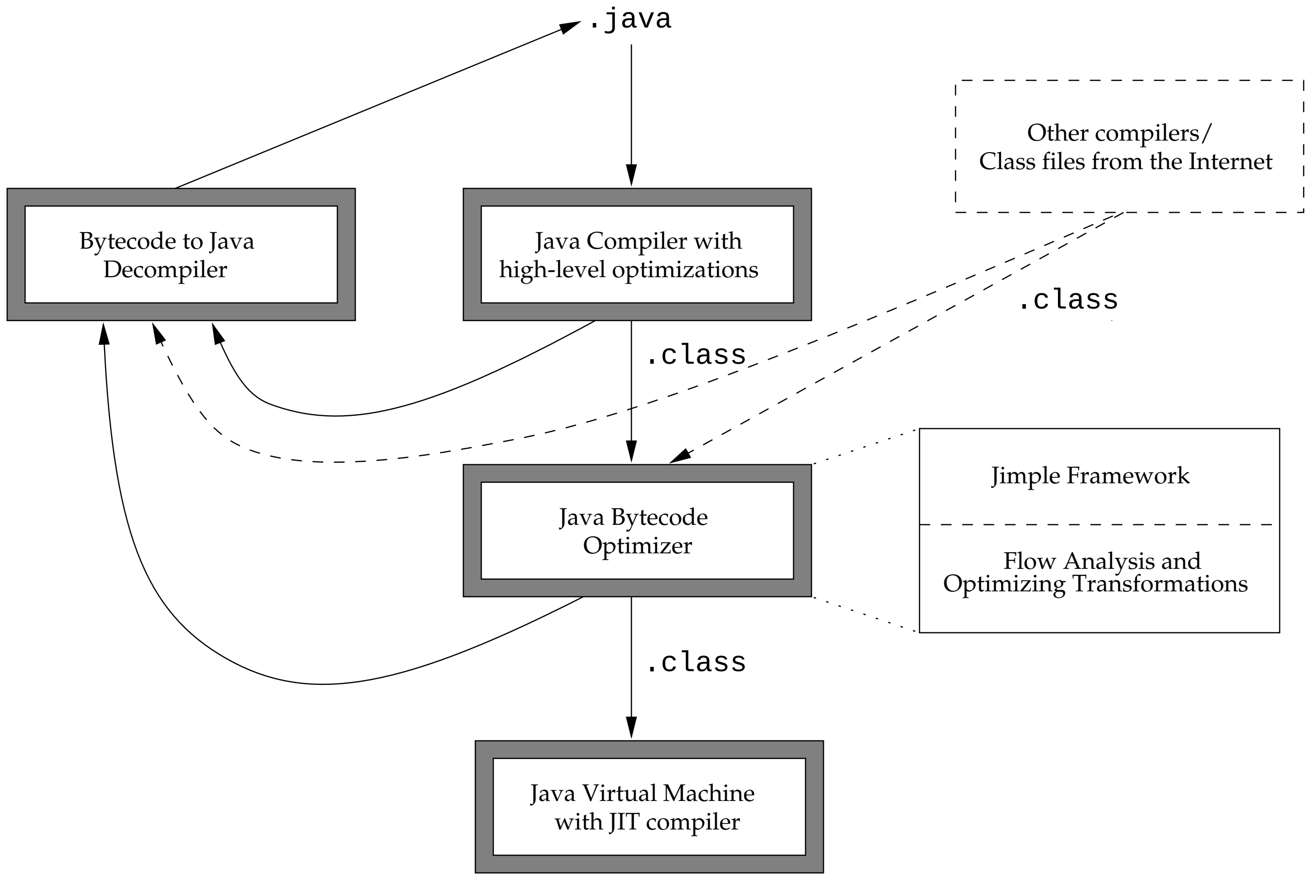

Figure 1: The Sable Project, and how Jimple fits in.

2 Framework Overview

Although we need a tool which transforms Java bytecode to Jimple code, we also need the ability to inspect and modify classfiles. Coffi [3] has this ability, but it forces us to deal with low-level Java Virtual Machine concepts such as the constant pool which is undesirable.

We have tried to create a simple and intuitive framework. The framework consists of Java classes such as `JClass`, `Method` and `Field`, each representing aspects of the Java classfile. Each class has methods which provide useful functionality such as enumerating the fields in a class, renaming a method, and changing the modifiers of a field.

While designing the API, we developed a series of coding conventions to simplify the API and make it as consistent as possible with existing object oriented practices. These conventions will be documented in an upcoming document entitled "Sable API Design Conventions".

A list of the major Jimple classes with their signatures follows. See figure 2 for the Jimple class hierarchy, and figure 3 for the containment hierarchy.

2.1 ClassManager

`ClassManager` is a container class for `JClass` objects, the Jimple representations of Java classes. Using `ClassManagers`, instead of lumping all `JClass` instances in a `static` set, allows multiple groups of classes to be simulateneously processed independently of each other. This will be useful when comparing the optimization possibilities of Jimple classes under different set of assumptions (e.g. intraclass optimizations vs interclass optimizations.)

```
public class  ClassManager
     extends java.lang.Object
{
     public ClassManager();

     public void addClass(JClass) throws AlreadyManagedException, DuplicateNameException;
     public JClass getClass(String);
     public JClass[] getClasses();
     public void loadClass(String) throws ClassFileNotFoundException, CorruptClassFileException,
        DuplicateNameException;
     public boolean managesClass(String);
     public void removeClass(JClass) throws IncorrectManagerException;
}
```

2.2 JClass

`JClass` objects are the Jimple representations of Java classes. They are mostly containers for objects of type `Field` and `Method` but also contain information about their names, superclasses and implemented interfaces.

Instances of `JClass` come in two flavors: complete or external. A `JClass` is complete if it has been loaded and jimplified through the `loadClass` method of a `ClassManager` object. It is external if it has not been loaded in this fashion, but instead been created by being referenced by a complete `JClass`. An external class contains all information gathered about it through such references. Methods such as `getFields()` and `getMethods()`, called on external classes do not throw exceptions but only return partial lists.

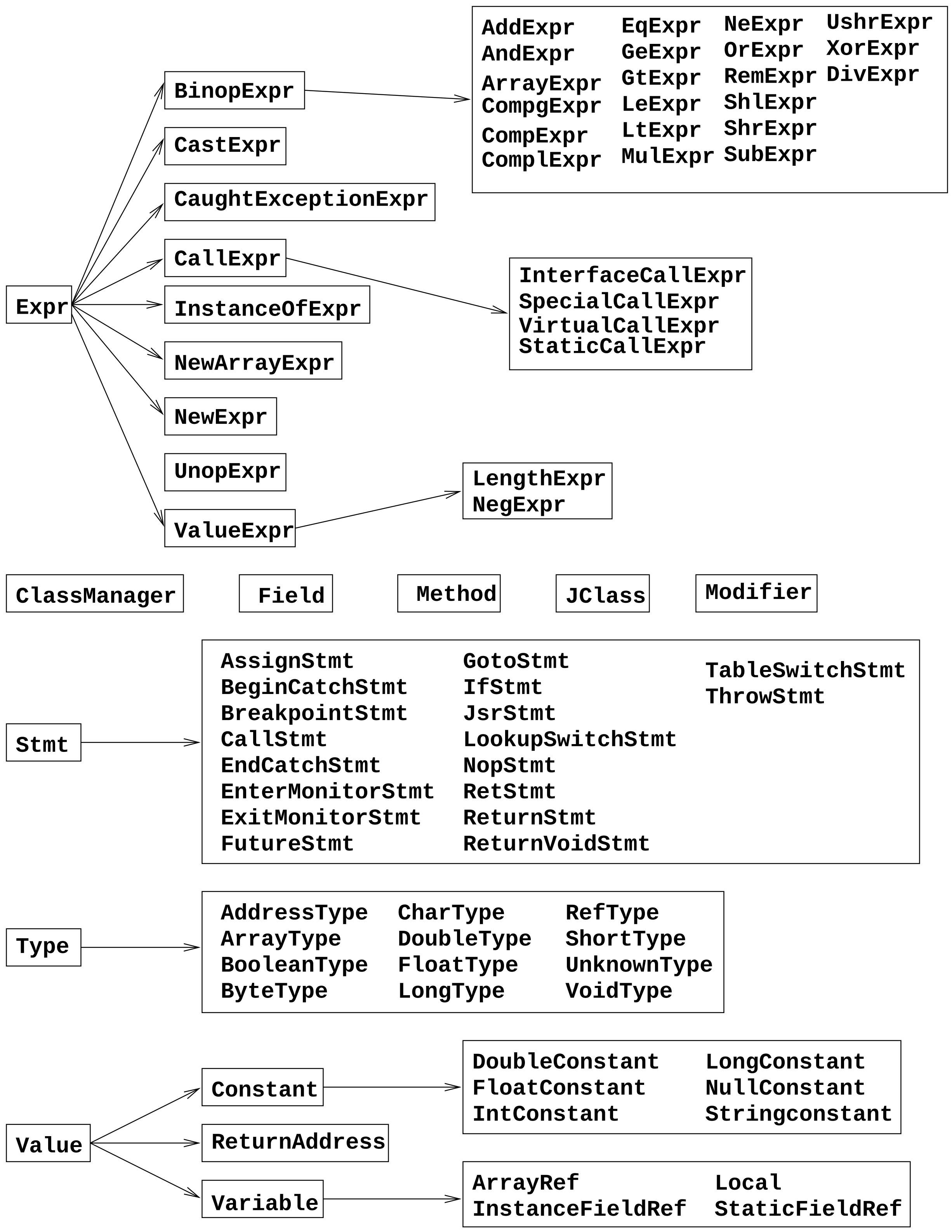

Figure 2: Jimple Class Hierarchy

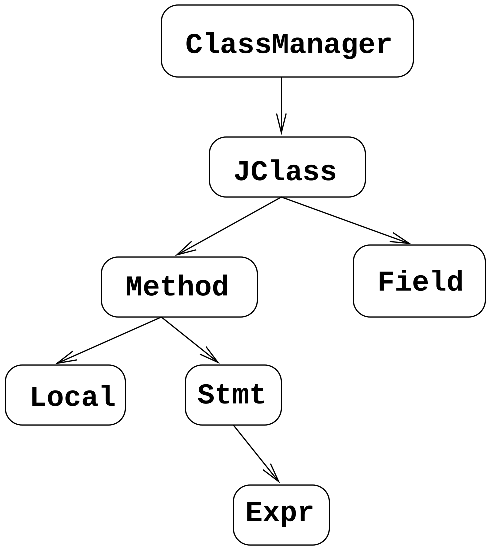

Figure 3: The containment hierarchy.

```
public class  JClass
     extends java.lang.Object
{
     public JClass(String);
     public JClass(String, int);

     public void addField(Field) throws AlreadyDeclaredException, DuplicateNameException;
     public void addInterface(JClass) throws DuplicateNameException;
     public void addMethod(Method) throws AlreadyDeclaredException, DuplicateNameException;
     public boolean declaresField(String);
     public boolean declaresMethod(String, Type[]);
     public Field getField(String) throws NoSuchFieldException;
     public Field[] getFields();
     public JClass[] getInterfaces();
     public ClassManager getManager() throws NotManagedException;
     public Method getMethod(String, Type[]) throws NoSuchMethodException;
     public Method[] getMethods();
     public int getModifiers();
     public String getName();
     public JClass getSuperClass() throws NoSuperClassException;
     public boolean hasSuperClass();
     public boolean implementsInterface(String);
     public boolean isExternal();
     public boolean isManaged();
     public void print(PrintStream);
     public void printSignature(PrintStream);
     public void removeField(Field) throws IncorrectDeclarerException;
     public void removeInterface(JClass) throws NoSuchInterfaceException;
     public void removeMethod(Method) throws IncorrectDeclarerException;
     public void setExternal(boolean);
     public void setModifiers(int);
     public void setName(String) throws DuplicateNameException;
     public void setSuperClass(JClass);
}
```

2.3 Method

The methods of the `Method` class give access to all the aspects of methods in Java classes. Here are some examples:

- `getLocals()` returns the list of local variables used,
- `setParameterTypes(Type[])` sets the parameter types of the method,
- `getModifiers()` returns the modifiers of a method.

The Jimple representation of the Java code for a method can be retrieved with `getCode()`. This method returns a `List` of statements. See section 2.6 on `Stmts` for more information.

The `Method` class is also a container for `Locals`, for in Jimple, all locals must be explicitly typed and declared.

```
public class  Method
     extends java.lang.Object
{
     public Method(String, Local[], Type);
     public Method(String, Local[], Type, List);
     public Method(String, Local[], Type, int);
     public Method(String, Local[], Type, int, List);

     public void addException(JClass) throws AlreadyThrowsException;
     public void addLocal(Local) throws DuplicateNameException, AlreadyDeclaredException;
     public boolean declaresLocal(String);
     public List getCode();
     public JClass getDeclaringClass() throws NotDeclaredException;
     public JClass[] getExceptions();
     public Local getLocal(String) throws NoSuchLocalException;
     public Local[] getLocals();
     public int getModifiers();
     public String getName();
     public Local getParameter(int);
     public int getParameterCount();
     public Type[] getParameterTypes();
     public Local[] getParameters();
     public Type getReturnType();
     public Local getThisLocal() throws NoSuchLocalException, StaticMethodException;
     public boolean isDeclared();
     public boolean isStatic();
     public void printSignature(PrintStream);
     public void removeException(JClass) throws DoesNotThrowException;
     public void removeLocal(Local) throws IncorrectDeclarerException;
     public void setCode(List);
     public void setModifiers(int);
     public void setName(String);
     public void setParameters(Local[]);
     public void setReturnType(Type);
     public void setThisLocal(Local) throws StaticMethodException;
     public boolean throwsException(JClass);
}
```

2.4 Field

The `Field` class is the straightforward representation of a Java field in a class. Here is its signature:

```
public class  Field
     extends java.lang.Object
{
     public Field(String, Type);
     public Field(String, Type, int);

     public JClass getDeclaringClass() throws NotDeclaredException;
     public int getModifiers();
     public String getName();
     public Type getType();
     public boolean isDeclared();
     public void setModifiers(int);
```

```
    public void setName(String) throws DuplicateNameException;
    public void setType(Type);
    public String toString();
}
```

2.5 Local

Local variables in Jimple must be explicitly created and stored in a Method. Its signature:

```
public class  Local
    extends ca.mcgill.sable.jimple.Variable
{
    public Local(String, Type);

    public Method getDeclaringMethod() throws NotDeclaredException;
    public String getName();
    public Type getType();
    public boolean isDeclared();
    public void setName(String) throws DuplicateNameException;
    public void setType(Type);
    public String toString();
}
```

2.6 Stmt

Java code in a `Method` object is represented as `List` of `Stmts`. The `Stmt` class itself is an abstract class, and so the subclasses provide actual meaning. See figure 2 for a list of the possible `Stmts`. Here are some sample signatures:

```
public class  AssignStmt
    extends ca.mcgill.sable.jimple.Stmt
{
    public Variable lhs;
    public Expr rhs;

    public AssignStmt(Variable, Expr);

    public String toString(Map);
}

public class  LookupSwitchStmt
    extends ca.mcgill.sable.jimple.Stmt
{
    public Stmt defaultTarget;
    public Value key;
    public Stmt[] matchTargets;
    public int[] matchValues;

    public LookupSwitchStmt(Value, int[], Stmt[], Stmt);

    public String toString(Map);
}

public class  GotoStmt
    extends ca.mcgill.sable.jimple.Stmt
{
    public Stmt target;

    public GotoStmt(Stmt);
    public String toString(Map);
}
```

Given these signatures, we can represent the assignment of local variable `x` to `y` as `new AssignStmt(x, y)`, and if one has `Stmt` objects `label1, label2` and `default` then a lookupswitch on a local `x` can be constructed with

```
new LookupSwitchStmt(x, new int[] {1, 2}, new Stmt[] {label1, label2}, default);
```

2.7 Expr

A few `Stmts` classes such as `IfStmt` and `AssignStmt` expect an `Expr` as an argument when being constructed. `Exprs` express computations (with `AddExpr`, `MulExpr`, etc.) and references to local variables, fields and arrays (through `ValueExpr`, `InstanceFieldRef`, `StaticFieldRef` and `ArrayRef`.) See figure 2 for the subclasses of `Expr`. Some signatures:

```
public abstract class BinopExpr
     extends ca.mcgill.sable.jimple.Expr
{
     public Value x;
     public Value y;

     public BinopExpr();

}

public class  CastExpr
     extends ca.mcgill.sable.jimple.Expr
{
     public Type type;
     public Value x;

     public CastExpr(Value, Type);

     public String toString();
}
```

2.8 Modifier

We have followed the design of `java.lang.reflect.Modifier` and decided to not dedicate a modifier object for each modifier. Instead, modifiers are simply `ints` and are manipulated through the static methods and constants of the `Modifier` class. You combine two modifiers by "OR"ing them.

```
public class  Modifier
     extends java.lang.Object
{
     public static final int ABSTRACT;
     public static final int FINAL;
     public static final int INTERFACE;
     public static final int NATIVE;
     public static final int PRIVATE;
     public static final int PROTECTED;
     public static final int PUBLIC;
     public static final int STATIC;
     public static final int SYNCHRONIZED;
     public static final int TRANSIENT;
     public static final int VOLATILE;

     public static boolean isAbstract(int);
     public static boolean isFinal(int);
     public static boolean isInterface(int);
     public static boolean isNative(int);
     public static boolean isPrivate(int);
     public static boolean isProtected(int);
     public static boolean isPublic(int);
     public static boolean isStatic(int);
     public static boolean isSynchronized(int);
     public static boolean isTransient(int);
```

```
    public static boolean isVolatile(int);
    public static String toString(int);
}
```

2.9 Type

Types in Jimple are represented by subclasses of the `Type` class. These types cannot be explicitly constructed but are accessed through the static `v()` method of each subclass. This facilitates type comparisons by ensuring that two identical types are the same object and so the `==` operator can be used.

2.10 Value

As seen from figure 2 there are three different types of values:

`Constant` Subclasses of this class represent constant values. They are organized with the same scheme as types, in the sense that the `v()` method is used to access particular instances. Here is a sample subclass:

```
public class  IntConstant
    extends ca.mcgill.sable.jimple.Constant
{
    public final int value;

    public boolean equals(Constant);
    public String toString();
    public static IntConstant v(int);
}
```

`ReturnAddress` This is a special class used to store instances of `Stmt` in local variables, used solely to handler the `jsr` bytecode instruction.

`Variable` Subclasses of this class represent L-values, locations that can contain values. There are four subclasses of `Variable`: `ArrayRef`, `Local`, `InstanceFieldRef` and `StaticFieldRef`.

2.11 Exception Handling

In Java classfiles, sections of code to be trapped by exception handlers are indicated by explicit ranges. For example, the Java code

```
class Test
{
    public void f()
    {
        try {
            int x = 5;
            int y = 0;

            x = x / y;
```

```
        } catch(RuntimeException e)
        {
            System.out.println("Caught one!");
        }
    }
}
```

is compiled to the following Java bytecode, with explicit exception handler ranges:

```
Method void f()
   0 iconst_5
   1 istore_1
   2 iconst_0
   3 istore_2
   4 iload_1
   5 iload_2
   6 idiv
   7 istore_1
   8 goto 20
  11 astore_1
  12 getstatic #2 <Field java.io.PrintStream out>
  15 ldc #3 <String "Caught one!">
  17 invokevirtual #4 <Method void println(java.lang.String)>
  20 return
Exception table:
   from   to  target type
     0     8    11   <Class java.lang.RuntimeException>
```

Having an equivalent Jimple table in the Jimple representation would have its disadvantages. For example, whenever the insertion of a statement needs to be made, the exception trapping ranges would need to be checked and possibly expanded. This problem is solved by introducing Jimple exception trap delimiters. These are of the form: **`<begincatch exceptiontype,handler>`** and **`<endcatch exceptiontype>`**. Thus the above code in its Jimple form is:

```
class Test extends java.lang.Object
{
    public void f()
    {
        java.io.PrintStream $java.io.PrintStream0;
        int b, a, $int1, $int0;
        java.lang.RuntimeException $java.lang.RuntimeException0, c;
        java.lang.String $java.lang.String1;

     0  <begincatch java.lang.RuntimeException, 11>
     1  $int0 = 5;
     2  a = $int0;
     3  $int0 = 0;
     4  b = $int0;
     5  $int0 = a;
     6  $int1 = b;
     7  $int0 = $int0 / $int1;
     8  a = $int0;
     9  <endcatch java.lang.RuntimeException>
    10  goto 16;
    11  $java.lang.RuntimeException0 = @caughtexception;
    12  c = $java.lang.RuntimeException0;
    13  $java.io.PrintStream0 = java.lang.System.out;
    14  $java.lang.String1 = "Caught one!";
    15  call (virtual) $java.io.PrintStream0.println($java.lang.String1);
    16  return;
    }

    ...
}
```

3 Jimplification

Jimplifying a Java classfile consists of two phases: converting the infrastructure of the Java classfile, and converting the Java bytecode to Jimple code.

Converting the infrastructure is a straightforward technical exercise. The contents of the classfile is explored and for every method and field contained therein, a corresponding Jimple object is created. A detailed knowledge of the Java Virtual Machine's classfile format would have been required to implement this construction, but we used Coffi [3] which alleviated this requirement.

Converting the bytecode to Jimple code, however, is not as easy. The remainder of this section presents the algorithm used to do so.

3.1 Main Idea

The problem of converting Java bytecode to 3-address like code was initially studied by Clark Verbrugge [1]. We modified his algorithm because it failed on certain examples. We recently discoverd that ours fails as well on some examples, but we will soon be correcting it (see section 3.4.)

The main idea behind converting stack-based code to 3-address code is as follows: view the operand stack as an set of temporary variables, and convert each stack-based bytecode to 3-address code, transforming the implicit references to the stack to explicit references to these temporary variables. Here is an illustration:

```
iload 1         // load local variable x1 onto the stack
iload 2         // load local variable x2 onto the stack
iadd            // add the two top stack elements and push the result onto the stack
istore 1        // store the top stack element into local variable x1
```

Every bytecode in this example implicitly operates on stack elements. By declaring these stack elements explicitly, we can easily transform this code into the following 3-address code:

```
int $int0, $int1;   // temporary variables representing the two top elements of the stack

$int0 = x1;             // iload 1
$int1 = x2;             // iload 2
$int0 = $int0 + $int1   // iadd
x1 = $int0              // istore 1
```

A precise description of this algorithm follows.

3.2 The Algorithm

This algorithm is applied once for each Java method in a classfile.

Step 1: Build `instructionToSuccessors` Map

The first step is to construct a `Map`[1] between all instructions and their possible successors. Note that the concept of successors includes those instructions which are exception handlers for a given instruction. This is a straightforward process given the control flow graph of the Java method.

Step 2: Perform Pseudo Verification Flow-Analysis

The goal of this step is to construct three maps: `instructionToTypeStack`, `instructionToPostTypeStack` and `instructionToLocalArray`. With these maps the state of the operand stack before and after every bytecode instruction can be determined, as well as the state of the local variable array before any instruction. (Instances of the classes `TypeStack` and `TypeArray` are stacks of types and arrays of types, respectively. The term LocalArray refers to an instance of `TypeArray` used to store the types of the locals.)

These maps are constructed with the following flow analysis algorithm:

```
algorithm()
{
    Set visitedInstructions = new Set();
    List changedInstructions = new List();

    instructionToTypeStack.put(firstInstruction, TypeStack.emptyStack);
    instructionToLocalArray.put(firstInstruction, TypeArray of [parameter types
        (and type of 'this' if non-static method)]);

    visitedInstructions.add(firstInstruction);
    changedInstructions.add(firstInstruction);

    while(!changedInstructions.isEmpty())
    {
        Instruction ins = changedInstructions.remove(0);

        TypeStack postTypeStack;
        TypeArray postLocalArray;

        (postTypeStack, postLocalArray) = processFlow(ins,
            instructionToTypeStack.get(ins), instructionToLocalArray.get(ins));

        instructionToPostTypeStack.put(ins, postTypeStack);

        for all successors s in instructionToSuccessors.get(ins)
        {
            if(!visitedInstructions.contains(s))
            {
                // First time visiting.

                if(s is the beginning of an exception handler)
                {
                    instructionToTypeStack.put(s, empty typeStack with exception type
                        pushed on top)
                    instructionToLocalArray.put(s, postLocalArray);
                }
                else {
                    instructionToTypeStack.put(s, postTypeStack);
```

[1]The concept of a Map issimilar to that of a `java.util.Hashtable`.

```
                instructionToLocalArray.put(s, postLocalarray);
            }

            visitedInstructions.add(s);
            changedInstructions.add(s);
        }
        else {
            TypeStack newTypeStack,
                      oldTypeStack = instructionToTypeStack.get(s);
            TypeArray newLocalArray,
                      oldLocalArray = instructionToLocalArray.get(s);

            if(s is the beginning of an exception handler)
                newTypeStack = empty typeStack with exception type pushed on top;
            else
                newTypeStack = postTypeStack.merge(oldTypeStack);

            newLocalArray = postLocalArray.merge(oldLocalArray);

            if(!newLocalArray.equals(oldLocalArray) ||
                !newTypeStack.equals(oldTypeStack))
                changedInstructions.add(s);

            instructionToTypeStack.put(s, newTypeStack);
            instructionToLocalArray.put(s, newLocalArray);
        }
    }
  }
}

TypeStack.merge(TypeStack other)
{
    TypeStack newTypeStack = new TypeStack();

    for all elements i in this
        newTypeStack.put(this.get(i).merge(other.get(i)))
            // create a type stack which merges elements index-wise

    return newTypeArray;
}

TypeArray.merge(TypeArray other)
{
    TypeArray newTypeArray = new TypeArray();

    for all elements i in this
        newTypeArray.put(this.get(i).merge(other.get(i)))
            // create a type array which merges elements index-wise

    return newTypeArray;
}

Type.merge(Type other)
{
    if(other == this)
        return this;
    else if(other is a reference type && this is a reference type)
        return least common superclass of (other, this);
    else
        return unusuable;
}

(TypeStack, TypeArray) processFlow(Instruction ins, TypeStack typeStack,
    TypeArray localArray)
{
```

```
    Model the effect that the instruction 'ins' has on the stack operands,
    typeStack, and the local variable array, localArray.  Return the new
    TypeStack and TypeArray.

    e.g.

    switch(ins.byteCode)
    {
        case ISTORE:
            localArray = localArray.set(ins.arg, IntType.v());
            typeStack = typeStack.pop();
            break;

        case ALOAD:
            typeStack = typeStack.push(localArray.get(ins.arg))
            break;

        case ICONST_0:
            typeStack = typeStack.push(IntType.v());
            break;

        case IADD:
        case IMUL:
        case IDIV:
        case IREM:
        case ISHL:
        case ISHR:
        case IUSHR:
        case IAND:
            typeStack = typeStack.pop();
            typeStack = typeStack.pop();
            typeStack = typeStack.push(IntType.v());
            break;
        ...

    }

    return (typeStack, localArray);

}
```

Step 3: Generate Jimple Code

This step consists of generating actual Jimple code for every Java bytecode instruction, using the `Maps` built up in step 2. There are two essential rules for this generation:

1. Every implicit reference to a stack operand at position `x` is replaced with an explicit reference to a temporary variable named `"$" + (instructionsToTypeStack.get(ins)).get(x)).typeName() + x` or `"$" + (instructionsToPostTypeStack.get(ins)).get(x)).typeName() + x` depending on whether the stack operand is read from or written to. The incorporation of the type into the temporary variable name ensures that stack positions used as different types are associated to multiple temporary variables.

2. Every reference to a local variable numbered `x` of type `t` (which can be determined with the `TypeArray instructionsToLocalArray.get(ins))` is transformed into a reference to a Jimple local variable of type `t`. Thus the same local variable slot can also be associated to multiple local Jimple variables of different types.

Following these rules, each Java bytecode can be transformed into Jimple code on a case by case basis. Here are a few examples:

```
Stmt generateJimple(Instruction ins, TypeStack typeStack, TypeStack postTypeStack,
    TypeArray localArray)
{
    Stmt stmt;
    Expr e;
    Expr rhs;

    switch(ins.bytecode)
    {
        case ICONST_0:
            e = new ValueExpr(IntConstant.v(0));
            stmt = new AssignStmt(getLocalForStackOp(postTypeStack,
                postTypeStack.topIndex()), e);
            break;

        case ASTORE:
        {
            Local local =
                getLocalOfIndexedType(indexedTypeToLocal, ins.arg, typeStack.top());

            stmt = new AssignStmt(local, new ValueExpr(
                getLocalForStackOp(jmethod, typeStack, typeStack.topIndex())));
            break;
        }

        case IADD:
            rhs = new AddExpr(getLocalForStackOp(typeStack,
                typeStack.topIndex() - 1), Util.getLocalForStackOp(typeStack,
                typeStack.topIndex()));

            stmt = new AssignStmt(getLocalForStackOp(postTypeStack,
                postTypeStack.topIndex()), rhs);
            break;

    }

    return stmt;
}
```

3.3 Problem: Incorrect Jimple Code

Unfortunately, our algorithm does not work correctly on all Java bytecode. Here is an example. Suppose class `B` extends `A` and `C` extends `A`, and local variables 1 and 2 are of type `B` and `C` respectively. Now consider the pseudo Java bytecode sequence

```
if(...)
    aload 1
else
    aload 2

astore 3
```

With the given algorithm, this gets jimplified to:

```
if(...)
```

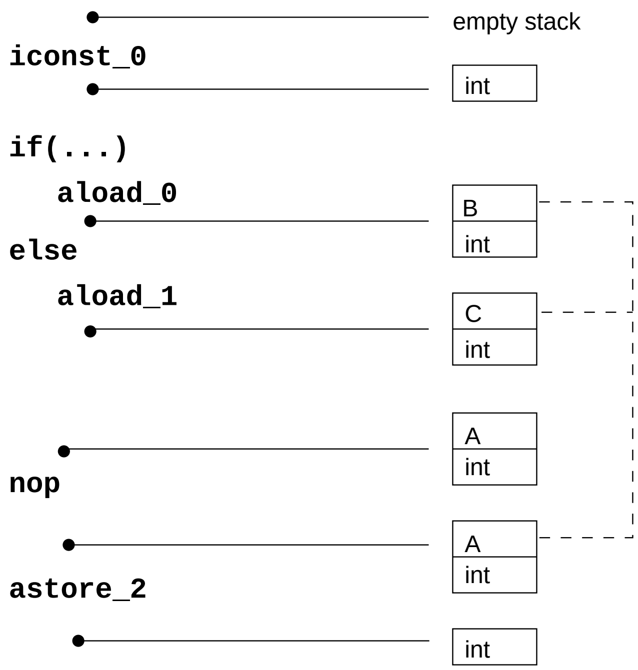

Figure 4: An example of linked slots.

```
    $B0 = B1;
else
    $C0 = C2;

A3 = $A0;
```

This is clearly incorrect because the variables `$B0`, `$C0` and `$A0` should be all referring to the same entity, a variable of type `A` like `$A0`.

3.4 Proposed Correction

The problem with our algorithm is that it computes the most specific types of the local variable and stack operand slots at a given program point without taking into account their future uses. Our proposed solution is to link all local variables and stack operands *slots* which should share the same type, and then perform the flow analysis. Note that the types of the stack operands are stored in the slots of type stacks, with one type stack per program point. Similarly for local variable types. It is the types in these slots that must be kept consistent across instructions. This concept is illustrated in figure 4.

The corrected algorithm is basically the algorithm in section 3.2 with the following modifications:

- Initially, assign to every local variable and stack operand the type "anything".
- Linked local variables and stack operand slots always have the same type.

- When assigning a type to a local variable or stack operand, merge the current type with the desired type. (for references this is the least common superclass)
- When changing the type of a slot, add the set of successors of the instructions which use that linked slot to the set `changedInstructions`.

Now, how are the slots linked? Intuitively, we want slots to be linked if they are correspond to the same instance of stack operand or local variable. Set of linked slots can be defined precisely as follows:

```
Map slotToLinkedSet;

for every instruction ins assigning a value to a slot S do
{
    Set<Slot> linkedSet;

    if(slotToLinkedSet.contains(S))
        linkedSet = slotToLinkedSet.get(S);
    else
    {
        linkedSet = new Set();
        slotToLinkedSet.put(S, linkedSet);
    }

    Set<Instruction> usingDefs = using definitions of the assignment in ins;

    for every instruction usedIns in usingDefs do
    {
        linkedSet.add(usedIns.slot(S));

        Set<Instruction> possibleDefs = reaching definitions of S at usedIns;

        for every instruction defIns in possibleDefs do
            linkedSet.add(defIns.slot(S));
    }
}
```

4 Future Work

4.1 Simplify Jimple Code

Consider the following Java code:

```
int x;

x = x + 1;
```

In Java bytecode, it might be represented as:

```
iload 1
iconst_1
iadd
istore 1
```

and given our algorithm for jimplifying Java bytecode, we would get:

```
int x;
int $int0, $int1;

$int0 = x;
$int1 = 1;
$int0 = $int0 + $int1;
x = $int0;
```

This Jimple code is correct, but note that `x = x + 1` would also be correct. In fact, it would be much better. In general, the code that our algorithm produces is too verbose, and could be considerably improved. Performing copy and constant propagation on the Jimple code would improve its quality and is something we will implement in the near future.

4.2 Generate Classfiles

The recreation of Java classfiles from Jimple objects is a two step process. First, the classfile infrastructure must be re-created. This is a technical exercise in satisfying the Java class file format and constraints, and is easily accomplished with the use of Coffi. Second, the 3-address like code must be transformed back to stack based code. Here is a straightforward approach to generate efficient code:

1. Generate naive stack code for each instruction,
2. Perform flow analysis on the stack code in order to eliminate the temporary variables that Jimple introduced.

As an illustration for the need of flow analysis, consider the following code:

```
temp = x + 1;
y = temp + 2;
```

This could be converted simply to

```
0 <--- x
1 <--- temp
2 <--- y

iconst_1
iload_0
iadd
istore_1

iload_1
iconst_2
iadd
istore_2
```

But note that the local variable `temp` in the 3-address code is a temporary variable which is just stored and then immediately reloaded. Through proper stack based flow analysis, we should be able to reduce the number of local variables needed and generate stack code like:

```
0 <--- x
1 <--- y

iconst_1
iload_0
iadd
iconst_2
iadd
istore_1
```

Other issues to consider are local variable allocation schemes since some Jimple local variables could share the same bytecode local variable slot. This is akin to the well studied register allocation problem.

5 Conclusion

The Jimple framework was developed as a general framework for inspecting and modifying Java classfiles. By representing Java bytecode as 3-address like code, we hope to simplify the task of writing analyses and transformations for our bytecode optimizer.

We still need to correct the jimplification algorithm, but the implementation is almost complete. Several students in our Sable group have already begun using Jimple succesfully in their own projects. These projects only investigate Java classfiles, but we expect to see some in the near future which produce modified classfiles as well.

References

[1] Clark Verbrugge and Laurie Hendren. Simplifying Java Bytecode for Compiler Analyses and Transformations. Internal Sable Memo. Sch. of Comp. Sci., McGill U., Montréal, Qué., Jul. 1997.

[2] Laurie J. Hendren and Bhama Sridharan. The SIMPLE AST - McCAT compiler. ACAPS Technical Note 36, Sch. of Comp. Sci., McGill U., Montréal, Qué., Oct. 1992.

[3] Clark Verbrugge. Coffi can be obtained at `http://www-acaps.cs.mcgill.ca/~clump`.

[4] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison-Wesley. p. 128-130.

A Where To Obtain Jimple

You can obtain our freely available Jimple framework from our web site at `http://www.sable.mcgill.ca`