Decompiling Java Bytecode: Problems, Traps and Pitfalls

Jerome Miecznikowski and Laurie Hendren
[jerome,hendren]@cs.mcgill.ca

Sable Research Group, School of Computer Science, McGill University

Abstract. Java virtual machines execute Java bytecode instructions. Since this bytecode is a higher level representation than traditional object code, it is possible to decompile it back to Java source. Many such decompilers have been developed and the conventional wisdom is that decompiling Java bytecode is relatively simple. This may be true when decompiling bytecode produced directly from a specific compiler, most often Sun's `javac` compiler. In this case it is really a matter of inverting a known compilation strategy. However, there are many problems, traps and pitfalls when decompiling arbitrary verifiable Java bytecode. Such bytecode could be produced by other Java compilers, Java bytecode optimizers or Java bytecode obfuscators. Java bytecode can also be produced by compilers for other languages, including Haskell, Eiffel, ML, Ada and Fortran. These compilers often use very different code generation strategies from `javac`.

This paper outlines the problems and solutions we have found in our development of `Dava`, a decompiler for arbitrary Java bytecode. We first outline the problems in assigning types to variables and literals, and the problems due to expression evaluation on the Java stack. Then, we look at finding structured control flow with a particular emphasis on issues related to Java exceptions and synchronized blocks. Throughout the paper we provide small examples which are not properly decompiled by commonly used decompilers.

1 Introduction

Java bytecode is a stack-based program representation executed by Java virtual machines. It was originally designed as the target platform for Java compilers. Java bytecode is a much richer and higher-level representation than traditional low-level object code. For example, it contains complete type signatures for methods and method invocations. The high-level nature of bytecode makes it reasonable to expect that it can be decompiled back to Java; all of the necessary information is contained in the bytecode. The design of such a decompiler is made easier if it only decompiles bytecode produced by specific compilers, for example the popular `javac` available with Sun's JDKs. In this case the problem is mostly one of inverting a known compilation strategy. The design of a decompiler is also simplified if it does not need to determine the exact types of all variables, but instead inserts spurious type casts to “fix up” code that has unknown type.

We solve a more difficult problem, that of decompiling arbitrary, verifiable bytecode. In addition to handling arbitrary bytecode, we also try to ensure that the decompiled code can be compiled by a Java compiler and that the code does not contain extraneous type casts or spurious control structures. Such a decompiler can be used to decompile bytecode that comes from many sources including: (1) bytecode from `javac`; (2) bytecode that has been produced by compilers for other languages, including Ada, ML, Eiffel and Scheme; or (3) bytecode that has been produced by bytecode optimizers. Code from these last two categories many cause decompilers to fail because they were designed to work specifically with bytecode produced by `javac` and cannot handle bytecode that does not fit specific patterns.

To achieve our goal, we are developing a decompiler called `Dava`, based on the `Soot` bytecode optimization framework. In this paper we outline the major problems that we faced while developing the decompiler. We present many of the major difficulties, discuss what makes the problems difficult, and demonstrate that other commonly used decompilers fail to handle these problems properly.

Section 2 of this paper describes the problems in decompiling variables, types, literals, expressions and simple statements. Section 3 introduces the problem of converting arbitrary control flow found in bytecode to the control flow constructs available in Java. Section 4 discusses the basic control flow constructions, while the specific problems due to exceptions and synchronized blocks are examined in more detail in Section 5. Related work and conclusions are given in Section 6.

2 Variables, Types, Literals, Expressions and Simple Statements

In order to illustrate the basic challenges in decompiling variables and their types, consider the simple Java program in Figure 1(a), page 4. Classes `Circle` and `Rectangle` define circle and rectangle objects. Both of these classes implement the `Drawable` interface, which specifies that any class implementing it must include a `draw` method.

To illustrate the similarities and differences between the Java representation and the bytecode representation, focus on method `f` in class `Main`. Figure 1(b) gives the bytecode generated by `javac` for this method.

2.1 Variables, Literals and Types

First consider the names and signatures of methods. All of the key information for methods originally from Java source is completely encoded in the bytecode. Both the method names and the type signatures are available for the method declarations and all method invocations. However, the situation for variables is quite different.

In the Java source each variable has a name and a static type which is valid for all uses and definitions of that variable. In the bytecode there are only untyped locations — in method `f` there are 4 stack locations and 5 local locations. The stack locations are used for the expression stack, while the local locations are used to store parameters and local variables. In this particular example, the `javac` compiler has mapped the parameter `i` to local 0, and the four local variables `c`, `r`, `d` and `is_fat` are mapped to locals 1, 2, 3 and 4 respectively. The mapping of offsets to variable names and the types of variables must be inferred by the decompiler.

Another complicating factor in decompiling bytecode is that while Java supports several integral data types, including `boolean`, `char`, `short` and `int`, at the bytecode level the distinction between these types is only made in the signatures for methods and fields. Otherwise, bytecode instructions consider these types as integers. For example, at `Label2` in Figure 1(b) the instruction `iload 4` loads an integer value for `is_fat` from line 16 in Figure 1(a), which is a boolean value in the Java program. This mismatch between many integral types in Java and the single integer type in bytecode provides several challenges for decompiling.

These difficulties are illustrated by the result of applying several commonly used decompilers. Figure 2 shows the output from three popular decompilers, plus the output from our decompiler, `Dava`. `Jasmine` (also known as the SourceTec Java Decompiler) is an improved version of `Mocha`, probably the first publicly available decompiler[10, 7]. `Jad` is a decompiler that is free for non-commercial use whose decompilation module has been integrated into several graphical user interfaces including `FrontEnd Plus`, `Decafe Pro`, `DJ Java Decompiler` and `Cavaj`[6]. `Wingdis` is a commercial product sold by WingSoft [16]. In our later examples we also include results from `SourceAgain`, a commercial product that has a web-based demo version[14].[1] Our tests used the most current releases of the software available at the time of writing this paper, namely Jasmine version 1.10, Jad version 1.5.8, Wingdis version 2.16, and SourceAgain version 1.1.

Each of the results illustrate different approaches to typing local variables. In all cases the variables with types `boolean`, `Circle` and `Rectangle` are correct. The major difficulty is in inferring the type for variable `d` in the original program, which should have type `Drawable`. The basic problem is that on one control path `d` is assigned an object of type `Circle`, whereas on the other, `d` is assigned an object of type `Rectangle`. The decompiler must find a type that is consistent with both assignments, and with the use of `d` in the statement `d.draw();`. The simplest approach is to always chose the type `Object` in the case of different constraints. Figure 2(a) shows that `Jasmine` uses this approach. This produces incorrect `Java` in the final line where the variable `object` needs to be cast to a `Drawable`. `Jad` correctly inserted this cast in Figure 2(c). `Wingdis` exhibits a bug on this example, producing no a variable for the original `d`, and incorrectly emitting a static call `Drawable.draw();`.

[1] The demo version does not support typing across several class files, so it is not included in our first figure.

```
public class Circle
       implements Drawable {
  public int radius;
  public Circle(int r)
    { radius = r; }
  public boolean isFat()
    { return(false); }
  public void draw()
    { // code to draw ... }
}

public class Rectangle
       implements Drawable {
  public short height,width;
  public Rectangle(short h, short w)
    { height=h; width=w; }
  public boolean isFat()
    { return(width > height); }
  public void draw()
    { // code to draw ... }
}

public interface Drawable {
  public void draw();
}

public class Main {
  public static void f(short i)
  { Circle c; Rectangle r; Drawable d;
    boolean is_fat;

    if (i>10)                    // 6
     { r =  new Rectangle(i, i); // 7
       is_fat = r.isFat();       // 8
       d = r;                    // 9
     }
    else
     { c = new Circle(i);        // 12
       is_fat = c.isFat();       // 13
       d = c;                    // 14
     }
    if (!is_fat) d.draw();       // 16
  }                              // 17

  public static void main(String args[])
    { f((short) 11); }
}
```

(a) Original Java Source

```
.method public static f(S)V
   .limit stack 4
   .limit locals 5
.line 6
   iload_0
   bipush 10
   if_icmple Label1
.line 7
   new Rectangle
   dup
   iload_0
   iload_0
   invokenonvirtual Rectangle/<init>(SS)V
   astore_2
.line 8
   aload_2
   invokevirtual Rectangle/isFat()Z
   istore 4
.line 9
   aload_2
   astore_3
   goto Label2
.line 12
Label1:
   new Circle
   dup
   iload_0
   invokenonvirtual Circle/<init>(I)V
   astore_1
.line 13
   aload_1
   invokevirtual Circle/isFat()Z
   istore 4
.line 14
   aload_1
   astore_3
.line 16
Label2:
   iload 4
   ifne Label3
   aload_3
   invokeinterface Drawable/draw()V 1
.line 17
Label3:
   return
.end method
```

(b) bytecode for method `f`

Fig. 1. Example program source and bytecode generated by `javac`

As shown in Figure 2(d), our decompiler correctly types all the variables and does not require a spurious cast to `Drawable`. The complete typing algorithm is presented in our paper entitled "Efficient Inference of Static Types for Java Bytecode"[5]. The basic idea is to construct a graph encoding type constraints. The graph contains *hard* nodes representing the types of classes, interfaces, and the base types; and *soft* nodes representing the variables. Edges in the graph are inserted for all constraints that must be satisfied by a legal typing. For example, the statement `d.draw();` would insert an edge from the soft node for `d` to the hard node for `Drawable`. Once the graph has been created, typing is performed by collapsing nodes in the graph until all soft nodes have been associated with hard nodes. In this case the soft node for `d` would be collapsed into the hard node for `Drawable`. There do exist bytecode programs that cannot be statically typed, and for those programs we resort to assigning types that are too general and inserting down casts where necessary. However, we have found very few cases

```
public static void f(short s)
{ Object object;
  boolean flag;
  if (s > 10)
    { Rectangle rectangle =
        new Rectangle(s, s);
      flag = rectangle.isFat();
      object = rectangle;
    }
  else
    { Circle circle =
        new Circle(s);
      flag = circle.isFat();
      object = circle;
    }
  if (!flag)
    object.draw();
}
```

(a) Jasmine

```
public static void f(short short0)
{ boolean boolea4;
  if (((byte)short0) <= 10)
    { Circle circle1=
          new Circle(short0);
      boolea4= circle1.isFat();
    }
  else
    { Rectangle rectan2=
        new Rectangle(((short)short0),
                      ((short)short0));
      boolea4= rectan2.isFat();
    }
  if (boolea4 == 0)
    Drawable.draw();
}
```

(b) Wingdis

```
public static void f(short word0)
{ Object obj;
  boolean flag;
  if (word0 > 10)
    { Rectangle rectangle =
        new Rectangle(word0, word0);
      flag = rectangle.isFat();
      obj = rectangle;
    }
  else
    { Circle circle =
        new Circle(word0);
      flag = circle.isFat();
      obj = circle;
    }
  if(!flag)
    ((Drawable) (obj)).draw();
}
```

(c) Jad

```
public static void f(short s0)
{ boolean z0;
  Rectangle r0;
  Drawable r1;
  Circle r2;

  if (s0 <= 10)
    { r2 = new Circle(s0);
      z0 = r2.isFat();
      r1 = r2;
    }
  else
    { r0 = new Rectangle(s0, s0);
      z0 = r0.isFat();
      r1 = r0;
    }
  if (z0 == false)
    r1.draw();
  return;
}
```

(d) Dava

Fig. 2. Decompiled code for method `f`

where such casts need to be inserted, and in general our approach leads to many fewer casts than simpler typing algorithms.

The decompiled code produced by `Wingdis`, Figure 2(b), demonstrates the difficulties produced by different integral types. This decompiler inserts spurious typecasts for all uses of the variable `short`. Furthermore, constants as well as variables must be assigned the correct integral type. For example, a call to method `f` with a constant value must be made as `f((short) 10);` in order to avoid a type conflict between the type of the argument (int) and the type of the parameter (short).

2.2 Expressions and Simple Statements

From our example we can also see that `javac` uses a very simple code generation strategy. Basically each simple statement in Java is compiled to a series of bytecode instructions, where the assumption is that the Java evaluation stack is empty before the statement executes and is empty after the statement executes. For example, consider the bytecode generated for statement 8 (see the line with `// 8` in Figure 1(a) and the bytecode generated at the directive `.line 8` in Figure 1(b)). In this case the object reference stored in local 2 is pushed on the stack, the isFat method is invoked, which pops the object reference and pushes isFat's return value, and finally the return value is popped from the stack and stored in local 4. The expression stack had height 0 at the beginning of the statement and height 0 at the end of the statement.

This straight forward code generation strategy makes it fairly simple for a decompiler to rebuild the statement. However, many other bytecode sequences could express the same computations. Consider the example in Figure 3. Figure 3(a) gives the original bytecode as produced by javac, whereas Figure 3(b) gives an optimized version of the bytecode. The optimized version uses 5 fewer instructions and 3 fewer locals.[2] An example of a simple optimization is found at line 7. At this point the second iload_0 instruction has been replaced with a dup instruction. A more complex optimization makes use of the expression stack to save the values. For example, rather than storing the result of line 7 and then reloading it at line 8, the value is just left on the stack. Furthermore, since this same value is needed later, its value is duplicated (third dup at line 7). Line 8 demonstrates that the return value from the call to isFat can just be left on the stack. The swap instruction at line 8 exchanges the boolean value on top of the stack with the object reference just below it. Line 9 stores the object reference from the top of the stack and Line 12 uses the boolean value that is now on top of stack for the infne test.

When the optimized code from Figure 3(b) is given to the other decompilers, they all fail because the bytecode does not correspond to patterns they expect (see Figure 4, page 8). Jasmine and Jad emit error messages saying that the control flow analysis fails and emit code that is clearly not Java. Wingdis emits code that resembles Java but is clearly not correct as the calls to the method isFat have been completely missed, and the type for the left operand of == is an object rather than a boolean. SourceAgain also produces something that looks like Java, but it is also incorrect since it allocates too many objects and has lost the boolean variable.

Our Dava decompiler produces exactly the same Java code as for the unoptimized class file, except for the names of the local variables. Figure 2(d) contains no variables starting with $, whereas in Figure 4(e) three variables do start with $. In our generated code we prefix variables with $ to indicate variables corresponding to stack locations in the bytecode.

Dava is insensitive to the input bytecode because it is built on top of the Soot framework which transforms the bytecode into an intermediate representation called Grimp[13, 15]. Soot begins by reading bytecode and converting it to simple three address statements (this intermediate form is called Jimple). When generating Jimple the stack locations become specially named variables. Soot then uses U-D webs to separate different variables that may share the same local offset in bytecode, and finally performs simple code cleanup and the typ-

[2] It should be noted that this is not a contrived example; it merely illustrates the problems we encountered when applying other decompilers to bytecode produced by Java bytecode optimizers (even very simple peephole optimizers) and to bytecode produced by compilers for other languages.

```
.method public static f(S)V
   .limit stack 4
   .limit locals 5
.line 6
   iload_0
   bipush 10
   if_icmple Label1
.line 7
   new Rectangle
   dup
   iload_0
   iload_0
   invokenonvirtual Rectangle/<init>(SS)V
   astore_2
.line 8
   aload_2
   invokevirtual Rectangle/isFat()Z
   istore 4
.line 9
   aload_2
   astore_3
   goto Label2
.line 12
Label1:
   new Circle
   dup
   iload_0
   invokenonvirtual Circle/<init>(I)V
   astore_1
.line 13
   aload_1
   invokevirtual Circle/isFat()Z
   istore 4
.line 14
   aload_1
   astore_3
.line 16
Label2:
   iload 4
   ifne Label3
   aload_3
   invokeinterface Drawable/draw()V 1
.line 17
Label3:
   return
.end method
```

(a) original bytecode

```
.method public static f(S)V
   .limit stack 4
   .limit locals 2
.line 6
   iload_0
   bipush 10
   if_icmple Label1
.line 7
   new Rectangle
   dup
   iload_0
   dup
   invokenonvirtual Rectangle/<init>(SS)V
   dup
.line 8
   invokevirtual Rectangle/isFat()Z
   swap

.line 9
   astore_1
   goto Label2

.line 12
Label1:
   new Circle
   dup
   iload_0
   invokenonvirtual Circle/<init>(I)V
   dup
.line 13
   invokevirtual Circle/isFat()Z
   swap

.line 14
   astore_1

.line 16
Label2:
   ifne Label3
   aload_1
   invokeinterface Drawable/draw()V 1

.line 17
Label3:
   return
.end method
```

(b) optimized bytecode

Fig. 3. Original bytecode as generated by `javac` and optimized bytecode

ing algorithm. Given the typed Jimple, an aggregation step rebuilds expressions and produces `Grimp`. `Grimp` is the starting point for our restructuring algorithms described in the next section.

3 Control Flow Overview

The last major phase of our decompiler recovers a structured representation for a method's control flow. There may be more than one structured representation for any given control flow graph (CFG), so in `Dava`, we focused on producing a correct restructuring that would be easy to understand. Other goals, such as fast restructuring or representing control flow with a restricted set of control flow statements, are possible but not explored in `Dava`.

For correctness, we use a graph theoretic approach and focused on the capabilities of the Java grammar. For us, the key question was: "For any given set of control flow features in the CFG, can we represent it with pure Java?" When answering this question we must consider the following:

```
public static void f(short s)
{ Object object;
  if (s <= 10) goto 24 else 6;
  expression new Rectangle
  dup 1 over 0
  expression s
  dup 1 over 0
  invoke Rectangle.<init>
  dup 1 over 0
  invoke isFat
  swap
  pop object
  expression new Circle(s)
  dup 1 over 0
  invoke isFat
  swap
  pop object
  if != goto 47
  object.draw();
}
```

(a) Jasmine

```
public static void f(short short0)
{ if ((((byte)short0) <= 10)?
      (Circle circle1= new Circle(short0)):
      (Rectangle rectan1=
          new Rectangle(
            ((short)short0), ((short)short0)))
      == false)
     { Drawable.draw();
     }
}
```

(b) Wingdis

```
public static void f(short word0)
{ Rectangle rectangle;
  if(word0 <= 10)
    break MISSING_BLOCK_LABEL_24;
  rectangle =
    new Rectangle(word0, word0);
  rectangle.isFat();
  Object obj;
  obj = rectangle;
  break MISSING_BLOCK_LABEL_38;
  Circle circle =
    new Circle(word0);
  circle.isFat();
  obj = circle;
  JVM INSTR ifne 47;
  goto _L1 _L2
_L1:
  break MISSING_BLOCK_LABEL_41;
_L2:
  break MISSING_BLOCK_LABEL_47;
  ((Drawable) (obj)).draw();
}
```

(c) Jad

```
public static void f(short si)
{ Object obj;
  Object tobj;
  Object tobj1;

  if( si > 10 )
    { Object tobj2;
      tobj = new Rectangle( si, si );
      tobj2 = ((Rectangle) tobj).isFat();
      obj = new Rectangle( si, si );
     }
  else
    { tobj = new Circle( si );
      tobj1 = ((Circle) tobj).isFat();
      obj = new Circle( si );
     }
  if( tobj1 == 0 )
     ((Drawable) obj).draw();
}
```

(d) SourceAgain

```
public static void f(short s0)
{ boolean $z0;
  Drawable r0;
  Rectangle $r1;
  Circle $r2;

  if (s0 <= 10)
    { $r2 = new Circle(s0);
      $z0 = $r2.isFat();
      r0 = $r2;
    }
  else
    { $r1 = new Rectangle(s0, s0);
      $z0 = $r1.isFat();
      r0 = $r1;
    }
  if ($z0 == false)
    r0.draw();
  return;
}
```

(e) Dava

Fig. 4. Decompiled code for optimized method `f`

1. Every control flow statement in Java has exactly one entry point, and one or more exit points.
2. Java provides labeled blocks, labeled control flow statements, and labeled breaks and continues. With these, it is possible to represent any CFG that forms a directed acyclic graph (DAG) in pure Java. Consider the following. We can topologically sort the statements from the bytecode representation of such a DAG and place a labeled block around the first node. We now represent any control flow from the first node to the second as a labeled break out of our newly created labeled block. Next, we place a labeled block around the first two statements, and represent any control flow going *to* the third statement as labeled breaks out of the second block. Similarly, we can place a labeled block around the first three statements, and so on. Although this will produce an ugly restructuring, it illustrates that it is possible to restructure any control flow DAG.
3. The representation of a strongly connected component in the CFG must include at least one Java language loop. There is no direct representation, then, for strongly connected components with two or more entry points, since there is no control flow statement in the grammar that supports more than one entry point. If such a strongly connected component is found, it must somehow be transformed to a semantically equivalent strongly connected component with only a single entry point.
4. The Java language provides exception handling with `try`, `catch`, and `finally` statements. Unfortunately, the Java bytecode exception handling mechanism is more flexible than these statements, and may produce control flow that is not directly expressible in the Java language.
5. The Java language provides object locking with `synchronized` statements. As with exception handling, the object locking mechanism in the Java bytecode specification is more flexible than the specification of the `synchronized` statement, and may produce lockings in the bytecode that are not directly expressible in the Java language.

For readability, we felt that a terse representation of control flow should be easier to understand than a diffuse one. In `Dava`, we attempt this secondary goal by building Java language statements that each represent as many of the CFG features as possible with the intention of minimizing the number of statements produced altogether. Although not necessarily an optimal solution, it has, in practice, yielded excellent results.

3.1 A Brief Introduction to SET Restructuring

The restructuring phase of `Dava` uses three intermediate representations to perform its function: 1) `Grimp`, a list of typed, unstructured program statements, which loosely corresponds to the method's bytecode instruction stream, 2) a CFG representing the control flow from the `Grimp` representation, and 3) a Structure Encapsulation Tree (SET)[9]. The `Grimp` representation is fed to the restructurer, which produces the CFG and the SET. The finished SET is very similar to an abstract syntax tree, and the final Java language output is obtained simply by traversing it.

The CFG is built by finding all the potential successors to each Grimp statement. All branches in Java bytecode are direct, so this is a straightforward task. The only novel feature of this CFG is that is distinguish edges representing normal control flow from those representing the throwing of an exception.

The SET is built in 6 phases. A more complete description can be found in our paper entitled "Decompiling Java Using Staged Encapsulation"[9]; here we provide a brief overview. Each phase searches for a specific type of feature in the CFG and produces structured Java language statements that can represent that feature. The Java statement is then bundled with the set of nodes (wrapped Grimp statements) from the CFG that would correspond to its body. Since every structured Java statement has only one entry point, we can usually use dominance to determine the body. For example, a `while` statement would consist of the appropriate condition expression plus those statements from the CFG that the condition dominates, minus those statements reachable by the control flow from the condition that escapes the loop. The structured bundle is then nested in the SET such that the set of statements in the bundle is a subset of those in its parent node and a superset of those in its children nodes. In this way the SET can be built up in any arbitrary order of node insertion. Note also that the properties searched for in the CFG (ie. dominance and reachability) are transitive, which guarantees us that the superset/subset relations between SET bundles and their children will always hold.

4 Basic Control Flow Constructs

A decompiler must be able to find `if`, `switch`, `while`, and `do-while` statements, labeled blocks, and labeled `break`s and `continue`s.

Many decompilers use reduction based restructuring. These work by searching the CFG for local patterns that directly correspond to those produced by Java grammar productions. When a pattern is found it is reduced to a single node in the CFG and the search is repeated. This process is iterated until no more reductions can be found. In general this approach is difficult because the library of patterns that are matched against does not cover all possible patterns in the CFG. At some point, one may not find any more reductions, but still have not reduced the program to a single structured statement.

In contrast, `Dava` searches for features in the control flow graph in order of how flexibly they can treated. For example, strongly connected components must be represented by loops, which is an inflexible requirement. Accordingly, the conditions of loops are to be found before the conditions of `if` statements.

4.1 Loops

The most general way to characterize cyclic behavior in the CFG is to begin by searching for the strongly connected components (SCC). For each SCC, we build a Java loop. By examining the properties of the entry and exit points in the SCC we can determine which type of Java loop (`while`, `do-while` or `while(true)`) is suitable for the structured representation. Once we know the type of loop, we know which statement in the CFG yields us the conditional expression (if any) for the structured loop, and we can find the loop body.

We know that for every iteration of a Java loop, if the loop is conditional, the condition expression must be evaluated, or if the loop is unconditional, the entry point statement must be executed. To find nested loops, we simply remove the condition statement, or the entry point statement, from the CFG and re-evaluate to see if any SCCs remain. This process is iterated until no more SCCs are found.

This process seems to be more robust than reduction based techniques. Consider the small, if somewhat contrived, example in figure 5, page 12. Method `foo()` has no real purpose other than to illustrate the performance of a restructurer on difficult, loop based control flow. The original Java source was compiled with `javac` and the resulting bytecode class was not modified in any way. This example has two interesting components, (1) the outer loop only executes if an exception is thrown, and (2) if the inner loop exits normally, the next statement that affects program state is the `return`.

We can see that only `Dava` produces correct, recompilable code, though it does not greatly resemble the original program. `Jad` alone produces code that is reminiscent of the original, but unfortunately it is neither correct nor recompilable.

We may encounter multi-entry point SCCs. Here the input does not directly correspond to a Java structured program, so all decompilers will output ugly Java code. There are several solutions, but all involve transforming the CFG. Our solution converts the multi-entry point SCC to a single entry point SCC by breaking the control flow to the original entry points and rerouting it to a dispatch statement. This dispatch then acts as the single entry point and redirects control to the appropriate destination.

4.2 Labeled Statements, Blocks, and `break` and `continue` Statements

As shown in section 3, page 7, labeled blocks can resolve any difficulties in restructuring control flow DAGs. In `Dava`, once we have found all the nodes for the SET from the CFG, we then determine if any of the control flow necessitates the introduction of labeled statements, labeled blocks, `break`s or `continue`s. Once this phase is done, we have fully restructured our target program.

One might expect control flow necessitating the use of these statements to present difficulties to pattern-based decompilers since (1) the code produced by these statements is not fully structured, and (2) human programmers rarely exercise these features. It seems, however, that much work has been done on this problem as several other decompilers, notably `Jad` and `SourceAgain`, deal well with producing labeled statements, blocks, breaks, and continues.

```
public int foo( int i, int j)
{
    while (true)
    {   try
        {   while (i < j)
                i = j++/i;
        }
        catch (RuntimeException re)
        {   i = 10;
            continue;
        }
        break;
    }
    return j;
}
```

(a) Original Java Source

```
public int foo(int i, int j)
{
    while(true)
        try
        {   while(i < j)
                i = j++ / i;
            break MISSING_BLOCK_LABEL_25;
        }
        catch(RuntimeException runtimeexception)
        {   i = 10;
        }
    return j;
}
```

(b) Jad

```
public int foo(int i, int j)
{
    RuntimeException e;
    for (i = j++ / i; i < j; i = j++ / i)
        /* null body */ ;
    return j;
    pop e
    i = 10;
}
```

(c) Jasmine

```
public int foo(int i, int j)
{
    while( i < j )
        i = j++ / i;
    return j;
}
```

(d) SourceAgain

```
    public int foo(int int1, int int2)
    {

// WingDis cannot analyze control flow
// of this method fully

B0:
        goto B3;
B1:
        try {
            goto B3;
B2:
            int1= int2++ / int1;
B3:
            if (int1 < int2)goto B2;
        }
B4:
        goto B8;
B5:
        catch (RuntimeException null)
    {
B6:
        int1= 10;
    B7:
        goto B3;
    }
```

(e) Wingdis

```
public int foo(int i0, int i1)
{
    int $i2;

    while (true)
    {   try
        {   if (i0 < i1)
            {   $i2 = i1;
                i1 = i1 + 1;
                i0 = $i2 / i0;
                continue;
            }
        }
        catch (RuntimeException $r2)
        {   i0 = 10;
            continue;
        }
        return i1;
    }
}
```

(f) Dava

Fig. 5. Decompiled code for method `foo()`

5 Exceptions and Synchronized Blocks

Java bytecode and the Java language treat exception handling in very different ways. Bytecode is simply a numbered sequence of virtual machine instructions. Here, exception handling is specified by a table, where each entry holds a starting instruction number, a finishing instruction number, a reference to an exception class, and a pointer to a handler instruction. If an exception is thrown, the virtual machine runs through the table checking to see if the current instruction is in the instruction range given by any of the table entries. If it is in range, and the thrown exception matches the table entry's exception class, then control is transferred to that entry's handler instruction.

In bytecode, regular control flow imposes few restrictions on exception handling. Control flow may enter or exit at any instruction within a table entry's area of protection, and does not have to remain constantly within that area once it enters. Multiple control flow paths may enter a single area of protection at different points, and different areas of protection may overlap arbitrarily. The handler instruction may be anywhere within the class file, limited by the constraints of bytecode verification, including within the table entry's own area of protection. Finally, more than one exception table entry may share the same exception handler. In short, exception handling in Java bytecode is mostly unstructured.

By contrast, exception handling in the Java language uses the `try`, `catch` and `finally` grammar productions and is highly structured. There is only one entry point to a `try` statement, control flow within it is contiguous, and each of these Java statements nests properly. There is no way to make `try` statements partially overlap each other. Also, each `try` must be immediately followed by a `catch` and/or a `finally` statement. There may be any number of `catch` statements but no more than one `finally`.

If an exception is thrown and is not caught in a `catch` statement, then the method in which this occurs must declare that it `throws` that exception. Method declarations must agree between subclasses and superclasses. Therefore, if some method m_1 declares a `throws` and overrides or is overridden by another method m_2, then m_2 must also declare the `throws`.

There is a complication to the `throws` declaration rule. Object locking is provided in Java with the `synchronized()` statement. If a thrown exception causes control to leave a `synchronized()` statement, the Java language specification requires that the object lock be released. This is accomplished in the bytecode by catching the exception, releasing the lock in the exception handler and finally rethrowing the exception. This exception handling should not be translated into `try catch` statements, but remains masked by the `synchronized()` statement. Consequently, `throws` that are to be implied by a `synchronized()` statement's exception handling are not explicitly put in the Java language representation, and therefore are also ignored in the method declaration.

There are numerous consequences from this "semantic gap" in exception handling. An area of protection must be represented by a `try` statement, and handlers by a `catch` or `finally`. However, a `try` statement has only one entry point. So, an area of protection with more than one entry point must be split into as many parts as there are entry points. Each of these new areas of protection share the same handler, but a `catch` statement can only be immediately preceded by a single `try`. To reconcile this, the handler statement (at least) must be duplicated for each area of protection. If two areas of protection overlap but neither fully encapsulates the other, we must break up at least one of the areas to allow the resulting `try` statements to either be disjoint or nest each other properly.

Although these problems do not normally appear in bytecode generated by `javac`, they still may arise in perfectly valid Java bytecode. Consider the example control flow graph in figure 6(a), page 15. Here, we created a class file by hand that has a straight line of statements `a b c d f` with two areas of protection. If a `RuntimeException` is thrown in area of protection [`b c`], control flow is directed to `g`. If, however, an `Exception` is thrown in area of protection [`c d`], control flow is directed to `e`.

We cannot simply represent the two areas as two `try` statements because they will not be able to nest each other properly. The correct solution to this problem is to break the two areas of protection into three `try` statements, and to split and aggregate their handlers into appropriate `catch` statements, as shown in the output from `Dava` in figure 6(b). Again, other decompilers seem to rely on the bytecode reflecting an already structured program, and produce incorrect output.

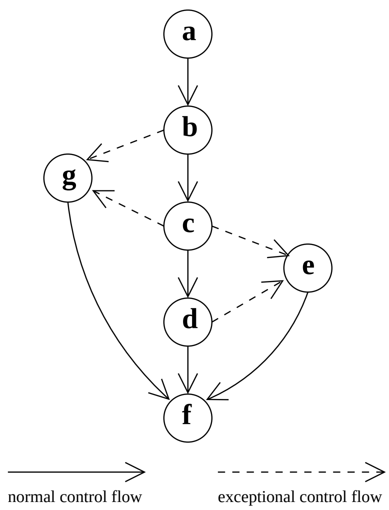
For example, Wingdis' output in 6(e) looks close to a correct solution. However, besides omitting statement `f`, the chief problem is that statement `d` has been placed in two areas of protection, which violates the semantics of the original control flow graph. The output program does operate correctly, but only because the illegal `RuntimeException` exception handler is masked off by the correct `Exception` exception handler. Since this masking only occurs because `RuntimeException` happens to be a subclass of `Exception`, it is not likely part of a correct general approach.

Object locking with `synchronized()` statements poses even greater problems. Java bytecode provides locking with `monitorenter` and `monitorexit` instructions. The Java virtual machine specification only states that for any control flow path within a method, the number of `monitorexit`s performed on some object should equal the number of `monitorenter`s. The precise conditions for representing the locked object's "critical section" with `synchronized()` statements may not exist within the target program, or equally likely, multiple "critical sections" may intersect without either nesting the other.

These problems cannot be represented with `synchronized()` statements. Luckily, it *is* possible to build an implementation of monitors in pure Java and to replace the monitor instructions with static method calls to this implementation.

As well as providing a solution for "unrepresentable" situations, this fallback mechanism gives the decompiler writer a choice about how aggressively to try to build `synchronized()` statements. At the most aggressive extreme, one might try to transform the control flow graph so as to maximize the representation of object locking with `synchronized()` statements, using the fallback mechanism only where provable necessary. At the other extreme, one might always use fallback mechanism.

We began in `Dava` by trying to make the most aggressive `synchronized()` statement restructurer possible. Through testing, however, we found that the most important issue for `synchronized()` restructuring is good exception handling. Since the set of features necessary in the bytecode to produce `synchronized()`



(a) Original control flow graph

```
public void foo()
{
    System.out.println("a");
    label_0:
    {   try
        {   System.out.println("b");
        }
        catch (RuntimeException $r9)
        {   System.out.println("g");
            break label_0;
        }
        try
        {   System.out.println("c");
        }
        catch (RuntimeException $r9)
        {   System.out.println("g");
            break label_0;
        }
        catch (Exception $r5)
        {   System.out.println("e");
            break label_0;
        }
        try
        {   System.out.println("d");
        }
        catch (Exception $r5)
        {   System.out.println("e");
            break label_0;
        }
    }
    System.out.println("f");
    return;
}
```

(b) Dava

```
public void foo()
{
    System.out.println("a");
    System.out.println("b");
    try
    {   System.out.println("c");
        System.out.println("d");
    }
    // Misplaced declaration of an exception variable
    catch(D this)
    {   System.out.println("e");
    }
    System.out.println("g");
    return;
    this;
    System.out.println("f");
    return;
}
```

(c) Jad

```
public void foo()
{
    System.out.println("a");
    System.out.println("b");
    System.out.println("c");
    System.out.println("d");
    pop this
    System.out.println("e");
    System.out.println("f");
    return;
    pop this
    System.out.println("g");
}
```

(d) Jasmine

```
public void foo()
{
    System.out.println("a");
    try
    {   System.out.println("b");
        try
        {   System.out.println("c");
            System.out.println("d");
        }
        catch (Exception e0)
        {   System.out.println("e");
        }
    }
    catch (RuntimeException e0)
    {   System.out.println("g");
    }
}
```

(e) Wingdis

```
    public void foo()
    {   System.out.println( "a" );
label_9:
        {   try
            {   System.out.println( "b" );
                try
                {   System.out.println( "c" );
                    break label_9;
                }
                catch( Exception exception1 )
                {   System.out.println( "e" );
                }
            }
            catch( RuntimeException runtimeexception1 )
            {   System.out.println( "g" );
            }
            System.out.println( "f" );
            return;
        }
        System.out.println( "d" );
    }
```

(f) SourceAgain

Fig. 6. Decompiled code for method `foo()`

blocks is both complex and specific, it turns out that the occurrence of the proper feature set is almost always the result of a `synchronized()` block in the bytecode's source. As such, it is already in a form that is easily restructured and an aggressive approach provides little improvement over simple pattern matching.

6 Related Work and Conclusions

To our knowledge there are few papers on the complete problem of decompiling arbitrary bytecode to Java. There are many tools including the decompilers we tested in this paper, however there is very little written about the design and implementation of those tools.

The implementation of the `Krakatoa` decompiler has been described in the research literature[11], however, we were unable to test this decompiler because it is not publically available. Krakatoa uses an extended version of Ramshaw's goto-elimination technique [12], which produces legal, though somewhat convoluted, Java structures by introducing loops and multi-level breaks. Krakatoa then applies a series of rewrite rules to this structured representation where each rule attempts to replace a program substructure with a more "natural" one. Such a relatively strong restructurer may be able to handle complicated loops. While it is not clear from the paper how the typing and expression building works, Krakatoa appears to use the same approach as the decompilers we tested. All program examples come from bytecode generated from `javac`. This approach does not address the problems with exceptions and synchronization.

There has been related work on restructuring Java and other high-level languages. Research on restructuring can usually be divided into restructuring with gotos, versus eliminating gotos. The independent works of Baker[2] and Cifuentes[3] are prominent examples of the first category while Erosa[4] and Z. Ammarguellat[1] are good examples of the second. These are general approaches and would require modifications to deal with the special requirements of Java, such as dealing with synchronization and exceptions.

Knoblock and Rehof[8]. have worked on finding static types for Java programs. Their approach differs from ours in that it works on an SSA intermediate representation and may change the type hierarchy when types conflict due to interfaces.

This paper has presented some of the problems, traps and pitfalls encountered when decompiling arbitrary, verifiable Java bytecode. We demonstrated the problems in dealing with variables, literals and types, and showed how existing decompilers deal with the typing problem by inserting spurious type casts (or by producing incorrect code). We showed that bytecode that has been optimized is not correctly decompiled by any of the four decompilers we tested. This demonstrates that such decompilers target bytecode that has been produced by a known compilation strategy, such as that used by `javac`. We discussed the overall problem of control flow structuring and showed that even control flow produced by `javac` can be difficult to handle. Finally, we demonstrated bytecode allows for more general use of exceptions and synchronizations than what is produced from Java. In all cases our Dava compiler was able to produce a correct Java program.

Now that we have a robust decompiler, we will begin to concentrate on a postprocessor that converts control flow constructs into idioms likely to be used by a programmer, and on mechanisms for choosing readable variable names for parameters and local variables. We will also continue to stress test the decompiler by decompiling class files from a variety of sources. The decompiler will be released as part of the Soot framework, and will be publically available. Currently, interested parties can contact the first author for a "preview version" of the software.

References

1. Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–250, March 1992.
2. B. S. Baker. An algorithm for structuring flowgraphs. *Journal of the Association for Computing Machinery*, pages 98–120, January 1977.
3. C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, July 1994.
4. A. M. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240, May 1994.
5. E. M. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium 2000*, Lecture Notes in Computer Science, pages 199–219, Santa Barbara, June 2000.
6. Jad - the fast JAva Decompiler. `http://www.geocities.com/SiliconValley/-Bridge/8617/jad.html`.
7. SourceTec Java Decompiler. `http://www.srctec.com/decompiler/`.
8. T. Knoblock and J. Rehof. Type elaboration and subtype completion for java bytecode. In *Proceedings 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*, 2000.
9. J. Miecznikowski and L. Hendren. Decompiling Java using staged encapsulation. In *Proceedings of the Working Conference on Reverse Engineering*, pages 368–374, October 2001.
10. Mocha, the Java Decompiler. `http://www.brouhaha.com/~eric/computers/-mocha.html`.
11. T. A. Proebsting and S. A. Watterson. Krakatoa: Decompilation in Java (Does bytecode reveal source?). In *3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS'97)*, pages 185–197, June 1997.
12. L. Ramshaw. Eliminating go to's while preserving program structure. *Journal of the Association for Computing Machinery*, 35(4):893–920, October 1988.
13. Soot - a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.
14. Source Again - A Java Decompiler. `http://www.ahpah.com/`.
15. R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In D. A. Watt, editor, *Compiler Construction, 9th International Conference*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, Berlin, Germany, March 2000. Springer.
16. WingDis - A Java Decompiler. `http://www.wingsoft.com/wingdis.html`.