Programmer-friendly Decompiled Java

Sable Technical Report No. 2006-2

Nomair A. Naeem Laurie Hendren
{nnaeem, hendren}@cs.mcgill.ca

March 13, 2006

Contents

List of Figures

Abstract

Java decompilers convert Java class files to Java source. Java class files may be created by a number of different tools including standard Java compilers, compilers for other languages such as AspectJ, or other tools such as optimizers or obfuscators. There are two kinds of Java decompilers, **javac-specific decompilers** that assume that the class file was created by a standard javac compiler and **tool-independent decompilers** that can decompile arbitrary class files, independent of the tool that created the class files. Typically **javac-specific decompilers** produce more readable code, but they fail to decompile many class files produced by other tools.

This paper tackles the problem of how to make a **tool-independent** decompiler, Dava, produce Java source code that is programmer-friendly. In past work it has been shown that Dava can decompile arbitrary class files, but often the output, although correct, is very different from what a programmer would write and is hard to understand. Furthermore, tools like obfuscators intentionally confuse the class files and this also leads to confusing decompiled source files.

Given that Dava already produces correct Java abstract syntax trees (ASTs) for arbitrary class files, we provide a new back-end for Dava. The back-end rewrites the ASTs to semantically equivalent ASTs that correspond to code that is easier for programmers to understand. Our new back-end includes a new AST traversal framework, a set of simple pattern-based transformations, a structure-based data flow analysis framework and a collection of more advanced AST transformations that use flow analysis information. We include several illustrative examples including the use of advanced transformations to clean up obfuscated code.

1 Introduction

Java compilers, such as the standard javac compiler, produce Java class files and these are the binary form of the program which can be distributed or made available via the Internet for execution by Java Virtual Machines (JVMs). Although the javac compiler is the most usual way of producing class files, there are an increasing number of other tools that also produce Java class files, including: compilers for other languages including AspectJ [1, 3, 4, 10] and C [2] that produce class files; bytecode optimizers which produce faster and/or smaller class files; and obfuscators which seek to produce class files that are hard to decompile and understand.

Since Java class files contain Java bytecode, which is fairly high-level intermediate representation, there has been considerable interest and success in developing decompilers which convert class files back to Java source. Such decompilers are useful for programmers to understand code for which they don't have Java source code and to help understand the effect of tools such as optimizers, aspect weavers and obfuscators.

1.1 Javac-specific Decompilers

The original decompilers, such as Mocha [14], Jad [8], Jasmin [9], Wingdis [19] and SourceAgain [17], are *javac-specific decompilers* in that they work by reversing the specific compilation patterns used by the standard javac compiler. When given class files produced by a javac compiler they can produce very readable source files that correspond closely to the original program. For example, consider the original Java program in Figure 1(a). When this program is compiled using javac from jdk1.4 to produce a class file and then decompiled with SourceAgain and Jad, one gets the very respectable results in Figure 1 (b) and (c).

These javac-specific decompilers work by assuming that the bytecode was produced with a specific javac compiler and then they look for code generation patterns which are then reversed to form the source code. Sometimes these patterns are very specific. For example, compare the results for Jad between the case when the original program was compiled with jdk1.4 (Figure 1(c)) and with jdk1.3 (Figure 1(d)). Clearly the Jad decompiler was implemented to understand the code generation patterns from javac from jdk1.3 and it does not produce as nice an output when used on class files produced using javac from jdk1.4.

1.2 Tool-independent Decompilers

Dava [12, 13] is a *tool-independent decompiler* built using the Soot [16, 18] Java optimizing framework. Dava makes no assumptions regarding the source of the Java bytecode and is therefore able to decompile arbitrary verifiable bytecode. However, this generality comes with a price. Since the Dava decompiler relies on complex analyses to find

(a) Original Code

```
while(done && alsoDone){
  if((a<3 && b==1) || b+a<1 )
     System.out.println(b-a);
}
```

(b) SourceAgain (jdk1.4)

```
while( bool && bool1 ){
  if( (i >= 3 || j != 1) && j + i >= 1 )
      continue;
  System.out.println(j-i);
}
```

(c) Jad (jdk1.4)

```
do{
  if(!flag || !flag1)
     break;
  if(i < 3 && j == 1 || j + i < 1)
     System.out.println(j-i);
} while(true);
```

(d) Jad (jdk1.3)

```
while(flag && flag1){
   if(i < 3 && j == 1 || j + i < 1)
      System.out.println(j - i);
}
```

(e) Dava (jdk1.4)

```
label_2:{
  label_1:
  while(z0 != false){
    if z1 == false){
        break label_2;
   }
    else{
      label_0:{
        if(i0 < 3){
            if(i1 == 1){
              break label_0;
            }
         }
         if(i1 + i0 >= 1){
           continue label_1;
         }
       } //end label_0:
       System.out.println(r1);
     }
   }
 } //end label_2:
```

Figure 1: Comparing decompiler outputs

control-flow structure in arbitrary bytecode, the decompiled code is often not programmer-friendly. For example, in Figure 1(e), the output from Dava is correct, but not very intuitive for a programmer. One of the goals of this paper is to provide tools that can convert the correct, but unintuitive, output of Dava to a more programmer-friendly output.

(a) Original Code

```
class test {
  private Vector buffer = new Vector();
  int getStringPos(String string) {
    for(int i=0;i<buffer.size();i++){
      String curString =
         (String)buffer.elementAt(i);
      if (curString.equals(string)) {
        buffer.remove(i);
        return i;
      }
    }
  return -1;  }  }
```

(b) Jad

```
  <snip>
     if(flag)/* Loop isnt completed */
        continue;
     s1.equals(s);
     if(flag) goto _L4; else goto _L3
_L3: JVM INSTR ifeq 59;
     goto _L5 _L6
_L5: break  MISSING_BLOCK_LABEL_48;
_L6: break MISSING_BLOCK_LABEL_59;
  <snip>
```

(c) SourceAgain

```
 <snip>
do{
   String str = null;
   if( i >= a.size() ){
  //the following goto could
   //not be resolved
     goto 81
   }
   <snip>
}while( !bool );
 <snip>
```

(d) Dava

```
class a{
  private java.util.Vector a;
  public static boolean b;
  public static boolean c;

  int a(java.lang.String r1){
    boolean z0, $z2, z3;
    int i0, $i2, i3;
    java.lang.String r2;

    z0 = c;
    i0 = 0;
    label_1:{
      label_0:
      while (i0 < a.size()){
        r2 = (String) a.elementAt(i0);
        if ( ! (z0)){
           z3 = r2.equals(r1);
           i3 = (int) z3;
           $i2 = i3;
           if (z0) break label_1;
           if (i3 == 0)
              i0++;
           else{
              a.remove(i0);
              return i0;
           }
        }
        if (z0){
           if ( ! (b))
              $z2 = true;
           else
              $z2 = false;
           b = $z2;
           break label_0;
        }
      }
      $i2 = -1;
    } //end label_1:
  return $i2; }   }
```

Figure 2: Decompiling Obfuscated Code

The challenge of providing programmer-friendly output for bytecode produced by non-javac tools is even more complex. For example, consider the example in Figure 2. In this example we compiled the Java program given in Figure 2(a) with javac and then applied the Zelix KlassMaster obfuscator [11] to the generated class file. Figures 2(b) and (c) show the results of decompiling the obfuscated class file with Jad and SourceAgain (only key snippets of the code are shown). In both cases the decompilers failed to produce valid Java code. However, as shown in Figure 2(d), Dava does create a valid Java program, which exposes the extra code introduced by the obfuscator. Even though correct, clearly this code is not very programmer-friendly and thus another big challenge addressed in this paper is how we can convert the obfuscated code into something that is more readable.

1.3 Contributions

As we have shown, the previously existing Dava decompiler produces correct, but potentially complicated Java code. The purpose of this paper is to use the existing Dava decompiler as a front-end which delivers correct, but overly complex abstract syntax trees (ASTs), and to develop a completely new back-end which converts those ASTs into semantically equivalent, but more programmer-friendly ASTs. The new ASTs are then used to generate readable Java source code. In order to build this new back-end we have developed several new components.

- Since our new back-end works by rewriting the AST we developed a visitor-based AST traversal framework, as outlined in Section 2.
- Using the visitor-based framework we then developed a large number of simple structural patterns that could be used to perform structural rewrites of the AST. These mostly correspond to common programming idioms and representative examples are given in Section 3.
- Simple structural patterns can be used for many basic tasks, but in order to do many more complicated rewrites we needed to have data flow information. Thus, we have developed a structural data flow analysis framework, as outlined in Section 4.
- Given the flow analysis information computed using the framework we have developed several more advanced patterns. In Section 5 we discuss our advanced pattern for reconstructing `for` loops, and we show how analysis information can be used to remove useless code from obfuscated bytecode.

We have integrated all these techniques and tools into Dava and as we demonstrate with the examples in the rest of the paper, we can apply these to produce more programmer-friendly code.

2 Visitor-based AST Traversal Framework

A first step to implementing analyses/transformations on a tree structure is to have a good traversal mechanism. Analyses to be performed on Dava's AST require a traversal routine that provides hooks into the traversal allowing modification to the AST structure or the traversal routine.

Inspired by the traversal mechanism provided by SableCC [6], tree walker classes were created using an extended version of the visitor design pattern. The Visitor-based traversal allows for the implementation of actions at any node of the AST separately from AST creation. This allows for modular implementation of distinct concerns and a mechanism which is easily adaptable to needs of different analyses.

3 Simple Structural Patterns

Dava's initial implementation focused on correct detection of Java constructs and did not address the complexity of the output. To be useful as a program understanding tool it is important that Dava competes with other decompilers not only in the range of applicability but also the quality of output.

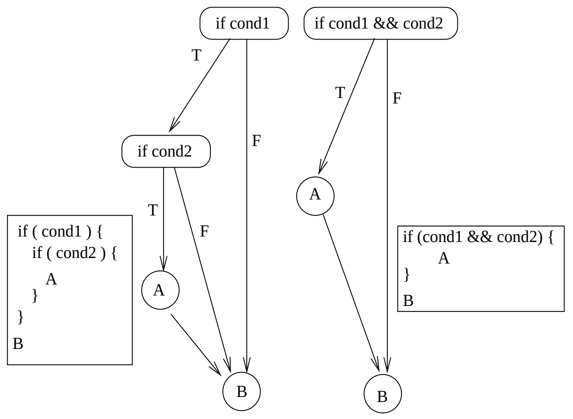
The cryptic control flow in the decompiled output is complex largely due to the fact that Java bytecode only allows binary comparison operations for deciding control flow. However, this restriction does not exist in Java where boolean expressions can be aggregated using the $\&\&$ and $||$ operators. Dava does not make use of this ability and hence converts each comparison operation into a separate conditional construct. This results in the creation of unnecessary Java constructs and their complicated nesting further increases code complexity. For instance, an `If` statement evaluating two conditions using the $\&\&$ operator in the source code gets decompiled into two `If` statements one completely nested within the other. Similarly if a loop checks for multiple conditions in the source this gets transformed into a loop with one condition. The remaining conditions are checked within the loop body. By statically checking for such patterns, and merging the different conditions, the number of Java constructs can be reduced thereby reducing the complexity of the output.

Abrupt control flow in the form of labeled blocks and `break`/`continue` statements, created by Dava to handle any `goto` statements not converted to Java constructs, also complicate the output. Programmers rarely use such constructs, since it makes understanding code harder, and it is therefore desirable to minimize their use.

AST rewriting in Dava's back-end is done using multiple traversals. As long as the AST is modified, because of a matched pattern, the traversals are repeated until no further patterns apply. This is necessary since application of one transformation might enable subsequent transformations. In Sections 3.1- 3.9 we discuss some of the important patterns that we identified.

3.1 And Aggregation

`And` aggregation is used to aggregate two `If` statements into one using the && symbol. Figure 3(a) shows the control flow of two `If` conditions one fully nested in the other. From the control flow graph it can be seen that A is executed only if both `cond1` and `cond2` evaluate to true. B is executed no matter what. In Figure 3(b) we see the reduced form of this graph where the two `If` statements have been merged into one by coalescing the conditions using the && operator. Statements 9 to 13 in Figure 1(e) match this pattern. The matched pattern and the transformed code is shown in Figure 4.



```
if ( cond1 ) {
   if ( cond2 ) {
      A
   }
}
B
```

```
if (cond1 && cond2) {
      A
}
B
```

(a) Unreduced (b) Reduced

Figure 3: Reducing using the && operator.

(a) Original Code

```
9    if(i0 < 3){
10    if(i1 == 1){
11     break label_0;
12    }
13   }
```

(b) Transformed Code

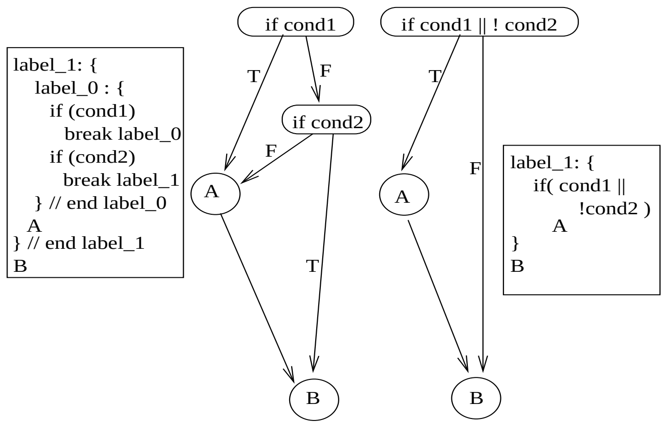
```
if(i0 <3 && i1 == 1){
   break label_0;
}
```

Figure 4: Application of `And` Aggregation

3.2 Or Aggregation

Figure 5 shows the control flow of the `Or` Operator. The unreduced version of the control flow shows that A is executed if `cond1` evaluates to true. If, however, the false branch is taken then `cond2` is evaluated and A is executed if this condition is false. B is executed no matter what. In short, A is executed if the first condition is true or the negated second condition is true, followed by the execution of B in all cases. This graph can therefore be reduced to that in Figure 5(b) where the `If` statement aggregates the two conditions using the || operator.

One of the patterns to which the control flow graph in Figure 5(a) can map is shown in Figure 5. The pattern looks for a sequence of n `If` statements (n is 2 in Figure 5) with the first n-1 statements breaking a particular label (label0 in Figure 5) and the nth statement targeting an outer label (label1 in Figure 5). During execution this results in the evaluation of a sequence of `If` conditions and as soon as any of the n-1 conditions evaluates to true or the nth condition evaluates to false a certain chunk of code (A in the figure) is targeted. If the program gets to the nth condition and this evaluates to true then in this case A is not executed. This code therefore corresponds to an `If` statement with A as its body and the condition the aggregated result of ORing the n-1 conditions and the negated nth condition.



```
label_1: {
   label_0 : {
      if (cond1)
         break label_0
      if (cond2)
         break label_1
   } // end label_0
   A
} // end label_1
B
```

```
label_1: {
   if( cond1 ||
         !cond2 )
      A
}
B
```

(a) Unreduced (b) Reduced

Figure 5: Reducing using the || operator

The decompiled code in Figure 1(e) has one occurrence of this pattern. Statement 2 is the outer label and Statement 8 the inner one. There are two `If` statements in the sequence: statement 9 breaking the inner label and statement 14 targeting the outer one. The transformation removes the second `If` statement by moving its negated condition into the first statement. The new body of this statement consists of statement 18. Assuming that And Aggregation has already occurred the end result after Or Aggregation is shown in Figure 6.

```
label_2:{
  label_1:
  while(z0 != false){
    if (z1 == false){
      break label_2;
    }
    else{
      if( (i0 < 3 && i1 == 1)
                  || i1 + i0 < 1 ){
        System.out.println(r1);
      }
    }
  }
} //end label_2:
```

Figure 6: Application of Or Aggregation

An interesting side-effect of the transformation is the removal of labeled blocks and `break` statements. The first n-1 statements all break label0 whereas the nth statement targets label1. After the transformation all n-1 `break` statements have been removed which also allows the removal of label0. Also, although we cannot directly remove label1, without checking that the `If` body does not target it, we have reduced the number of abrupt edges targeting it by one. The next subsection discusses an algorithm that checks for spurious labels and subsequently removes them.

3.3 Useless Label Remover

The `Or` and `And` aggregation patterns provide new avenues for the reduction of labeled blocks and abrupt edges. With the help of pattern detection and use of DeMorgans Theorem the number of abrupt edges and labels can be reduced considerably.

Labels can occur in Java code in two forms: as labels on Java constructs e.g. `While` loop or as labeled blocks. If a label is shown to be spurious, by showing that there is no abrupt edge targeting it, then in the case of a labeled construct the label is simply omitted. However, in the case of a labeled block, a transformation is required which removes the labeled block from the AST. Algorithm 1 shows how a spurious labeled block is removed by replacing it with its body in the parent node. Using this pattern label1 in Figure 6 can be removed since no abrupt edge targets it.

Algorithm 1: Removing Spurious Labeled Blocks

```
Input: ASTNode node
body ← GetBody(node)
while body has more ASTNodes do
    node1 ← GetNextNode(body)
    if node1 is a Labeled Block Node then
        if IsUselessLabelBlock (node1) then
            body1 ← GetBody(node1)
            Replace node1 in body by body1
        end
    end
end
```

3.4 Loop strengthening

Similar to `If` and `If-Else` statements, loops can also hold aggregated conditions to be evaluated before execution of the loop body. Therefore pattern matching can be used to strengthen the conditions within a loop. One such pattern, for a `While` loop is shown on the left of Figure 7(a).

Reasoning about the control flow shows that Body A is executed if both `cond1` and `cond2` evaluate to true. If either of the conditions are false the loop exits. This fits in with the notion of a conditional loop with two conditions as seen in the reduced form of the code in Figure 7(a). Notice that the label on the `While` loop is still present in the reduced code. This is because there can be an abrupt edge in Body A targeting this label. After the reduction the algorithm in Section 3.3 is invoked to remove the label from the loop, if possible.

Figure 7(b) shows a similar strengthening pattern for unconditional loops. The only difference is that in this case the `If-Else` statement is free to have any construct in both branches as long as one of the branches has an abrupt edge targeting the labeled loop. The reduction works by converting the `Unconditional-While` loop to a conditional loop with Body A as the body of the loop. Body B is then moved outside the loop. The specialized pattern where Body B is empty makes this pattern the same as the pattern for `While` loops.

Looking at our working example (Figure 6) where `And` and `Or` aggregation have already been applied we can see that statements 3 to 12 make a `While` loop which has one `If-Else` statement. Notice that in this case the `If-Else` statement is reversed: the `If` branch contains the break out of the loop and the `else` branch contains Body A (statements 8 and 9). In this case we can apply the `While` strengthening pattern by adding the negated condition of the `If-Else` statement into the `While` condition. The transformed code is shown in Figure 8. Notice that label2

(a) Strengthening conditional loops

```
   (Unreduced)

label_0:
while(cond1){
  if(cond2){
    Body A
  }
  else{
    break label_0
  }
}//end while
```

```
  (Reduced)

label_0:
while(cond1 && cond2){
   Body A
}
```

(b) Strengthening Unconditional loops

```
  (Unreduced)

label_0:
while(true){
  if(cond1){
    Body A
  }
  else{
    Body B
    break label_0
  }
}//end while
```

```
  (Reduced)

label_0:
while(cond1){
  Body A
}
Body B
```

Figure 7: Strengthening Loops

and label1 which were at statement 1 and 2 in Figure 6 have been removed by the `UseLessLabelRemover` of Section 3.3.

There are a number of other patterns which can be used to strengthen conditions in a while loop. One pattern worth mentioning is when a `While` body contains only one `If` statement. This transformation result in empty while bodies with the work being done from within the conditions of the loop. Such kind of loops are often encountered in concurrent programs e.g. busy waiting.

```
while(z0 != false && z1 != false){
   if( (i0 < 3 && i1 == 1)
            || i1 + i0 < 1 ){
      System.out.println(r1);
   }
}
```

Figure 8: Application of While Strengthening

3.5 Condition Simplification

Expressions evaluating to boolean types are often used as unary conditions. The original Dava, however, represented these as binary operations, comparing the expressions to the boolean constants `false` or `true`.

Figure 9 shows the different conversions that can be carried out. Since most programmers are used to reading boolean expressions in the form of unary conditions the effect of these transformation is that code becomes easier to read

Applying this pattern on our working example of Figure 8 results in the simplification of the two boolean conditions

```
    A != false --->  A
    A != true  ---> !A
    A == false ---> !A
    A == true  --->  A
```

Figure 9: Converting Binary Conditions to Unary Conditions

in Statement 1. The resulting code is given in Figure 10. Looking back at the original source code from which this decompiled output was generated (Figure 1(a)) we see that, after applying the AST rewriting, Dava's output matches the original source code.

```
while(z0  && z1 ){
   if( (i0 < 3 && i1 == 1)
            || i1 + i0 < 1 ){
      System.out.println(r1);
   }
}
```

Figure 10: Application of Boolean Simplification

3.6 Reducing the scope of labeled blocks

In an attempt to remove a labeled block some pattern might not get matched because the labeled block contains too many children in its body. It is sometimes possible to reduce the scope of the labeled block by reducing the number of children of a labeled block. An example would be a labeled block which consists of some code that does not target the label followed by code which does target it. Since the initial code does not involve the use of the label there is no reason why this code cannot occur outside the scope of the labeled block. Moving this code outside (above) the label makes the labeled block tighter in the sense that it has fewer children in its body. The reasoning behind this transformation is that if there are fewer children in a labeled block, then there are better chances that some other pattern will match. If no pattern matches, reducing the labeled block size still has the advantage of improving code complexity since the programmer now has to concentrate on a smaller chunk of code to figure out the abrupt control flow targeting the labeled block.

3.7 Shortcut increments and decrements

Another simple transformation for ease of reading code is the use of shortcut increment and decrement statements. It is common practice to represent the increment statement i = i + 1 using the increment operator ++ and using a similar decrement operator for the i = i - 1 statement. This transformation replaces occurrences of i = i + 1 with i++ and i=i-1 with i–.

3.8 De-Inlining Static Final Fields

Standard Java compilers inline the use of static final fields. The reasoning is that since the field is final the value is not going to change and hence the constant value can be used in the bytecode instead of having to look up the value from a class attribute. The decompiled output therefore contains the constant values wherever there was a static final field in the original code. We think it is a good idea to try to recover the use of the field that was used in the original code since the name of the field might be able to deliver some contextual information to the programmer. A simple transformation was written which keeps a pool of all static final fields and their corresponding values found in a particular class. A simple depth first traversal is then carried out that checks for the occurrence of constant values in the code. When a constant value is encountered it is checked with the list of known values for the different static final fields. If there is a match then the use of the constant value is replaced by the use of the static final field. This kind of transformation allows for more use of identifiers in the code and allows the programmer to gather contextual information while reading the code.

3.9 Variable Declarations and Initialization

Dava was previously unable to convert multiple variable declarations into a single declaration statement. Also previously a declaration and the subsequent initialization of the variable was always broken into two consecutive statements. A simple transformation now allows for the aggregation of variables with the same type into one declaration statement. Also a variable which is initialized as soon as it is declared can now be initialized within the declaration statement. This is a common programming idiom and makes the code more natural.

(a) Unreduced

```
int a;
int b=3;
int c;
```

(b) Reduced

```
int a, b=3,c;
```

Figure 11: Variable Declarations and Initialization

4 Structure-based Flow Analysis

Although AST rewriting based on pattern matching greatly reduces the complexity of the decompiled output, this alone allows only for a limited scope of transformations. Sophisticated transformations need additional information which is available only through the use of static data flow analyses.

An example of this can be seen in Dava's output, Figure 2(d), for the obfuscated bytecode produced for the original Java source shown in Figure 2(a). Although semantically equivalent to the original code the output is hard to understand. However, since obfuscators have to ensure that their modifications do not change program semantics, a simplification of the output, making it similar to the original code, should be possible. This requires added information about the data and control flow to answer questions like: "What is the value of a particular variable at a program point?", "Is a particular piece of code ever executed?" and so on. This information cannot be obtained from pattern matching and we need data flow analysis for it. We discuss more about decompiling obfuscated code in Section 5.2.

To perform more sophisticated transformations an analysis framework was implemented that allows for simple implementation of static data flow analyses. The analyses' results are then leveraged to perform further transformation on the AST. The framework removes the burden of correctly traversing the AST from the analysis writer and allows him/her to concentrate on the analysis. With a framework in hand the process of writing analyses for Dava has been streamlined making it easier for new developers to extend the system.

Since the analyses for the decompiler are performed on the AST it is best to use a syntax-directed method of data flow analysis such as structural analysis [7, 15]. The advantage of using this technique is that it gives, for each type of high level control-flow construct in the language, a set of formulas that perform data flow analysis. For instance it allows the analysis of a `While` loop by analyzing only its components: the conditional expression and the body. Apart from supporting ordinary compositional constructs such as conditionals and loops, the structural flow analysis also supports `break` and `continue` statements (Section 4.1). We find that structural flow analysis provides a more efficient and intuitive implementation of analysis on the tree representation than iteration.

4.1 Flow Analysis Framework

The Structural Flow analysis framework for Dava's AST has been written by providing an abstract `StructuredAnalysis` Java class. Programmers wanting to implement an analysis need only implement the abstract methods in this class which deal with the initialization of the analysis and then subsequently dealing with the type of information to be stored by different constructs.

The analysis begins by traversing the AST. As each Java construct is encountered a specialized method responsible for processing this construct is invoked. An `input` set containing information gathered so far is sent as an argument. Each construct is handled differently depending on the components it contains and its semantics. The processing of the construct might add, remove or modify the `input` set. The result is returned in the form of an `output` set which then becomes the `input` set for the next construct. Figure 12 shows how the framework handles a sequence of statements. The processing method iterates through the statements in the sequence with the `output` set of one statement becoming the `input` of the next statement. The `output` set of the last statement is the `output` set of the sequence of statements. This kind of structure based flow analysis is not new. Similar work has been done by Emami et. al. [5, 7] for gathering alias and points-to-analysis information for the McCat C compiler. Dava's flow analysis framework is an implementation of the same approach utilized in McCat but implemented for Java.

```
process_StatementsNode(
  StatementSequenceNode node,Object input){
  List stmts = node.getStatements()
  out = clone(input)
  for each stmt, s in stmts
     out = process(s,out)
  return out
}
```

Figure 12: Analyzing a statement sequence

An important construct in flow analyses is the merge operation. Merge defines the semantics of combining the information present in two `flow-sets`. Such a situation arises for instance when dealing with the `flow-sets` obtained by processing the `If` and `else` branch of an `If-Else` construct. Since the framework gathers sets of information the programmer has the choice of choosing between union and intersection as the merge operation.

Before discussing how the framework handles complicated constructs like conditionals and loops lets look at how abrupt control flow statements are handled. Without going into the details of `break` and `continue` we know that when such a statement is encountered control passes to the target of the abrupt statement. In the case of `break` this is usually a loop, a switch or a labeled block whereas in the case of `continue` the target is always a loop. In our framework whenever a `break` or `continue` is encountered the targeted construct and the `flow-set` are stored into a hash table. Processing then continues with a special `flow-set` named `BOTTOM` sent onwards indicating that this path is never realized (as the abrupt statement leads execution to some other area of the code).

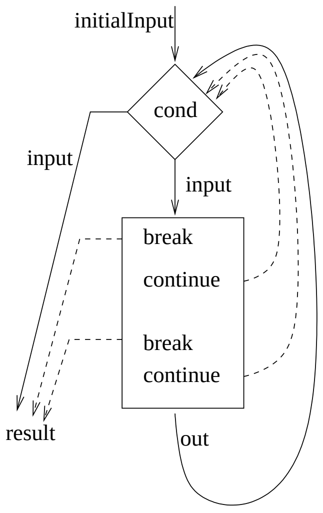
We use a hash table to store flow-sets so that when the target of an abrupt statement is processed the stored `flow-sets` that target this construct are retrieved and merged with the `flow-set` obtained through analysis of the construct.

Figure 13 shows the control flow and pseudo-code for handling a `While` loop. The solid back-edge indicates loop iteration and dotted lines indicate abrupt control flow. Since we are dealing with a loop, a fixed point computation is necessary to compute the final `output` set. Firstly the analysis processes the condition of the `While` construct. The `output` set of this becomes the `input` set for the fixed point computation. Within the fixed point computation the body of the `While` loop is processed followed by the generation of the `input` set for the next iteration. This is done by merging the `output` set of the current iteration with the `flow-sets` stored in the `continue` hash table, since `continue` statements could be targeting the loop. This is followed by a merge with the initial input to the `While` loop, hence taking care of all possible entry points of the loop. Once the fixed point is achieved then any `flow-sets` stored in the `break` hash table are also merged using the `handleBreaks` method. The output of this method is the final output of processing the `While` construct.

4.2 Implemented Flow Analyses

A number of typical compiler flow analyses have been implemented using the structure-based flow analysis framework. Some of them are briefly discussed below along with their usage:

Reaching Defs: This analysis computes information regarding which definition of a variable may reach a particular



```
process_While(WhileNode node,Object input){
  initialInput = clone(input)
  input = processCondition(condition,
                           initialInput)
  do{
    lastin = clone(input)
    out = processBody(node,input)
    out = handleContinue(out,node)

    //merge cond evaluating to false
    input = merge(initialInput,out)
    input = processCondition(condition,input)
  } while(lastin != input)
  result = handleBreaks(input,node)
  return result
}
```

Figure 13: Analyzing the `While` construct.

program point. The results of this analysis are used to compute uD-dU chains which are all possible definitions for a particular use of a variable and conversely all possible uses for a particular definition. This information is crucial in deciding which variables and definitions are needed for a particular chunk of code. We touch on this again in Section 5.1.

Constant Propagation: This analysis stores information about values a variable must have at a program point. Although statically a lot cannot be said about the runtime value of a variable, the results of this analysis have surprisingly good results in simplifying obfuscated code (Section 5.2).

Reaching Copies: A copy statement is defined as a statement of the form `a`=b; *i.e.*, a statement where the value of one variable is being copied into another. Reaching copies gathers information about copies that reach a particular program point. This information in conjunction with the uD-dU chains obtained from the reaching defs flow analysis can be used to implement the copy elimination transformation. An example of this is shown in Figure 14. The unreduced form of the code shows a copy statement `x`=a; which gets eliminated in the reduced version due to copy elimination.

(a) Unreduced

```
  x = a; //copy stmt
  if(b == 3)
    foo(x);
```

(b) Reduced

```
if (b == 3)
   foo(a);
```

Figure 14: Copy Elimination

Must Assign: A local or field is `MUST` initialized at a program point p if on all paths from the start to this point the local or field occurs on the left side of an assignment statement.

The analysis is a forward analysis with intersection as the merge operation (there needs to be an assignment on both paths for the `MUST` condition to be satisfied). Information stored by the analysis at different points of the program are the set of locals or fields that are `MUST` initialized so far. A variable is added to this set if there is an assignment to the variable. There are no specific constructs which kill a particular variable. Variables are therefore removed only by the intersection operation applied at merge points. The out(start) and in(s_i) are empty sets indicating no variable has been `MUST` initialized so far.

May Assign: The `MAY` assign analysis works similarly to the `MUST` analysis and differs only in the use of union as the merge operation. Hence this analysis gathers the local or fields that have at least one assignment on at least one path in the code. The analysis adds variables to flow sets similar to the `MUST` analysis. However, once a variable is added it is never removed from the set indicating the fact that a variable `MAY` be assigned on at least some path of the program. An example of the use of `MUST` and `MAY` analyses is discussed in Section **??**.

5 Complex Patterns using Flow Analyses

With the structure-based flow analysis framework in hand we now have the resources to gather any additional information required for more complex transformations. Simple analyses like reaching defs, constant propagation *etc.* can provide enough information to considerably improve the code. In the next two sections we discuss transformations which would not have been possible without the flow analysis framework.

5.1 For Loop Construction

Certain conditional `While` loops can be represented more compactly as `For` loops. Programmers generally prefer to use `For` loops specially when the loop has a consistent update. A `For` loop has four important constructs: The `Init` where variables to be used in the body can be declared and initialized. This is invoked once before the first iteration of the loop. Then there is the `condition` which is evaluated before each iteration of the loop. The loop only executes if the condition evaluates to true. The `update` construct is executed at the end of each iteration and performs updates on variables. The last part of the `For` loop is the `Body` which contains the loop code.

(a) Unreduced

```
Body A;
Init Stmts
while (cond) {
  Body B
  Update C
} //end while
```

(b) Reduced

```
Body A
for (Init Stmts;cond;Update C)
{
  Body B
} // end for
```

Figure 15: The `While` to `For` conversion

We define natural `For` loops as those loops where all four constructs of the `For` loop contain at least one expression/statement. The `While` to `For` transformation looks for patterns which can be converted into natural `For` loops. The pattern is shown in Figure 15(a).

The general form of the reduction is shown in Figure 15(b). However, there are a number of restrictions on the different constructs and the transformation succeeds only if all restrictions are fulfilled. The procedure and the restrictions can be best explained by going through the algorithm for the transformation.

Algorithm 2 outlines the steps taken to transform a `While` loop into a `For` loop. The body of an ASTNode is searched for a sequence of statements followed by a `While` loop. The statement sequence is the combination of Body A and Init Stmts in Figure 15(a). These statements are then analyzed to retrieve the init using the `GetInit` function.

The `GetInit` function goes through the sequence of statements and gathers all statements that are initializing any variables. Once all such statements have been gathered they are analyzed to check whether the initialized variables are only used within the `While` loop body. This information is readily available through the uD-dU chains created using the reaching defs flow analysis. If all uses of variables initialized in the `init` are present only in the `While` body then we know that the variable is live only within this body and hence the initialization is converted into a loop-local declaration and initialization statement.

The next step in the algorithm is to retrieve the update statements for the `For` loop to be created. This is achieved using the `GetUpdate` function. We know that the last statements to be executed before starting a new iteration are the update statements. Hence we look for these statements in the last node of the body of the `While` loop. The

`GetUpdate` function retrieves the last node and checks that it is a sequence of statements. If so the sequence of statements is checked to see if they update a variable which is either initialized in the `init` or is part of the condition of the `While` loop. If we can not find such a statement the transformation fails since we only want to create *natural* `For` loops. However, if we are able to identify update statements these are stripped away from the sequence of statements. This again requires the use of the uD-dU chains to check that any update being made is not utilized in the statements following the update statement. If there is a use of the update statement before the loop body ends then this statement cannot be removed from its current location in the sequence.

If an `init` and `update` list are successfully retrieved then we can create the `For` loop. The first step is to create the sequence of statements that will replace the existing sequence (the combined Body A and Init stmts node of Figure 15(a)). This is achieved by the `RemoveInitStmts` function which goes through the statements and keeps only those which do not belong to the `init`. Basically we are left with Body A which is then used to create a new statement sequence node.

Algorithm 2: The `While` to `For` conversion

```
Input: ASTNode node
body ← GetBody(node)
while body has more ASTNodes do
    node1 ← GetNextNode(body)
    node2 ← GetNextNode(body)
    if node1 is a series of statements and node2 is a conditional while loop then
        init ← GetInit(node1)
        update ← GetUpdate(init,node2)
        newStmts←removeInitStmts(node1,init)
        stmtsNode ← ASTStatementSequenceNode(newStmts)

        condition ← GetCondition(node2)
        whileBody ← GetBody(node2)
        forNode ← ASTForLoop(init,condition,update,whileBody)

        Replace node1 and node2 by stmtsNode and forNode in body
    end
end
```

The `For` loop is then created with the condition of the `While` loop as its condition and the body of the `While` loop as its body minus the update statements which becomes the update part of the `For` loop. The new statement sequence node and the `For` loop then replace the old statement sequence node and `While` loop in the AST. An example of this transformation is discussed in the next section.

5.2 Program Obfuscation

In Section 4 we mentioned that without additional information, provided by flow analyses, Dava is unable to simplify the confusing output produced by decompiling obfuscated code. Figure 2(d) shows such an output. Program transformations targeting decompiled obfuscated code and using data flow analysis were implemented to simplify the output. One such transformation uses the constant propagation analysis discussed in Section 4.2. In the case of our example constant propagation is able to prove that z0 is false at Statement 15 in Figure 2(d). This is so since z0 is only assigned once from the boolean c, Statement 9, which is always false. The consequences of this additional information are that we are able to statically predict that the `If` body is always executed since the condition in Statement 15 always evaluates to true. Hence the conditional is redundant and is removed. Similarly at Statement 27, constant propagation tells us that z0 is still false. Hence the `If` body, Statements 28 to 33, will never get executed and is effectively dead code. This is also removed from the output. With just constant propagation the output of Figure 2(d) changes to that shown in Figure 16.

Once such code has been removed from the output the simpler AST transformations (Section 3) get activated which result in further simplification of the output. For instance the `While` loop on Statement 8 in Figure 16 gets converted

```
1  class a{
2    private java.util.Vector a;

3    int a(java.lang.String r1){
4      boolean z3;
5      int i0, $i2, i3;
6      java.lang.String r2;

7      i0 = 0;
8      while (i0 < a.size()){
9         r2 = (String) a.elementAt(i0);
10        z3 = r2.equals(r1);
11        i3 = (int) z3;
12        $i2 = i3;
13        if (i3 == 0)
14           i0++;
15        else{
16           a.remove(i0);
17           return i0;
18        }
19   }
20   $i2 = -1;
21   return $i2;  }
```

Figure 16: Constant Propagation

to a `For` loop with Statement 7 as the init and Statement 14 as the update.

Another interesting and very important transformation is indicated on statement 11 in Figure 16. In this case the obfuscator was in fact able to confuse Dava by assigning a boolean to an integer variable. However, Dava now uses a flow analysis to check for such instances and removes the unnecessary assignment introduced. Also notice that declarations of variables that are no longer used are also removed by Dava. The final output from Dava for the obfuscated code is shown in Figure 17.

```
class a{
  private java.util.Vector a;

  int a(java.lang.String r1){
    boolean z3;
    java.lang.String r2;

    for(int i0 = 0;i0 < a.size(); i0++){
        r2 = (String) a.elementAt(i0);
        z3 = r2.equals(r1);
        if (z3){
           a.remove(i0);
           return i0;
        }
   }
   return -1; }
```

Figure 17: Final result of decompiling obfuscated code of Figure 2

6 Related Work

There are numerous decompilers available for Java bytecode. Two notable ones are Jad [8] and SourceAgain [17]. Jad is a javac-specific decompiler which is free for non-commercial use. Its decompilation module has been integrated into several graphical user interfaces including FrontEnd Plus, Decafe Pro, DJ Java Decompiler and Cavaj. It is relatively easy to break the decompiler by introducing non-standard, though verifiable, bytecode.

SourceAgain is a commercial decompiler with an online version available to test its capabilities. The decompiler creates a flow graph representation from which it detects Java constructs. It does a better job at decompilation than Jad but fails when given bytecode produced by non-java compilers, *e.g.*, AspectJ. Although SourceAgain claims to be able to decompile obfuscated code our tests have shown that it is only able to handle name obfuscation(by converting these to indexed names) and fails when even simple control flow obfuscation has been carried out.

Structural Flow analysis initially presented by Sharir [15] is ideal for data-flow analysis using a structured representation of the program. This technique has been successfully used in creating an optimizing compiler which uses a hierarchy of structured intermediate representations [7]. Various compiler optimizing techniques *e.g.*, inter-procedural analysis, forward or backward analysis can all be implemented on the structured representation of the program in a much more intuitive way than simple iteration.

7 Conclusions and Future Work

We have introduced the challenges involved in producing programmer-friendly Java source with a tool-independent decompiler. A tool-independent decompiler must deal with arbitrary verifiable bytecode as produced by a wide variety of tools including compilers for other languages such as AspectJ and C, bytecode optimizers and obfuscators.

The previously developed Dava decompiler dealt with the problem of producing correct Java output, but often this output was hard to understand for the programmer. In this paper we demonstrated a variety of techniques that we have used to develop a new back-end for Dava that converts the complex AST structures produced by Dava into semantically equivalent ASTs that are more programmer-friendly.

Our approach is based on AST rewriting. This rewriting is supported by a visitor-based AST framework. We first demonstrated a variety of simple structure-based patterns that handle many program idioms and demonstrated these with a variety of examples. We then described the development of a structure-based flow analysis framework that we have used for implementing a variety of flow analyses. Using the results from these analyses we presented several more complex AST rewriting rules including `for` loop structuring and the elimination of redundant computation and control flow introduced by an obfuscator.

We continue to actively develop more rewriting patterns and analyses, including those that allow us to decompile code produced by AspectJ compilers. All of the techniques presented in this paper have been implemented in the Soot framework and will appear in the next public release of Soot.

References

[1] abc. The AspectBench Compiler. Home page with downloads, FAQ, documentation, support mailing lists, and bug database. `http://aspectbench.org`.

[2] Axiomatic Multi-Platform C compiler suite. `http://www.axiomsol.com`.

[3] AspectJ Eclipse Home. The AspectJ home page. http://eclipse.org/aspectj/, 2003.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD 2005*, pages 87–98, March 2005.

[5] M. Emami. A practical interprocedural alias analysis for an optimizing/parallelizing c compiler. Master's thesis, School of Computer Science, McGill University, August 1993.

[6] E. M. Gagnon and L. J. Hendren. Sablecc, an object-oriented compiler framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 140, Washington, DC, USA, 1998. IEEE Computer Society.

[7] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420. Springer-Verlag, 1993.

[8] Jad - the fast JAva Decompiler. `http://www.geocities.com/SiliconValley/Bridge/8617/jad.html`.

[9] SourceTec Java Decompiler. `http://www.srctec.com/decompiler/`.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[11] Zelix KlassMaster - The second generation Java Obfuscator. `http://www.zelix.com/klassmaster`.

[12] J. Miecnikowski and L. J. Hendren. Decompiling Java bytecode: problems, traps and pitfalls. In R. N. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127. Springer Verlag, 2002.

[13] J. Miecznikowski and L. Hendren. Decompiling Java using staged encapsulation. In *Proceedings of the Working Conference on Reverse Engineering*, pages 368–374, October 2001.

[14] Mocha, the Java Decompiler. `http://www.brouhaha.com/~eric/computers/mocha.html`.

[15] M. Sharir. Structural analysis: A new approch to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.

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

[17] Source Again - A Java Decompiler. `http://www.ahpah.com/`.

[18] 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.

[19] WingDis - A Java Decompiler. `http://www.wingsoft.com/wingdis.html`.