Decompiling Java Using Staged Encapsulation

Jerome Miecznikowski
Laurie Hendren
Sable Research Group, School of Computer Science
McGill University, Montreal, Canada
`[jerome,hendren]@sable.mcgill.ca`

Abstract

This paper presents an approach to program structuring for use in decompiling Java bytecode to Java source. The structuring approach uses three intermediate representations: (1) a list of typed, aggregated statements with an associated exception table, (2) a control flow graph, and (3) a structure encapsulation tree.

The approach works in six distinct stages, with each stage focusing on a specific family of Java constructs, and each stage contributing more detail to the structure encapsulation tree. After completion of all stages the structure encapsulation tree contains enough information to allow a simple extraction of a structured Java program.

The approach targets general Java bytecode including bytecode that may be the result of front-ends for languages other than Java, and also bytecode that has been produced by a bytecode optimizer. Thus, the techniques have been designed to work for bytecode that may not exhibit the typical structured patterns of bytecode produced by a standard Java compiler.

The structuring techniques have been implemented as part of the `Dava` *decompiler which has been built using the Soot framework.*

1. Introduction

Java bytecode is a portable, stack-based, intermediate representation which forms the instruction set for a Java Virtual Machine[7]. Java bytecode is produced by compilers for Java such as Sun's `javac` and IBM's `jikes` compiler. However, due to the portability benefits of Java bytecode, compilers for many other languages are being developed to produce Java bytecode as their target. For example, there now exist compilers for Ada, Cobol, C/C++, Fortran, Eiffel, Oberon and Smalltalk.[1]

A decompiler that converts from Java bytecode to Java source code is a useful tool for many purposes including: program understanding, visualization of optimized code, recovery of lost Java source code, and translating from one high-level language to another.[2]

Since Java bytecode is relatively high-level and is guaranteed to be well-formed and well-typed due to verification constraints, it provides an ideal basis for decompilation back to Java source. The development of Java decompilers such as Mocha (released in 1996) [9] started very soon after the popularization of Java. The early decompilers exploited the fact that most Java bytecode was produced by Sun's `javac` compiler and had a very predictable structure. Thus, relatively simple pattern matching techniques could do a reasonably good job finding the program structure.

Since then, the job of a Java decompiler has been complicated by the wide variety of tools now producing Java bytecode, including compilers for non-Java languages, bytecode optimizers, bytecode watermarkers and bytecode ofuscators. These tools produce bytecode that is correct and verifiable, but much more complex than the bytecode produced by `javac`. The bytecode produced from such tools may: (1) leave results on the Java stack between basic blocks; (2) contain dead code and useless branches; (3) contain irreducible control flow; (4) have non-nested synchronization areas; and (5) unstructured use of exceptions. Extracting program structure is no longer a simple task, and simple pattern matching techniques will not suffice.

This paper describes the program structuring algorithm used in the `Dava` decompiler, a decompiler written in Java and built upon the `Soot` framework[12, 14, 15]. The structuring algorithm is designed to work for any verifiable Java bytecode by using three main intermediate representations, including a *structure encapsulation tree* which provides a natural way of representing partial structurings. Based on

[1] For a more complete list see `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`.

[2] A translator from language `X` to Java can be constructed by combining a compiler for language `X` to bytecode with a decompiler from bytecode to Java.

```
public boolean search( ListNode list, Object o)
{   while (list != null) {
        try {
            if (list.value.equals( o))
                return true;
            else
                list = list.next;
        }
        catch (Exception e) {
            System.out.println( "Error in List");
            break;
        }
    }
    return false;
}
```

Figure 1. Example Java method.

```
     public boolean search(ListNode, java.lang.Object)
     {
         SearchList r0;
         ListNode r1;
         java.lang.Object r2;
         java.lang.Exception r3, r5;

a        r0 := @this;
b        r1 := @parameter0;
c        r2 := @parameter1;
         goto label4;

      label0:
d        if r1.value.equals(r2) == 0 goto label1;

e        return 1;

      label1:
f        r1 = r1.next;

      label2:
         goto label4;

      label3:
g        r5 := @caughtexception;
h        r3 = r5;
i        java.lang.System.out.println("Error in List");
         goto label5;

      label4:
j        if r1 != null goto label0;

      label5:
k         return 0;

         catch java.lang.Exception
               from label0 to label2 with label3;
     }
```

Figure 2. Grimp representation for Figure 1

these representations the algorithm proceeds in six stages, with each stage detecting the structures corresponding to a specific collection of Java constructs and thus refining the structure encapsulation tree. Some of the stages are very specific to Java constructs such as synchronization, exceptions, and continues/breaks to labeled statements. After completion of the six stages the structure encapsulation tree contains a completely structured program which directly maps to a Java program. The decompiler has been implemented and successfully tested on a variety of mid-sized benchmark programs written in Java.

The remainder of the paper is organized as follows. In Section 2 we outline the three representations, Section 3 outlines the six stages in the structuring algorithm, and Section 4 gives our conclusions together with some discussion of related work.

2. Program Representations

The restructurer is built using the Soot framework. Soot takes Java class files as input and produces several typed intermediate forms. For the purposes of the decompiler we use Soot to convert bytecode to an intermediate form called Grimp. We build a Control Flow Graph (CFG) from Grimp, and then using both the Grimp and CFG representations we build a third representation we call the Structure Encapsulation Tree (SET). The SET representation starts with no structure and is constructed by applying the six structuring stages presented in Section 3.

The example Java method shown in Figure 1 is used to illustrate each representation in the following subsections. This method is extracted from class `SearchList` and performs a linear search through a linked list of `ListNode` objects. The input bytecode for this example and the resulting decompiled output are given in Appendix A. The objective of our decompiler is to take the bytecode and create a Java program that is both semantically equivalent to the original Java program and as “natural” as possible for a human reader.

2.1. Grimp: a list of typed statements

The input to our restructurer is Grimp, a high level unstructured intermediate representation. A Grimp representation consists of a sequence of high-level statements such as aggregated expressions, directives for control flow, and the handlers for exceptions and concurrency.

Figure 2 shows the example method in Grimp format. Note that all statements that contribute semantic content to the code text have been labeled with letters. These letter labels are not part of Grimp, but have been added to aid with our presentation of the CFG and SET representations.

Although Grimp was originally designed as part of a bytecode optimizer [15], it also provides an excellent starting point for decompilation because it has already dealt with several relevant issues. For example, the Java expression stack has been eliminated, expressions such as `list.value.equals(r2)` have been aggregated, and all variables have declarations with appropriate types [5]. Except for the lack of structure and certain conventions - such as statements reflecting parameter passing (i.e. `r0 := @this;`) - Grimp very much resembles Java source code.

2.2. Control flow graph (CFG)

The Grimp representation is linear, so our first step in structuring is to build a control flow graph. Each node in the CFG wraps a single Grimp statement with predecessor, successor, dominator, and reachability information. There are two types of control flow edges in this graph, those that correspond to regular control flow and those that correspond

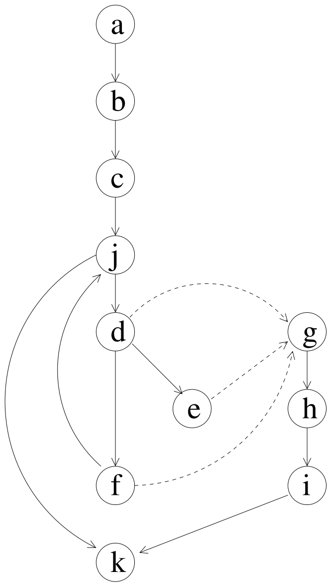

Figure 3. The CFG representation for figure 1

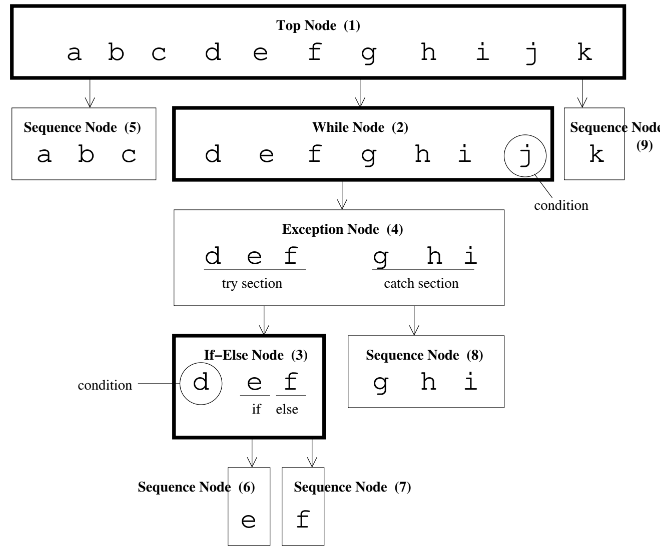

Figure 4. The SET representation for figure 1

to control flow generated by the throwing of an exception. The two types of edges are necessary because graph theoretic features such as dominance and reachability must use both types of edges while other features may consider only regular paths of execution.

Figure 3 illustrates the example method's CFG representation. Solid edges show normal control flow, while dotted edges indicate control flow generated by the potential throwing of an exception.

2.3. Structure Encapsulation Tree (SET)

The nodes of the SET represent structured Java constructs, such as `if-else` and `while` statements, and retain information about the constructs' characteristics. For example, all constructs have some type of body, so every SET node has a set of references to the CFG nodes that make up its body. Nodes corresponding to conditional statements, however, must also carry a reference to the necessary condition expression, while nodes that correspond to sequences of statements do not.

Edges in the SET represent the strict subset relations between the body sets of each SET node. As illustration, assume we have nodes a and b that respectively contain body sets A and B. If $A \supset B$, then a will be an ancestor of b.

At the start of the algorithm we have complete Grimp and CFG representations for the method being restructured. Then, starting with a SET consisting of a single Top node, whose body is the set of all nodes from the CFG, we iteratively create and add new SET nodes. New nodes may be added at a leaf, or they may be added at an intermediate level in the SET.

Figure 4 gives the final SET representation for the example program. At the beginning of the algorithm the SET representation would consist only of the Top node. Additional nodes are added in the order given by the numbers in parentheses, thus the Top node is labeled (1). The first construct detected, for this example, is the loop, so the While node is the next node added to the SET. The next stage of the algorithm detects `if` statements, so the If-Else node is added next. At this point in the algorithm the SET would consist only of the Top node with the While node as a child, and the If-Else node as a child of the While node (these three nodes are indicated with bold borders in Figure 4). Note that this would represent an intermediate point in finding the structure. Intuitively, the algorithm has discovered a set of statements comprising a `while` construct and a subset of those statements comprising an `if-else` construct. However, the SET representation does not imply that the `if-else` must be a direct child of the `while` in the final solution. The next phase of the restructurer, in fact, detects the `try-catch` construct and inserts the new TryCatch SET node. In the example, the last phases of the structuring algorithm insert the Sequence nodes.

In general, SET nodes are inserted as follows. Assume that several nodes have already been put in the SET, and that we wish to insert a newly created node. Start by setting the Top node to being the "current" node. Now, scan all the immediate children of the current node and determine if the new node is either a strict superset or a strict subset of any child. For all children who are subsets of the new node, we reassign them as children to the new node. Then, if the new node is *not* a subset of any of remaining the children, it is assigned as a child to the current node. Finally, if the new node *is* a subset of one of the children, that child is designated as the current node and the process is repeated.

This approach relies on the fact that the body sets between siblings are disjoint. Throughout the various stages of building the SET we use the information about the previous stages to try to minimize the possibility of an intersection occurring. If, however, an intersection does occur, we have specific techniques for each stage that are guaranteed to resolve the problem. The benefit to this is that, unlike a reduction based restructurer, the order in which nodes are added to the SET can be *independent of the nesting* of the control flow constructs that will be employed in the high level language representation. This allows us to look for whole graph features in the CFG rather than just isomorphisms.

3. Structuring Stages

A key feature of our approach is that it is composed of a collection of stages, where each stage in the restructuring algorithm looks for a specific set of features that indicate a desired control flow construct. These are ordered so that later stages allow more flexibility in the interpretation of target features than earlier ones. This ordering allows us to maximize our choices for "natural" restructuring.

3.1. Synchronized Blocks

Although not strictly a control flow construct, `synchronized()` blocks have a syntax which demands they nest properly within structured control flow. Java class files use `monitorenter` and `monitorexit` instructions to handle `synchronized()` blocks. To find the blocks, we scan the CFG looking for `monitorenter` instructions. When one is found, we scan forward, keeping track of monitor levels, until all paths are terminated by matching `monitorexit` instructions. To find all possible entry points to this subgraph we also scan backwards, and iterate this process until it stabilizes. At this point the set of nodes participating in a monitor region has been found.

Unfortunately, valid class files can use these instructions in two unstructured ways: 1) there may contiguous control flow subgraphs that should correspond to a `synchronized()` block, but have multiple entry points, and 2) it is possible to have two monitors on different objects where the critical sections intersect, but neither is a subset of the other, meaning that they cannot well-nest each other.

The former problem is resolved by merging the entry points to one representative entry point for the subgraph. To simulate the original control flow, then, we assign differing "signature" values to a special control flow variable on each path to the new entry point. After the point has been passed a switch instruction is inserted to dispatch control flow back to what would have been the original path.

There is no structured solution to the second problem that does not alter program behavior. To address this, we built a trivial library that implements monitors in pure Java and when presented with this case, substitute the monitor instructions with method calls to the library and abandon building the `synchronized()` block. This also provides us the option to remove existing `synchronized()` blocks at later stages in the algorithm if we find a mismatch been program structure and monitored regions.

To maximize the number of subgraphs converted to `synchronized()` blocks (and minimize the use of library calls) we build an undirected graph whose nodes represent each "monitor subgraph", and whose edges represent the "intersecting but non-nesting" property. We then choose the nodes of the minimum vertex cover of this graph to be converted to calls to the library, and the remaining nodes to be built as `synchronized()` blocks.

3.2. Loop Detection

Since strongly connected components in the CFG must be represented in the output Java source by loops, the first control flow constructs we look for are `while` and `do-while` constructs.[3] We begin by assuming that all strongly connected components will have only one entry point. This assumption is relaxed later in this section.

A Java language loop is a superset of a corresponding strongly connected component in the control flow graph. For example, a while loop can contain control flow paths terminated by return statements. Therefore, the body of a loop will be any node in the CFG that fulfills two requirements. It is 1) dominated by the component's entry point, and 2) not reachable from the control flow path of the loop condition that escapes the component.

Thus, to find the loop's body, we need to determine which statement, if any, hosts the loop condition. We split this into three cases.

1. If the entry point to the strongly connected component has a successor that escapes the component, then we have a `while` loop and the entry point is the condition statement.

2. If the entry point has a predecessor that is both in the strongly connected component *and* has a successor which escapes the component, then a `do-while` is declared. If there are several candidates for the `do-while` loop, we have several good heuristics to quickly choose the most appropriate candidate.

3. If steps 1 and 2 fail, we declare an unconditional loop, represented in Java as a `while(true)` loop.

[3] A subsequent phase could rewrite some of these into `for` loops.

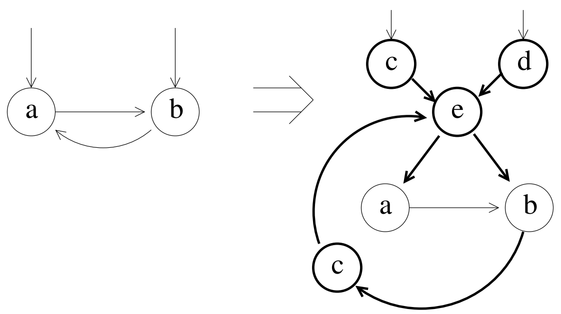

Figure 5. Resolution of irreducible multi-entry point loop.

A loop may nest other loops. In Java, for every iteration of a loop, we must execute the condition, if it is a conditional loop, or the first statement, if it is an unconditional loop. Therefore, to destroy the current loop and reveal nested loops, we remove the condition statement or the first statement from the CFG. We also know that removing this statement cannot destroy a nested loop because: 1) two Java loops cannot share the same condition, and 2) the first statement of an unconditional loop cannot be the condition for a nested loop (if it were, it would have already been evaluated as the condition for the current strongly connected component). Once we have removed the statement, we re-evaluate the strongly connected components. This process is re-iterated until the strongly connected component decomposes to a forest of DAGs and all nested loops have been found.

Finally, a strongly connected component may have several entry points. To build a Java loop, we first select one of the entry points as "natural" and treat the rest as the targets of gotos. The quick heuristic we are currently using to pick the natural entry point is to perform a depth-first-search of the strongly connected component, choosing the entry point that yields the least number of targets of back edges. We then simulate the gotos by redirecting the goto edges to the natural entry point, and follow existing control flow paths, avoiding code that would normally be run. The techniques used for guiding control flow are the same as used for multi-entry point `synchronized()` blocks.

Figure 5 gives an example resolution. Node **a** is chosen as the natural entry point, node **e** dispatches control flow according to a special control flow variable, and nodes **c** and **d** respectively set the special control flow variable to target nodes **a** and **b**. Then all control flow going to the natural entry point is re-routed to the dispatch node. For all other entry points, only the control flow from outside the component is re-routed to the dispatch node.

If there is a nesting conflict with a `synchronized()` block, the loop is given precedence and the `synchronized()` block is replaced with calls to the monitor library.

3.3. Remaining Conditionals

Once we have identified all conditions for loops, we know the remaining conditionals must be represented by `if` and `switch` statements and since Java bytecode doesn't provide indirect branching our problem is limited to finding the bodies of these constructs. We declare the body of the construct to be that part of the CFG which is: 1) dominated by the condition, and 2) reachable by only one of the conditional paths. An exception to this rule has to be made to allow for `case` fall throughs in `switch` statements.

In this case, if there is a nesting conflict when we insert into the SET, the existing nodes in the SET take precedence and we trim the body of the current conditional node so that it will nest properly. In Figure 4, only the highlighted nodes of the SET would exist at this point.

3.4. Exceptions

Information about exceptions is encoded in Java bytecode as exception tables in the code attribute for a method [7]. Each entry in the exception table gives a start and end PC, the handler PC and the catch type. The information from these exception tables occurs in Grimp as a list of trap handlers. When printing Grimp, these trap handlers are printed at the end of the method body. An example is given at the end of Figure 2. Each trap handler gives a range of Grimp statements that are under a single area of protection, an exception to watch for, and a trap handler statement. The `try` body is taken directly from the area of protection and the `catch` body consists of those statements dominated by the trap handler statement. Nesting problems may occur, especially when dealing with the code idiom that Soot generates for `try-catch-finally` clauses.[4] If there is a nesting problem with the `try` block, it is split into as many parts as necessary. For each new `try`, however, we must create a copy of the `catch` clause. If there is a problem with nesting the `catch`, the `catch` body is trimmed in the same way that the `if` and `switch` bodies are. Finally, for every new statement that is created by cloning the `catch` body, we have to add those new statements to the Top node and trickle it down the SET appropriately.

3.5. Statement Sequences

All leaf nodes in the SET now have statements which must be encapsulated in statement sequences. This is done in a single pass through the SET, creating and loading the new sequence nodes. They are well-nested by construction.

[4]Soot eliminates `jsr` bytecode instructions used to encode `finally` constructs by copying blocks of code, thus making the control flow for finally statements difficult to decompile.

3.6. Continues, Breaks and Arbitrary Labeled Blocks

We have now determined all control flow constructs and their contents, except for the special case of `continue` and `break` statements. While labeled breaks are most often used to exit loops, they are, in fact, more versatile than this and can be used to immediately exit any encapsulating labeled statement. Although this is not as general as the unstructured control flow allowed by a `goto` statement, combinations of labeled blocks and labeled breaks can be used to restructure any control flow DAG.

To deal with `continue` and `break` statements, the first task is to identify edges that represent abrupt control flow. This is straightforward because the non-abrupt exit points are well defined for every type of control flow construct. All control flow paths emerging from a SET node body that are not from such exit points are abrupt.

Continues are found by examining the predecessors of loop conditions to see if they are abrupt. If so they are labeled as being continues and not considered as possible breaks.

Next we build a set of graphs expressing the connectivity between SET siblings. Since all loops have already been expressed as constructs, there will be no loops in these new graphs and each graph can be topologically sorted. For any possible sorting we can now determine if we must convert any normal control flow path to being abrupt. There may be many correct solutions and we use several heuristics to choose a sorting which minimizes abruptness. This will then minimize labeled block insertion and the use of multi-level breaks.

Figure 6 shows a sorting of 3 nodes, G, H, and F, and some of their encapsulated children. In this example the edge DF is a `break`. Because H is F's immediate predecessor, the edge breaks from H and no labeled block needs to be inserted. The edge BF, however, cannot exist until labeled block J is created, encapsulating G and H, thus making B a member of F's immediate predecessor. BF then breaks J.

3.7. Epilogue: The AST

Once we finish the sixth phase, we have identified all the control flow constructs, have nested them properly and have identified all breaks, continues, exceptions and synchronized blocks. At this point we build an AST by traversing the SET. The AST is suitable for performing additional transformations to the structured representation (for example, detecting common idioms that may have a more natural form), and emitting Java source code.

4. Related Work and Conclusions

Research on restructuring can usually be divided into restructuring with gotos, and goto elimination. 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.

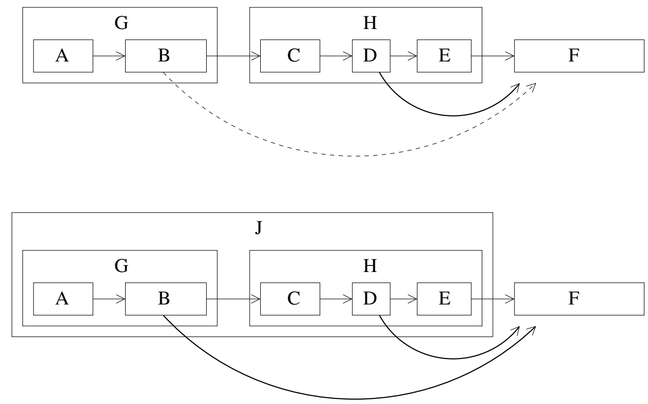

Figure 6. Arbitrary Labeled Block.

There are several Java decompilers already available. Mocha [9] and its derivatives, Jad [6], Ahpah Software's SourceAgain [13], Mayon Software Research's ClassCracker 2 [8], and WingSoft's WingDis [16] are some of the most visible.

Unfortunately, there are few research papers describing how to restructure Java and observation of the behavior of these decompilers suggests that they mostly recognize patterns that correspond to Java constructs. For example, when presented with complex control flow they may occasionally produce illegal Java with goto instructions or create a software encoding of the control flow graph using a switch and states representing nodes.

One interesting general approach is described by Proebsting and Watterson in their paper on the Krakatoa decompiler[10]. Their approach was to first apply an extended version of Ramshaw's goto-elimination technique [11], which produces legal, although somewhat convoluted, Java structures by introducing loops and multi-level breaks. They then apply a series of rewrite rules on this structured representation where each rule attempts to replace a program substructure with a more "natural" one. Our approach differs in that we attack the original graph and iteratively build up our structured representation using the SET data structure.

In this paper we have presented a staged restructuring which uses a data structure called a structure encapsulation tree (SET). The interesting characteristic of the SET data structure is that constructs can be detected and represented in any order. Each stage in the algorithm focuses on one family of Java constructs, using graph algorithms and transformations specific to those constructs. We have implemented the algorithm and it appears to do a good job of finding natural program structures. We plan to add several postpass transformations to the generated AST in order to make the generated code as natural looking as possible. The full implementation of the decompiler will be released in July 2001 as a new part of the publically-available Soot framework[12].

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. URL: `http://www.geocities.com/SiliconValley/Bridge/ /8617/jad.html`.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[8] Mayon Software, ClassCracker 2 Java decompiler. URL: `http://www.tip.net.au/~mayon/`.

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

[10] 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.

[11] L. Ramshaw. Eliminating go to's while preserving program structure. *Journal of the Association for Computing Machinery*, 35(4):893–920, October 1988.

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

[13] Ahpah Software, SourceAgain Java Decompiler. URL: `http://www.ahpah.com`.

[14] R. Valée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, July 2000.

[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. URL: `http://www.wingsoft.com/wingdis.html`.

A. Input bytecode and output Java source

Figure 7 gives the original bytecode for the example program, while Figure 8 gives the current result of the restructuring algorithm presented in this paper. The decompiled output can be further improved by renaming variables, shortening package names and removing unnecessary labels. Small improvements such as these are currently being completed. We expect to add the `Dava` decompiler to the publically available `Soot` infrastructure during July 2001.

```
Method boolean search(ListNode, java.lang.Object)
 0 bipush 112
 2 istore_3
 3 goto 43
 6 aload_1
 7 getfield #13 <Field java.lang.Object value>
10 aload_2
11 invokevirtual #14
       <Method boolean equals(java.lang.Object)>
14 ifeq 19
17 iconst_1
18 ireturn
19 aload_1
20 getfield #15 <Field ListNode next>
23 astore_1
24 goto 43
27 astore 4
29 getstatic #3 <Field java.io.PrintStream out>
32 ldc #17 <String "Error in List">
34 invokevirtual #10
       <Method void println(java.lang.String)>
37 bipush 116
39 istore_3
40 goto 47
43 aload_1
44 ifnonnull 6
47 iconst_0
48 ireturn

Exception table:
   from   to  target type
     6    24    27   <Class java.lang.Exception>
```

Figure 7. Bytecode for example Java method.

```
public boolean search(ListNode r1, java.lang.Object r2)
{
       label_0:
       while (r1 != null)
       {
            try
            {
                if (r1.value.equals(r2) == false)
                {
                     r1 = r1.next;
                }
                else
                {
                     return true;
                }
            }
            catch (java.lang.Exception r5)
            {
                java.lang.System.out.println("Error in List");
                break label_0;
            }
        }
        return false;
    }
}
```

Figure 8. Output of decompiler for example Java method.