The following paper was originally published in the
Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems
Portland, Oregon, June 1997

Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?)

Todd A. Proebsting, Scott A. Watterson
The University of Arizona

Krakatoa: Decompilation in Java
(Does Bytecode Reveal Source?)

Todd A. Proebsting Scott A. Watterson
The University of Arizona *

Abstract

This paper presents our technique for automatically decompiling Java bytecode into Java source. Our technique reconstructs source-level expressions from bytecode, and reconstructs readable, high-level control statements from primitive **goto**-like branches. Fewer than a dozen simple code-rewriting rules reconstruct the high-level statements.

1 Introduction

Decompilation transforms a low-level language into a high-level language. The Java Virtual Machine (JVM) specifies a low-level bytecode language for a stack-based machine [LY97]. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. Furthermore, the bytecode must be *verifiably well-behaved* in order to ensure safe execution. Decompilation systems can exploit this type information and well-behaved property to recover Java source code from the class file.

We present a technique for transforming low-level Java bytecode into legal Java source code. Our system, Krakatoa,[1] performs type inference to issue local variable declarations. The verifier does the same type of type inference, and the techniques are well known. Presently, we focus our research on two subproblems: recovering source-level expressions and synthesizing high-level control constructs from **goto**-like primitives.

Krakatoa uses a stack-simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations. We extend Ramshaw's **goto**-elimination algorithm to structure (and create source for) arbitrary reducible control flow graphs. This technique produces source code with loops and multi-level **break**'s. Subsequent techniques recover more intuitive constructs (e.g., **if** statements) via application of simple code rewrite rules.

Traditional decompilation systems use graph transformations to recover high-level control constructs. These systems require the author of the decompiler to anticipate all high-level control idioms. When faced with an unexpected language idiom, these systems either abort, or produce **goto**s (illegal in Java). Krakatoa represents a different approach. Krakatoa first produces legal Java source given legal Java bytecode with arbitrary reducible control flow, and *then* recovers intuitive high-level constructs from this source.

Figure 1 gives the five steps of decompilation performed by Krakatoa. First, the *expression builder* reads bytecode, recovers expressions and type information, and produces a control flow graph (CFG). Next, the *sequencer* orders the CFG nodes for Ramshaw's **goto**-elimination technique. Ramshaw's algorithm produces a convoluted—yet legal—Java abstract syntax tree (AST). Our system then transforms this AST into a less convoluted AST using a set of simple rewrites. The final phase produces Java source by traversing the AST.

*Address: Department of Computer Science, University of Arizona, Tucson, AZ 85721; Email: {todd, saw}@cs.arizona.edu.

[1] Krakatoa is a volcano located in the Sunda Strait between Java and Sumatra. Its 1983 eruption threw five cubic miles of debris into the air and was heard 2200 miles away in Australia.

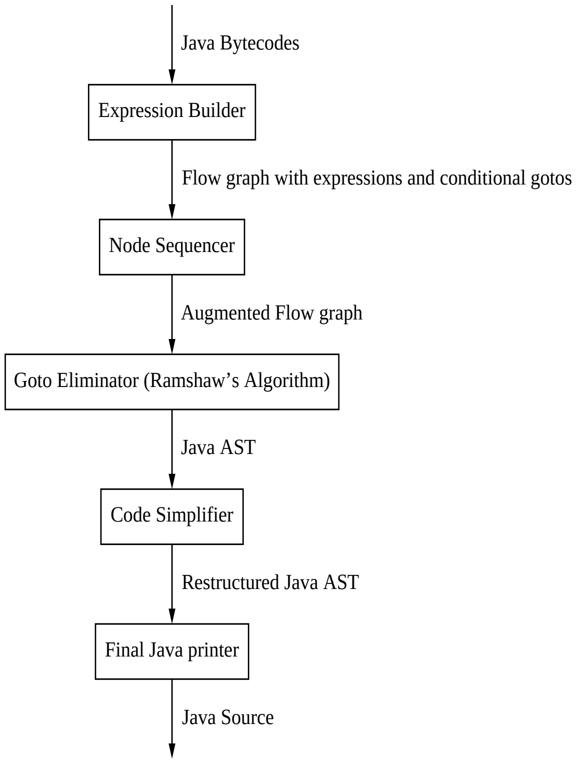

Figure 1: Java Bytecode Decompilation System

```
class foo {
   int sam;

   int bar(int a, int b) {
      if (sam > a) {
         b = a*2;
      }
      return b;
   }
}

Compiled from foo.java
class foo extends java.lang.Object {
    int sam;
    int bar(int,int);

Method int bar(int,int)
   0 aload_0
   1 getfield #3 <Field foo.sam I>
   4 iload_1
   5 if_icmple 12
   8 iload_1
   9 iconst_2
  10 imul
  11 istore_2
  12 iload_2
  13 ireturn
}
```

Figure 2: Simple Method and Bytecode Disassembly (via `javap -c`).

2 Expression Recovery

Java bytecodes bear a very close correspondence to Java source. As a result, recovering expressions from Java bytecode is often simple—much simpler than recovering expressions from machine language. Java class files include information that makes recovering high-level operations like field references easy. The fact that the bytecode must be well-behaved (i.e., verifiable) also simplifies analysis. Figure 2 gives a sample program and its abbreviated disassembly. Note the level of type information in the disassembly produced by Sun's **javap** utility.

Symbolic execution of the bytecode creates the corresponding Java source expressions. It also creates conditional and unconditional **goto**'s, which will be removed by subsequent decompilation steps. Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed. For instance, `iload_1`, which loads the value of the first local variable—with type `int`—could be represented on the stack as "`i1`". Similarly, if `i1` and `2` were the top two elements of the symbolic stack, and the next bytecode were `iadd` (integer addition), those elements would be popped off the stack and replaced with "`(i1+2)`". The symbolic execution of some expressions, like assignment, requires *emitting* Java source.

Our algorithm recovers expressions one basic block at a time. Some basic blocks (such as those produced by the conditional expression operation, *A?B:C*) do not begin with empty stacks, so some information is required to propogate from predecessors. Also, basic blocks that begin exception-handling blocks—which are easily identified—begin with the raised exception on the stack.

Figure 3 provides the step-by-step decompilation of the bytecode in Figure 2. The initial `aload_0` instruction pushes a Java reference onto the stack. In virtual functions, the "0'th" local variable, `a0`, always refers to `this`. The `getfield` instruction references a *named* field, "`sam`", of the current top of stack. Therefore, the "`this`" is popped and replaced with "`this.sam`". `iload_1` pushes "`i1`" onto the stack. The `ifcmple` compares the top two stack elements and branches to the appropriate instruction if the lower is less than or equal to the top element. Symbolically executing the `ifcmple` requires popping the top two elements and emitting the appropriate conditional branch. Translating the remaining instructions is similar.

Most of the bytecode instructions are equally simple to symbolically execute. Unfortunately, a few require more information. Some of the stack manipulation routines (e.g., `pop2`, `dup2`, etc.) depend on byte offsets from the stack top. For instance, `pop2` removes the top 8 bytes from the stack, whether those 8 bytes represent *one* 8-byte double value, or *two* 4-byte scalar values. To correctly simulate these instructions the symbolic execution keeps track of the size (and type) of each stack element.

3 Instruction Ordering

After recovering expressions, conditional and unconditional **goto**'s (along with implicit *fall through* behavior) determine control flow. Java, however, has no **goto** statement, so its control flow must be expressed with structured statements.

Ramshaw presented an algorithm for eliminating **goto**'s from Pascal programs while preserving the program's structure [Ram88]. This algorithm replaces each **goto** with a multilevel **break** to a surrounding loop. The algorithm determines the appropropriate locations for these surrounding loops. We trivially extended his algorithm to use multilevel **continue**'s.

Ramshaw's (extended) algorithm replaces each forward **goto** with a **break** and each backward **goto** with a **continue**. His algorithm inserts a loop that ends just before the target of each **break** statement. Likewise, it inserts a loop that starts just before the target of each **continue**. These loops ensure that each control-transfer statement jumps to the correct instruction. Each newly-inserted loop must also end with a **break** statement, so that control will *fall* out of the loop. Figure 4 shows an example of this technique. Additional loops and **break**/**continue**'s create a structured program with exactly the same control flow as the **goto**-only program.

Ramshaw's algorithm requires two inputs: the control flow graph, and an instruction ordering. His algorithm encodes this order into the flow graph using *augmenting edges*, such that every instruction has an augmenting edge to the next instruction in sequence. These augmenting edges occur between every pair of physically adjacent instructions even if actual control flow between them is impossible. He proves that if this augmented graph is reducible, then a *structurally equivalent* [PKT73] program can be created without **goto**'s. However, Ramshaw provides no algorithm for finding a reducible augmented flow graph from a given reducible flow graph.

The control-flow graphs of Java programs are reducible. Therefore, the compiled bytecode will likely form a reducible control-flow graph. Unfortunately, simple optimizations like loop inversion create irreducible augmented flow graphs. The flow graph of the program in Figure 8 has this problem because the augmenting edge between the first two statements creates a "jump" into the body of the loop formed by the next seven statements.

To utilize Ramshaw's algorithm, we developed an algorithm that orders a reducible graph's instructions such that the resulting augmented graph is also reducible.

3.1 Augmenting the Flow Graph

Creating a reducible augmented flow graph requires that no augmenting edge enters a loop anywhere other than at its header. Preventing this is simple—when ordering the instructions, make the header first and contiguously order the loop's instructions. Because physical adjacency determines augmenting edges, contiguously ordering the instructions guarantees that the only augmenting edge entering the loop from the outside will be entering at the top, which will not affect reducibility if it is the loop's header.

A loop with no nested loops inside is easy to order—simply remove the back edges and topologically sort the remaining directed acyclic graph (DAG). Handling interior loops requires replacing them with a single *placeholder* node in the graph and separately ordering both the loop and the surrounding graph. After ordering both, re-insert the loop's nodes at its placeholder. Re-ordering instructions may change whether or not one instruction *falls through* to another as it did in the original

Bytecode	Symbolic Stack	Emitted Source
aload_0	"this"	
getfield #3 <Field foo.sam I>	"this.sam"	
iload_1	"this.sam", "i1"	
if_icmple 12		if (this.sam <= i1) goto L12
iload_1	"i1"	
iconst_2	"i1", "2"	
imul	"(i1*2)"	
istore_2		i2 = (i1*2)
12: iload_2	"i2"	L12:
ireturn		return i2

Figure 3: Symbolic Execution of Bytecode

Before	After

*stmt*0
if *expr*1 **goto** L1;
if *expr*2 **goto** L2;
L1: *stmt*1
L2: *stmt*2
	*stmt*0
	L2: **for** (; ;) {
	L1: **for** (; ;) {
	    if *expr*1 **break** L1;
	    if *expr*2 **break** L2;
	    break L1;
	  } // L1
	  *stmt*1
	  break L2;
	} // L2
	*stmt*2

Figure 4: Ramshaw's Goto Elimination: Before and After

ordering. Where implicit control flow has changed, the algorithm must add new branches to restore the original control flow. Whenever possible, the topological sort attempts to maintain the original fall-through behavior.

This algorithm produces a reducible augmented graph. Because all loops are ordered separately, and laid out contiguously, the only augmenting edge entering from outside enters at the top. The topological sort of the loop (minus its backedges) guarantees that this top node is the loop header and that no internal edges cause irreducibility. Outside edges into the loop header cannot make a loop irreducible. Therefore, the resulting augmented graph is reducible.

Loops are not the only blocks of instructions which must be ordered contiguously. Exception handling regions must form contiguous sections of instructions. Class files specify which instructions are in which regions. Our algorithm orders those regions contiguously by treating them like loops.

After applying this technique to create a total ordering of the nodes (the augmenting path), Krakatoa can apply Ramshaw's technique to eliminate **goto**'s.

4 Code Transformations

4.1 Program Points

After applying Ramshaw's algorithm for eliminating **goto**'s, Krakatoa has a complex, yet legal, Java AST (see Figure 9). Krakatoa then proceeds to recover more of the natural high-level constructs of the original program (e.g. **if-then-else**, etc.). Krakatoa uses a *program point* analysis to summarize a program's control-flow and to guide recovering high-level constructs. A program point is a syntactic location in a program. Every statement has a program point both before and after it. These program points have two properties: *reachability* and *equivalence class*.

A program point is unreachable if and only if it is preceded along all execution paths by an unconditional jump statement (i.e. **return**, **throw**, **break**, or **continue**). For instance, in Figure 5, program point 3, Φ_3, is unreachable, since it is preceded by a **return** statement. Φ_6 is reachable, however, since one of the branches in the preceding **if** statement does not end with a jump statement.

Two program points are equivalent (denoted as $\Phi_x \approx \Phi_y$) if and only if future computation of the program is the same from both points. For instance, the program point before a **break** statement is equivalent to the program point after the loop it exits (Φ_3 and Φ_8 in Figure 6). As an example, in Figure 6, Φ_1, Φ_2, Φ_4, Φ_5, Φ_6, and Φ_7 are equivalent, as are program points Φ_3 and Φ_8.

Both reachability and equivalence are simple to compute via standard control-flow analyses [ASU86].

4.2 AST Rewrite Rules

Krakatoa performs a series of AST rewriting transformations to recover as many of the "natural" program constructs as it can (e.g. **if-then-else**, etc.). Krakatoa applies these rewriting rules repeatedly until no changes occur. We have found that the few rules below are sufficient to retrieve high-level constructs of the Java language, including **if-then-else** statements, and short-circuit evaluation of expressions. Each rewriting rule reduces the size of the AST, thus ensuring termination.

Table 1 summarizes the rules, which we describe below in greater detail. Many of these rules generalize. Those that apply to **for**-loops often apply to other loops. Many rules have several symmetric cases. For example, the first rule in Table 1 removes an empty **else**-branch from an **if-then-else** statement—there is a symmetric rule for removing an empty **then**-branch by negating the predicate.

4.3 if-then-else Rewriting Rules

The first transformation shown in Table 1 changes an **if-then-else** statement into an **if-then** statement when the **else** branch is empty. This transformation is always legal.

The second transformation creates an **if-then-else** statement from an **if-then** statement by hoisting the subsequent statement list into the **else**-part. Our algorithm performs this transformation if and only if no reachable program point in *Stmtlist1* is equivalent to the program point before *Stmtlist2*. Essentially, this means that no statement in the **then**-branch (*Stmtlist1*) can reach *Stmtlist2* directly.

4.4 Loop Rewriting Rules

The third rule in Table 1 removes useless **continue** statements. If the program point after a **continue** statement is equivalent to the program point before the **continue** statement, then that **continue** can be removed.

The fourth rule creates a short-circuit test expression within a **for**-loop by eliminating an interior **if** statement. Doing so requires that the loop body begin with an **if-then-else** statement, and that the **then** branch of that statement consists of a single jump to a program point equivalent to breaking out of the loop.

The fifth transformation provides an example of transforming loops into **if** statements. A loop is equivalent to an **if** if it can never repeat itself, and if all simple **break** statements can be safely removed during the transformation. A loop never repeats if its last program point is unreachable. **break**'s may be removed if the immediately following (unreachable) program point is equivalent to the last program point in the loop (Φ_1 in Table 1). The transformation replaces the loop with an **if** statement, and deletes all of the **break** statements for that loop.

4.5 Short Circuit Evaluation Rewriting Rules

The sixth rule shown in Table 1 recovers a short-circuit **Or** conditional. Short-circuit **Or**'s exist when two adjacent conditionals guard the same statement list and failure of either will cause a branch to equivalent locations.

The last transformation in Table 1 recovers short-circuit **And** expressions. This transformation is applicable whenever a simple **if** statement represents the entire body of another.

5 Status

We have implemented a prototype Java decompiler, Krakatoa, in Java. We have run Krakatoa on a number of class files, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well. Figures 7–10 provide an example of the stages of decompilation.

Rule	Before	After	Conditions

Eliminate Else	**if** *expr* {
	Stmtlist
	} **else** { }
		if *expr* {
		Stmtlist
		}
			None

Create **if-then-else**	**if** *expr* {
	Stmtlist1
	}
	Φ_1 : *Stmtlist2*
		if *expr* {
		Stmtlist1
		} **else** {
		Stmtlist2
		}
			Stmtlist1 contains no reachable program points equivalent to Φ_1.

Delete Continues	**for** (A ;B ;C) {
	Stmtlist
	Φ_1 **continue** Φ_2
	}
		for (A;B ;C) {
		Stmtlist
		}
			$\Phi_1 \approx \Phi_2$

Move Conditionals	**for** (A ;B ;C) {
	if *expr* {
	Φ_1 **jump**
	} **else** {
	Stmtlist1
	}
	Stmtlist2
	}
	Φ_2
		for (A ;
		B **and not** *expr* ;
		C) {
		Stmtlist1
		Stmtlist2
		}
			$\Phi_1 \approx \Phi_2$, X is either a **break** or **continue**

Remove Loop	**for** (*stmt* ; *expr* ;) {
	Stmtlist
	Φ_1
	}
	Φ_2
		stmt
		if *expr* {
		Stmtlist′
		}
			Stmtlist contains no reachable program points equivalent to Φ_1. The program point after any **break** must be equivalent to Φ_1.

Create Short Circuit Or's	**if** *expr1* {
	Φ_1 X
	} **else** {
	if *expr2* {
	Φ_2 Y
	} **else** {
	Stmtlist
	}
	}
		if *expr1* **or** *expr2* {
		X
		}
		else {
		Stmtlist
		}
			X and Y are equivalent jumps. (I.e., $\Phi_1 \approx \Phi_2$.)

Create Short Circuit And's	**if** *expr1* {
	if *expr2* {
	Stmtlist
	}
	}
		if *expr1* **and** *expr2* {
		Stmtlist
		}
			Neither **if** stmt has an **else** branch

Table 1: Canonical Code Transformation Rules

```
Φ1
if ( a < b ) {
    Φ2
    return a;
    Φ3 // (unreachable)
}
else {
    Φ4
    a = b;
    Φ5
}
Φ6
```

Figure 5: Reachable Points

```
Φ1 //  { Φ2, Φ4, Φ5, Φ6, Φ7 }
for ( ; ; ) {
    Φ2
    if ( a < b ) {
        Φ3 //  { Φ8 }
        break;
        Φ4 //  (unreachable)
    }
    else {
        Φ5
        continue;
        Φ6 //  (unreachable)
    }
    Φ7 //  (unreachable)
}
Φ8
```

Figure 6: Equivalent Points

Figure 7 shows the original source code of a sample program. Figure 8 shows the results of expression decompilation on the bytecode of this program. Figure 9 shows the results of applying Ramshaw's algorithm to the decompiled expression graph. Figure 10 shows the result of the grammar rewriting rules applied to the output of Ramshaw's algorithm. Obviously, using DeMorgan's laws would simplify the boolean expressions. Future versions of Krakatoa will do so.

For the JVM `dup` operators, which duplicate stack elements, Krakatoa simply creates a temporary variable to hold the duplicated value. This yields unnatural, but easily readable, decompilations. A more difficult problem is our failure to recover the conditional-expression operator, "**? :**". This operation presents two difficulties: it requires determining short-circuit operators during expression recovery, and it requires that expression recovery handle non-empty stacks at basic block boundaries. Fortunately, the short-circuit problem can be handled easily with four simple graph-writing rules given in [Cif93]. The non-empty stack problem is difficult because it requires combining expressions in our symbolic stack upon entering a basic block with multiple predecessors. Krakatoa again uses a temporary variable to hold the result of each branch of the conditional expression, and then assigns this temporary value to the conditional expression. We are currently investigating other solutions to this problem.

Appendix B contains additional examples of Krakatoa's output.

6 Countermeasures

Krakatoa is very effective at reproducing readable Java source from Java bytecode. This may be alarming to those who want to protect their source code from unwanted copying. Unfortunately, there are few countermeasures.

One could introduce irreducible control-flow through bogus conditional jumps to foil Ramshaw's algorithm. This, however, only stops the recreation of high-level constructs. Krakatoa could simply produce source code in a Java-like language extended with **goto**'s.

One could introduce bizarre stack behavior to foil expression recovery. This is difficult, however, because the behavior cannot be so bizarre as to yield unverifiable bytecode. It is possible, however, to create many bogus threads of control (i.e., threads that will never execute) that will confuse the expression recovery mechanism in basic blocks that are entered with non-empty stacks.

One code obfuscation technique that is modestly effective is to change the class file's symbol table to contain bizarre names for fields and methods. So long as cooperating classes agree on these names, the class files will link and execute correctly [vV96, Sri96].

Another suggested solution is to use dedicated

```
class foo {
  void foo(int x, int y) {
    while ((x + y < 10) && (x > 5)) {
      if ((y > x) || (y < 100)) {
        x = y;
      }
      else {
        x += 100;
      }
    }
  }
}
```

Figure 7: Original Source

```
class foo {
  void foo(int i1, int i2) {
          goto L4;
      L1: if (i2 > i1) goto L2;
          if (i2 >= 100) goto L3;
      L2: i1 = i2;
          goto L4;
      L3: i1 += 100;
      L4: if ((i1+i2)>=10) goto L5;
          if (i1 > 5) goto L1;
      L5: return;
  } // foo
} // foo
```

Figure 8: After Expression Decompilation

```
class foo {
  void foo(int i1, int i2) {
lp3:for ( ; ; ) {
      if ((i1 + i2) >= 10) break lp3;
      if !((i1 > 5)) break lp3;
lp2 : for ( ; ; ) {
lp1 :   for ( ; ; ) {
          if (i2 > i1) break lp1;
          if !((i2 >= 100)) break lp1;
          break lp2;
        } // lp1
        i1 = i2;
        continue lp3;
      } // lp2
      i1 += 100;
      continue lp3;
    } // lp3
    return;
  }
}
```

Figure 9: After Goto Elimination (Ramshaw's Algorithm)

```
class foo {
    void foo(int i1, int i2) {
lp3:  for ( ;!((i1+i2)>=10)&&((i1>5)); ) {
        if (i2 > i1) || !((i2 >= 100)) {
          i1 = i2;
        } // then
        else {
          i1 += 100;
        }
      } // lp3
      return;
    }
}
```

Figure 10: After AST Transformation (Final Decompilation Results)

hardware and encryption to protect class files [Wil97].

Many traditional countermeasures to reverse-engineering will not work for Java bytecode. It is impossible to mix code and data. It is impossible to jump to the middle of instructions. It is impossible to generate bytecode and then jump to it.

7 Related Work

Ramshaw presented a technique for eliminating **goto**'s in Pascal programs by replacing them with multilevel **break**'s and surrounding loops [Ram88]. He made no attempt to recover high-level control constructs. All high-level control structures were provided by the original Pascal.

Several decompilation systems have used a series of graph transformations to recover high-level constructs [Lic85, Cif93]. These systems encounter difficulties in the presence of nested loops, and other arbitrarily control flow. Multilevel **break**'s cause considerable problems. Exception handling introduces another difficulty to such systems, as the control flow graph can be entered in several places. Krakatoa easily creates multi-level **break**'s and **continue**'s, and is able to eliminate virtually all of the unnecessary ones via successive application of the rewrite rules.

"Mocha" (version 1 beta 1) [vV96] is a Java decompiler written by Hanpeter van Vliet. Mocha uses graph transformations to recover high-level constructs. Mocha often aborts when it confronts tangled—yet structured—control flow (including multi-level **break**'s and **continue**'s). The system does issue type declarations, and uses debugging information (when present) to recover local variable names.

Other graph transformation systems used node-splitting to transform an unstructured graph to a structured graph [WO78, PKT73, Wil77]. Peterson, Kasami, and Tokura present a proof that every flow graph can be transformed into an equivalent well-formed flow graph. Williams and Ossher use a similar technique, but they recognize five unstructured sub-graphs, and replace those with equivalent structured graphs. Node-splitting preserves the execution sequence of a program, but not the structure. We do not consider this reasonable for decompilation.

Baker presents a technique for producing programs from flow graphs [Bak77]. Baker generates summary control flow information to guide her graph transformations. Our goal is similar, since the output of the decompiler should be as readable as possible. Her technique structures old FORTRAN programs for readability. As a result, her technique may leave some **goto**'s in the resulting programs, which is not allowed in Java.

Other techniques for eliminating **goto**'s have been proposed [EH94, Amm92, AKPW83, AM75]. These techniques may change the structure of the program, and may add condition variables, or create subroutines.

8 Conclusion

In this paper, we present a technique for decompiling Java bytecode into Java source. Our decompiler, Krakatoa, produces syntactically legal Java source from legal, reducible Java bytecode. We focus on two subproblems of decompilation: recovery of expressions from Java's stack-based bytecode, and recovery of high-level control-flow constructs. We present our stack simulation method for recovering expressions. We present an extension of Ramshaw's **goto** elimination technique that can be applied to any reducible control-flow graph.

We also present a small, yet powerful, set of code rewriting rules for recovering the natural high-level control-flow constructs of the Java source language. These rewrite rules enable Krakatoa to successfully decompile many class files that graph transformation systems fail. If Krakatoa is presented with a high-level language idiom that it does not recognize, it may leave unnecessary **break**s or **continue**s in the code. It will still produce legal Java, however. If a system relies on a graph transformation system to produce high-level constructs, it will fail when presented with an unexpected construct.

Our techniques, combined with the abundant type information available in class files, make decompilation of Java bytecode quite effective.

9 Acknowledgment

Saumya Debray helped develop the instruction ordering algorithm.

References

[AKPW83] J.R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. pages 177–189, 1983.

[AM75] E. Ashcroft and Z. Manna. Translating programs schemas to while-schemas. *SIAM Journal of Computing*, 4(2):125–146, 1975.

[Amm92] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(2):237–250, 1992.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[Bak77] Brenda S. Baker. An algorithm for structuring flowgraphs. *Journal of the Association for Computing Machinery*, 24(1):98–120, January 1977.

[Cif93] Cristina Cifuentes. A structuring algorithm for decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Informatica*, pages 267–276, Buenos Aires, Argentina, August 1993.

[EH94] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. pages 229–240. International Conference on Computer Languages, May 1994.

[Lic85] Ulrike Lichtblau. Decompilation of control structures by means of graph transformations. In C. F. M. Nivat Hartmut Ehrig and J. Thatcher, editors, *Mathematical foundations of software development: Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT 85): volume 1 - Colloquium on Trees in Algebra and Programming (CAAP '85)*, volume 185 of *Lecture Notes in Computer Science*, pages 284–297. Springer-Verlag, March 1985.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1997.

[PKT73] W.W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat and exit statements. *Communications of the ACM*, 16(8):503–512, 1973.

[Ram88] Lyle Ramshaw. Eliminating go to's while preserving program structure. *Journal of the Association for Computing Machinery*, 35(4):893–920, October 1988.

[Sri96] KB Sriram. Hashjava. url: http://www.sbktech.org/hashjava.html, 1996.

[vV96] Hanpeter van Vliet. Mocha. current url: http://www.brouhaha.com/~eric/computers/mocha-b1.zip, 1996.

[Wil77] M.H. Williams. Generating structured flow diagrams: The nature of unstructuredness. *Computer Journal*, 20(1):45–50, 1977.

[Wil97] U. G. Wilhelm. Cryptographically protected objects, May 1997. A french version appeared in the Proceedings of RenPar'9, Lausanne, CH. `http://lsewww.epfl.ch/~wilhelm/CryPO.html`.

[WO78] M.H. Williams and H.L. Ossher. Conversion of unstructured flow diagrams to structured. *Computer Journal*, 21(2):161–167, 1978.

A Additional Rewriting Rules

We anticipate using a few other tree rewriting rules that might improve readability of our code. The anticipated rules build more natural **for**-loops. Table 2 presents addition code transformation rules that could be applied by Krakatoa. We expect to add these rules as we re-implement Krakatoa in Java.

B Sample Decompiler Output

We've included a representative sampling of Krakatoa's output on a classfile that implements sets in Java. The original Java source is on the left and Krakatoa's output is on the right. Table 3 provides original source class definitions as well as the

Rule	Before	After	Conditions

Include Init	I
	for (; *expr* ; *update*) {
	Stmtlist
	}
		for (I ; *expr* ; *update*) {
		Stmtlist
		}
			I is a simple statement

Include Update	**for** (*init* ; *expr* ;) {
	Stmtlist
	U
	Φ_1 }
		for (*init* ; *expr* ; U) {
		Stmtlist
		}
			Stmtlist contains no reachable program points equivalent to Φ_1. U is a simple statement

Table 2: Additional Code Transformation Rules

<table>
<tr><th>Original Source</th><th>Output from Krakatoa</th></tr>
<tr><td><pre>
import java.io.PrintStream;
import java.util.Vector;

public class Set
     implements Cloneable {

   // class variables
   static boolean echo_ops;

   // instance variables
   protected Vector members;

  // functions are defnied here....

}
</pre></td><td><pre>
import java.io.PrintStream;
import java.util.Vector;

public class Set
     extends java.lang.Object
     implements java.lang.Cloneable {

    static boolean echo_ops;

    protected java.util.Vector members;

    // functions are defined here...

}
</pre></td></tr>
</table>

Table 3: Class definition output from Krakatoa

corresponding Krakatoa output. Table 4 provides original source of several small functions together with Krakatoa output for those functions. Table 5 shows a larger function in original source as well as Krakatoa output for that function.

Original Source	Output from Krakatoa

public boolean isMember(Object o) {
 	
  return (members.contains(o));
} // isMember
	public boolean isMember(
	              java.lang.Object local1) {
	   return this.members.contains(local1);
	} // isMember

public void addMember(Object o) {
 	
  if (!(isMember(o))) {
    members.addElement(o);
  } // then
 	
} // addMember
	public void addMember(
	              java.lang.Object local1) {
	
	 if !((this.isMember(local1) != 0)) {
	    this.members.addElement(local1)
	 } // then
	 return;
	} // addMember

public void removeMember(Object o) {
 	
  members.removeElement(o);
 	
} // removeMember
	public void removeMember(
	              java.lang.Object local1) {
	
	 this.members.removeElement(local1)
	 return;
	} // removeMember

public int size() {
  return members.size();
} // size
	public int size() {
	   return this.members.size();
	} // size

boolean equals(Set s) {
  Set d1, d2;
 	
  d1 = difference(s);
  d2 = s.difference(this);
 	
  return ((d1.size() == 0) &&
          (d2.size() == 0));
 	
}
	boolean equals(Set local1) {
	 Set local2;
	 Set local3;
	
	 local2 = this.difference(local1);
	 local3 = local1.difference(this);
	 if !(((local2.size() != 0) ||
	       !((local3.size() == 0)))) {
	     return 1;
	 } // then
	 else {
	     return 0;
	 } // if
	} // equals

Table 4: Member Functions: Original Source and Krakatoa output

<table>
<tr><th>Original Source</th><th>Output from Krakatoa</th></tr>
<tr><td><pre>
// This returns a NEW set, with all of
// the elements from this set and
// Set s.
public Set union(Set s) {
  Set out;
  int size;
  int i;
  Object obj;

  if (echo_ops) {
    System.out.println("unioning");
  }

  out = new Set();

  out.members = (Vector) members.clone();

  size = s.size();

  for (i = 0; i < size; i++) {
    obj = s.members.elementAt(i);
    if (!(out.isMember(obj))) {
      out.addMember(obj);
    } // then

  } // for

  return out;
} // union
</pre></td><td><pre>

public Set union(Set local1) {
  Set local2;
  int local3;
  int local4;
  java.lang.Object local5;

 if !((Set.echo_ops == 0)) {
  java.lang.System.out.println("unioning");
 } // then

 local2 = new Set();

 local2.members = ((java.util.Vector)
                  this.members.clone());

 local3 = local1.size();
 local4 = 0;
 loop3 :
   for (; !(!(local4 < local3)) ;) {
     local5 =
          local1.members.elementAt(local4);
     if !((local2.isMember(local5) != 0)) {
         local2.addMember(local5)
     } // then
     local4 += 1;
 } // loop3

  return local2;
} // union
</pre></td></tr>
</table>

Table 5: Member functions: Original Source and Krakatoa output