McGill University
School of Computer Science
Sable Research Group

Of Graphs and Coffi Grounds: Decompiling Java [1]

Sable Technical Report No. 6

Patrick Lam

September 10, 1998

Abstract

Java programmers write applications and applets in plain English-like text, and then apply a java compiler to the text to obtain *class files*. Class files, which are typically transmitted across the web, are a low-level representation of the original text; they are not human-readable. Consider a compiler as a function from text to class files. My goal is to compute the inverse function: given the compiled class file, I wish to find the best approximation to the original text possible. This is called decompilation.

Given a class file, one can build an unstructured graph representation of the program. The main goal of my work is to develop graph algorithms to convert these unstructured graphs into structured graphs corresponding to high-level program constructs, and I will discuss these in detail. I shall also mention some results concerning possibility and impossibility which show that decompilation is always possible if the program may be lengthened.

1 Introduction

A computer program is written as plain human-readable text, and then converted, using a compiler, into a form that the computer can directly execute. A question that may be asked, then, is how effectively the compilation process can be reversed; given an executable file, we wish to return to the textual source file. This is decompilation.

The Java programming language[AG98], when compiled, produces executable code that is particularly information-rich, and thus especially amenable to decompilation. Several Java decompilers already exist, with varying degrees of success at decompilation and varying degrees of secrecy surrounding the innards of the decompilers.

As a project of the Sable research group, a decompiler for Java, called Sculpt, is being written. This paper discusses ongoing work in the control-flow restructuring phase of the decompiler, and quotes some results concerning the possibility and impossibility of control-flow restructuring under various assumptions.

Section 2 provides some background information on the problem of Java decompilation, including a brief summary of salient features of the Java language. The decompilation process itself has three main stages: handling local (nice) structures, handling non-local structures, and handling arbitrary structure; a detailed description of these can be found in section 3. Finally, section 4 reviews some possibility and impossibility results for control-flow restructuring, culminating in the statement of a theorem proving possibility with appropriate assumptions; it is followed by a discussion of future work and a summary.

The local reductions are implemented almost directly from [Muc97]; the concept of a non-local reduction and the reductions used in this stage were developed by the author independent of the literature. The possibility and impossibility results are quoted from the literature without proof.

2 Background

2.1 Compilation and decompilation

Computer programs are written by programmers as textual source code, and then translated into machine-readable form by a program called a *compiler*. Furthermore, when programming in Java, programs can be written as Java applets, which can be transmitted across the Internet and executed on remote computers. (See Figure 1.)

```
class Lebesgue
{
   public Number measure()
   {
      ...
   }
   ...
}

0100101010111100010100101
0101011010101011100101011
```

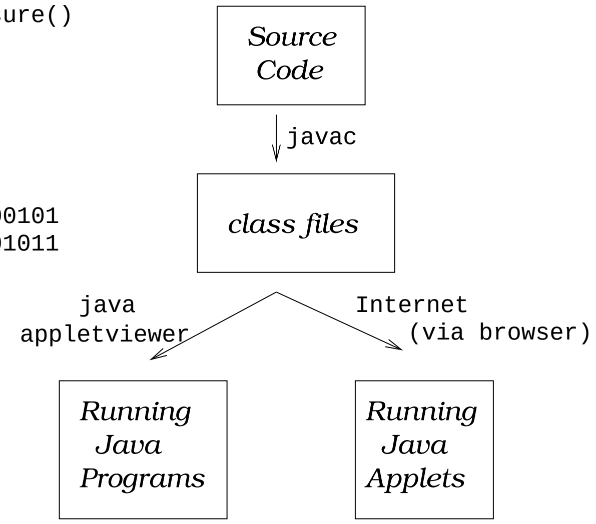


Figure 1: How Java is ground: the compilation process.

If a compiler is seen as a function from human-readable `.java` files to non-readable `.class` files, then a *decompiler* computes the inverse function: given a `.class` file F, compute the most likely `.java` program that compiles to F.

Compilers are many-to-one functions and not surjective, so there may not be any corresponding Java file, or there may be several.

Our goal is to produce a decompiler that will decompile all `.class` files, producing code as close as possible to what the programmer originally wrote. In particular, when a `.class` file is decompiled and then recompiled, the final recompiled `.class` file should behave identically to the original `.class file`.

2.2 Why decompilation?

There are several reasons to write a decompiler. The Sable Research Group deals with Java compiler optimizations, and it is instructive to optimize a program, and then decompile the optimized code to compare it to the original code. A programmer who loses his source files but keeps a compiled copy of his program around can recover his source files. Similarly, it becomes possible to modify other peoples' Java applets. Also, there are interesting graph problems to solve; finally, a decompiler is generally a fun program to have.

In particular, compiled Java class files contain a lot of information; it should be possible to produce files that are particularly close to the original Java code. Also, many class files are distributed over the Internet as Java applets, providing a large source of files to decompile.

2.3 Other decompilers

Others have already written Java decompilers. A quick search reveals the traditional three decompilers: DeJaVu, WingDis, and Mocha, which were the first decompilers generally available, and several others, such as Jad, Jasmine, and SourceAgain. These decompilers have varying degrees of success and are protected by varying degrees of secrecy; our decompiler, Sculpt, will be free software distributed under the GNU Library General Public License, and aims to succeed on arbitrary class files.

The above decompilers are linked from the Java Code Engineering & Reverse Engineering Page: `http://meurrens.ml.org/ip-Links/Java/codeEngineering/decomp.html`

2.4 The Soot framework

Soot is a project of the Sable Research Group, which belongs to the Compilers and Concurrency Lab at McGill University. Sable's mission is to study Java optimizations, and Soot is a framework for intermediate representations of Java; some parts of the Soot framework are illustrated in Figure 2.

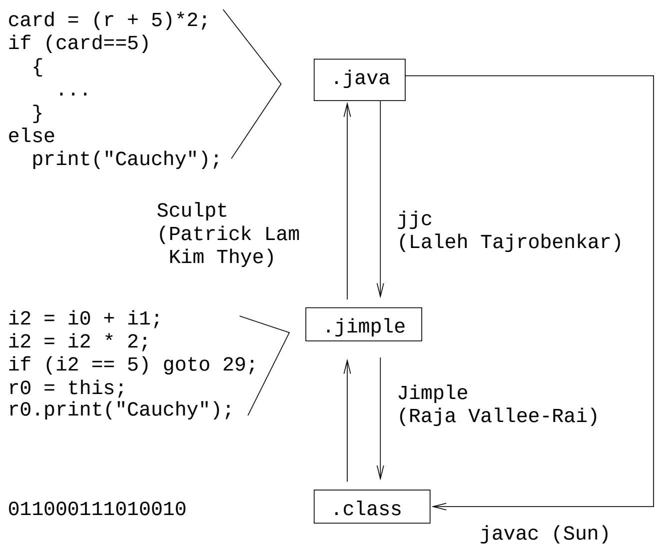


Figure 2: Parts of the Sable framework.

Good notation makes math easy to do; good intermediate representations make optimizations and the analyses required to support them easy to do. Our primary intermediate representation is called Jimple and described in [VRH98]; a `.class` to `.jimple` translator by Raja Vallee-Rai is in testing.

Jimple currently uses the Coffi library by Clark Verbrugge to manipulate `.class` files effectively; however, the use of Coffi is being phased out, and will replaced by Baf, which will be our low-level representation for representing the bytecode; it will allow us to manipulate class files with an interface consistent with that of the other parts of the Soot framework.

The Grimp intermediate representation is a higher-level representation and will be similar to Jimple, except that it will contain aggregated expressions, which are needed for some of the structuring algorithms used.

Our decompiler, Sculpt, currently uses the intermediate code representation output by Jimple and produces `.java` files. The decompiler will eventually use the higher-level Grimp representation.

Another student, Laleh Tajrobenkar, is writing a compiler that brings source `.java` files to the Jimple intermediate representation (which is easy to compile into `.class` files).

Can the task of Java decompilers be made more difficult? Phong Co is studying techniques by which a `.class` file can be made extremely difficult to decompile.

Various other Sable projects are ongoing. They are described on the Sable web page at `http://www.sable.mcgill.`

The Sable Research Group is under the supervision of Laurie Hendren, a professor at McGill University. She provides advice and direction, finds funding, and keeps the Sable project on track.

2.5 Java programming orientation

I will provide a brief description of the Java concepts necessary for the remainder of the paper.

A Java program is composed of a set of classes (see Figure 3). These classes contain methods and data. Our basic unit of interest is the method, which is a list of computer instructions. It is conceptually similar to a function, although sometimes it will not return a value; it may change the state of the machine. Methods are independent and we process one method at a time.

The *flow of control* is the path taken by the computer through the program. We can encode the possible control flow in a *control-flow graph.*

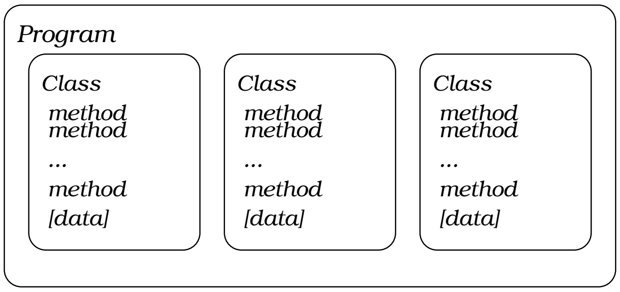


Figure 3: A Java program and some class files.

3 The Decompilation Process

3.1 Overview of decompilation

Starting from the intermediate Jimple code, there are two major phases in the decompilation to `.java` files.

Jimple code is unstructured; there is arbitrary flow of control. Furthermore, Jimple is a three-address code; that is, expressions are represented piece-by-piece. For instance, Jimple code looks like this:

```
c = a + b;
d = a * 2;
e = c / d;
```

Note that each instruction only contains a binary operation and a destination.

Kim Thye is writing an expression aggregator, to create unstructured Jimple code with complex expressions (which we will call Grimp). This code still contains arbitrary flow of control. An example of Grimp code is:

```
e = (a + b) / (a * 2);
```

Finally, the topic of this paper is the last phase: we transform control-flow graphs with arbitrary flow of control into structured Java code with regular flow of control; the structure of the code is made evident. A control-flow graph is shown in figure 4.

```
public static int gcd(int x, int y)
{
  if (x < y>
  {
    int t = x; x = y; y = t;
  }
  while (y != 0)
  {
    int t = y; y = x % y; x = y;
  }
  return x;
}
```

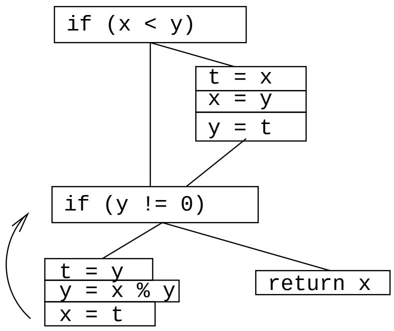


Figure 4: Example of Java code and a control-flow graph.

3.2 Stages of control-flow structuring

There are three main stages to the control-flow structuring implemented in Sculpt. The first stage handles all the simple cases in the control-flow graph. If unsuccessful, the second stage is invoked, which handles some slightly harder, but still very structured, cases. Finally, when all else fails, the third stage runs. This stage will provably handle all graphs correctly (the semantics are unchanged), but it will almost certainly produce code that differs from what the programmer originally wrote.

The overall algorithm used in control-flow structuring is as follows:

- **while** reduction is still occurring:
  - try to apply stage I reductions.
  - **if** no reduction occurs, **then** apply stage II.
  - **if** graph has not been completely reduced and stages I and II fail, **then** apply stage III.

Stage I deals with *local* reductions to simple graph components that are evident in the control-flow graph. We repeat this process until all local reductions are exhausted. An example of a stage I reduction is shown in Figure 5.

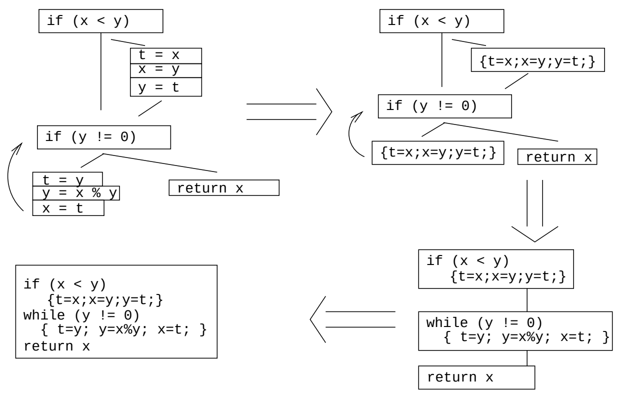


Figure 5: Reduction of the gcd method.

Stage II is run once Stage I reductions can no longer be done. It removes structured impediments to the Stage I reductions, after which the Stage I reductions are attempted again. A stage II reduction is illustrated in Figure 6.

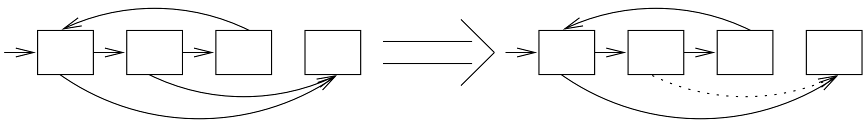


Figure 6: A generic Stage II reduction.

Java methods might contain structures that neither Stage I nor Stage II could handle: they carry no guarantees of success. When this happens, we have techniques that will always work, producing code which runs identically to the original code; however, the results produced will not look like the original code. If Stage III must be invoked, however, it is likely that the code was run through an obfuscator, optimizer, or simply is not the output of a Java compiler. Figure 7 shows a schematic example of a Stage III restructuring.

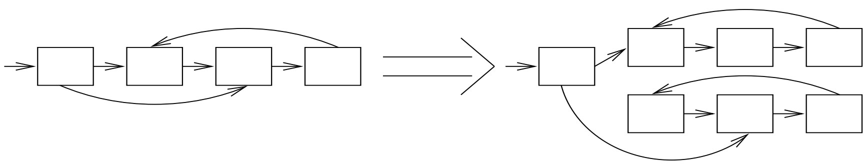


Figure 7: A generic Stage III reduction.

Some results have been obtained using Stages I and II; an example follows in figure 3.2.

We now illustrate parts of Java programs which we will structure. As these will be shown as graph components, we first define the notion of a directed graph.

```
public static int gcd
       (int x, int y)
{
  if (x < y)
  {
    int t = x;
    x = y;
    y = t;
  }
  while (y != 0)
  {
    int t = y;
    y = x % y;
    x = t;
  }
  return x;
}
```

```
public static int gcd(int, int)
{
  int i0, i1, i2;

  i0 := @parameter0;
  i1 := @parameter1;
  if i0 >= i1 goto label1;

  i2 = i0;
  i0 = i1;
  i1 = i2;
  goto label1;

label0:
  i2 = i1;
  i0 = i0 % i1;
  i1 = i0;
  i0 = i2;

label1:
  if i1 != 0 goto label0;

  return i0;
}
```

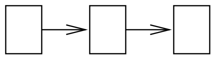
```
public static int gcd
            (int, int)
{
  int i0, i1, i2, i3;

  i1 := @parameter0;
  i2 := @parameter1;
  if (i1 >= i2)
    ;
  else
  {
    i0 = i1;
    i1 = i2;
    i2 = i0;
  }
  while (i2 != 0)
  {
    i3 = i2;
    i0 = i1 % i2;
    i2 = i0;
    i1 = i3;
  }
  return i1;
}
```

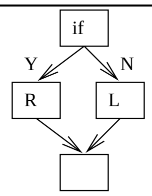
Figure 8: Decompilation of a simple method.

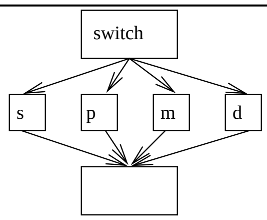
Definition 1 *Define $G = (V, E)$ to be a directed graph, where V is a finite set of vertices, and $E \subseteq V \times V$ is a set of edges. An edge (i, j) is an ordered pair of vertices.*

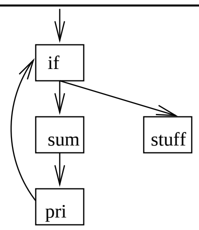
We now present various Java constructs, with skeletal Java code on the left and the corresponding graph components on the right.

Java construct	Graph Component

`/* Basic Block */`
`Cauchy(5);`
`nilpotent(27);`
`Lagrange();`
	

`if (f.step())`
`  Riemann(f)`
`else`
`  Lebesgue(f)`
`...`
	

`switch(t)`
`{`
`    case simple:  ...`
`      break;`
`    case positive:  ...`
`      break;`
`    case measurable:  ...`
`      break;`
`    default:  ...`
`      break;`
`}`
	

`while (sum > 1729)`
`  {`
`      sum += num;`
`     print(sum);`
`  }`
`stuff`
	

Note that nowhere in the above table was a `goto` statement, arbitrarily transferring the flow of control. This is because Java supports only structured flow of control. However, intermediate Grimp code does have arbitrary flow of control (it allows the `goto` statement.)

3.3 Implementation details for Stage I reductions

In this subsection we describe the conditions neeeded for recognition of each of the basic Stage I reduction types. The basic concept of a Stage I reduction is described as structural analysis in [Muc97].

3.3.1 Basic Block

First we find basic blocks. A basic block is a chain of statements with only one entry point and only one exit point. Formally, for each node $n \in N$, we collect the set of nodes M such that $m \in M$ if $|\texttt{succs}(m)| = |\texttt{preds}(m)| = 1$ and either:

- m is a direct successor of n, or:

- m is a direct successor of $m' \in M$

We also add the last node m in a straight-line chain, even if $|\texttt{succs}(m)| \neq 1$. We do require $|\texttt{preds}(m)| = 1$, though.

Figure 9 shows an example of a basic block.

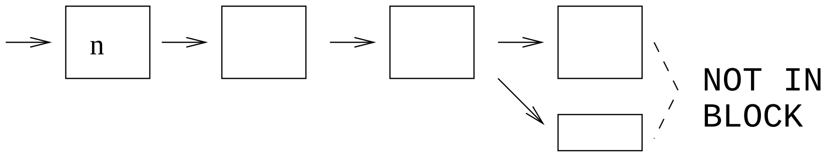


Figure 9: Example of a basic block.

3.3.2 If-then-else Block

An `if-then-else` block corresponds to an `if-then` statement in the original Java code. In order to declare a graph component to be an `if-then-else` block, we require an `if` node q, with two successors m and n; furthermore, m and n must have a unique common successor p and a unique common predecessor n. Figure 10 shows an example of an `if-then-else` block.

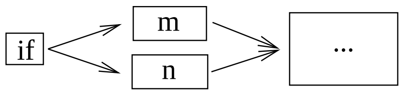


Figure 10: Example of an if-then-else block.

Note that this reduction only goes through if m and n, the branches of the conditional, have previously been reduced. This only happens if there were no jumps out of the `then` and `else` cases.

In figure 11, observe an `if-then-else` case which cannot be handled with uniquely Stage I reductions, as control flow jumps out of one of the branches of the conditional.

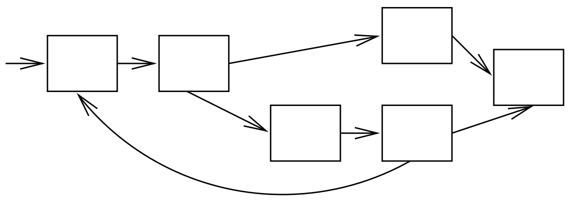


Figure 11: A graph that can't be reduced by stage I.

3.3.3 Switch Block

A `switch` block contains an integral expression e and a set of cases, each containing code to execute; the selection is made on the basis of the value of e. Refer to Figure 12 for an example of a `switch` block.

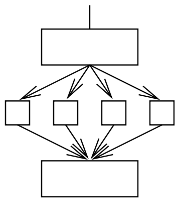


Figure 12: Example of a `switch` statement.

However, the assumption required for Stage I reductions, that subcomponents of the graph get previously reduced, fails especially often for `switch` statements. We deal with `switch` in phase II of the decompilation.

3.3.4 Cyclic control structures – the `while` loop

A `while` loop executes a set of statements, called the loop body, repeatedly, as long as a specified test condition is true. We will reduce structures that have the shape outlined in Figure 13.

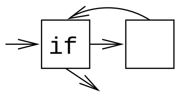


Figure 13: Example of a `while` statement.

In particular, we look for a set of 2 nodes, one of which is a test conditional, and one of which is a loop body. As always, the assumption required is that the loop body has previously been reduced to a single node.

3.3.5 More cyclic structures – `do-while` loop

This is a minor variant on the `while` loop which requires special treatment. As seen in Figure 14, the loop body is executed first, then the condition is checked. Once again, the loop body must be previously reduced for the larger reduction to carry through.

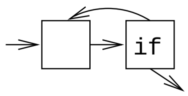


Figure 14: Example of a `do-while` statement.

3.3.6 The `for` loop

A `for` loop has three parts:

```
for (precondition; testcondition; postcondition)
  loop_body;
```

Before the loop starts, the precondition is executed. Next, the test condition is evaluated; if it is true, the loop body is executed, followed by the postcondition. Finally, the flow of control returns to the test condition, and runs through the loop again, until the test condition evaluates to false.

The `for` loop is almost a special case of the `while` loop. A heuristic will be used in a final cleanup pass to detect likely `for` loops. Note that in most cases we can't distinguish between a `while` and a `for` loop, but we can make a good guess. The duality is shown in the table below; both sides get compiled into the same `.class` file:

`for (i = 0; i < 5; i++)`
`  print(i);`
	`i = 0;`
	`while (i < 5)`
	`  { print(i); i++; }`

3.4 Implementation details for Stage II reductions

The Java programming language supports "break" and "continue" statements. These allow the flow of control to jump around in a structured way.

An example of a `break` statement is shown in figure 15. Note that the execution of the `break` statement causes the flow of control to exit the loop.

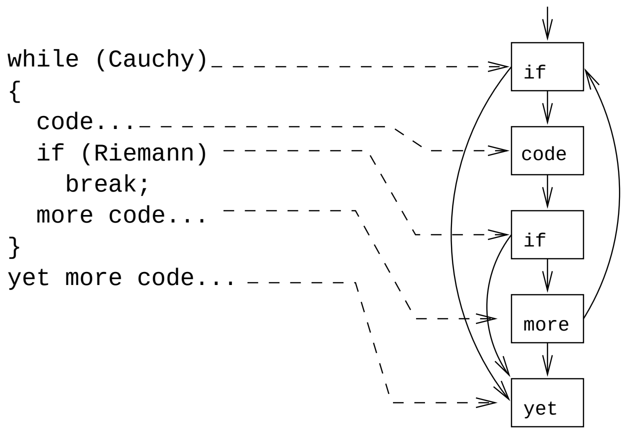


Figure 15: A `break` statement and its associated control-flow.

The reader should now observe that the body of the loop will not be recognized as a basic block – it does not constitute a basic block since one of the inner nodes has multiple successors. This also blocks the reduction of the `while` loop; the loop body does not get reduced, so the `while` loop cannot be reduced.

Hence, once there are no more stage I reductions to be done, and if the graph has not yet been reduced to a single node, we attempt a "Stage II" reduction. These are designed to catch `break` and `continue` statements.

There are two types of Stage II transformations: *cyclic-region* and *switch* transformations. First I describe a cyclic region.

3.4.1 Cyclic-region transformations

Definition 2 *For each node $n \in N$ we associate a* cyclic region $C(n)$. *We define $C(n)$ as the union of all cycles rooted at n.*

Some examples of cyclic regions are shown in figure 16.

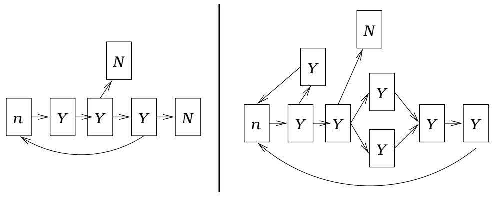


Figure 16: Cyclic regions illustrated.

Given a cyclic region $C(n)$, we can detect `break` as follows: let n have a successor e which is not contained in the cyclic region. Then if $m \in C(n)$ has e as a successor, we declare it to be a `break` node. We also remove the control-flow edge from the graph so that reduction may occur. This type of reduction is illustrated in figure 17.

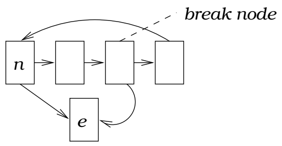
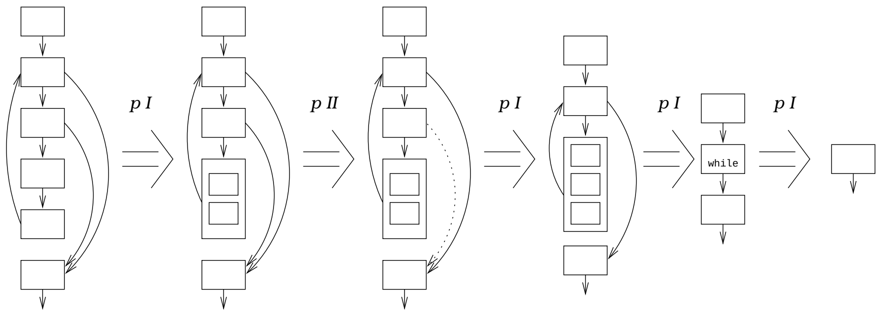
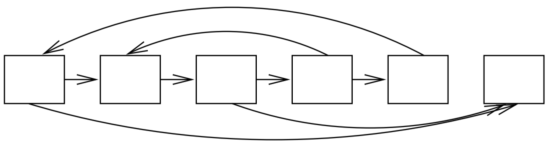


Figure 17: A single *cyclic-region* Stage II transformation.

A complete reduction containing both Stage I and Stage II reductions is shown below. A `while` loop containing a `break` is first reduced using basic-block aggregation, then a Stage II reduction occurs, and the entire loop is then reducible by more Stage I reductions.



Unfortunately, Java supports multi-level `break`, as shown in figure 18. Thus we need to be a little more careful with this transformation. In fact, we label all `break`s with their targets and, in some future cleanup pass, we will match all the `break`s with their targets and ensure that the targets are valid.



```
Q:
while(d)
{
  while(e)
  {
    if (!exists(d))
      break Q;
  }
}
print ("You lose");
```

Figure 18: Example of a multi-level break statement.

3.4.2 Switch transformations

We now reconsider the `switch` statement. Recall that we briefly stated that branches of the `switch` often don't get reduced. In order to correctly identify a `break` statement as such, we need to identify the correct target for the `break`.

Figure 19 contains a diagram of a typical `switch` statement; observe that many different `break` statements can occur in a branch of the `switch`.

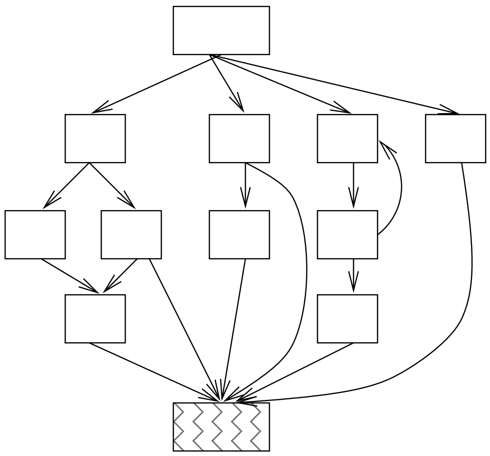


Figure 19: Example of a complicated `switch` statement.

In order to solve our problem, we carefully define what we are seeking:

Definition 3 *The* join node j *is the node nearest to a* `switch` *node* n *such that all acyclic paths from* n *to the end node must pass through* j.

First we label as M the set of nodes directly connected to n. The key observation is as follows: if j is a join node, then when the directions of the control-flow graph are reversed, all the nodes of M must be reachable from j.

Before we start our search, we can trim the search as follows: if b is a reachable node, and a is a successor of b (when edges are reversed), then a must also be reachable, so we can safely remove a from the set M. This is illustrated in figure 20.

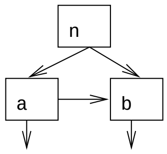


Figure 20: A redundant node.

Now we are ready to execute the search itself. I shall use a depth-first search and a breadth-first search; these are both documented in [CLR90]. Briefly, the idea of a depth-first search is to search all the nodes in one branch before visiting the next branch; the idea of a breadth-first search is to visit all the nodes at depth n before searching depth $n+1$, where n starts at 0.

The basic algorithm is simple: for each node $n \in N$, execute a (backwards) search for all the M-nodes; if we find them all, report success.

There are a few issues left to resolve (and they will guarantee correctness). We need to use breadth-first order while iterating over n, in order to guarantee the nearness property. As for the searches at each n, we use a depth-first search on the reversed edges.

Moreover, we can do better than a brute-force search at each node, at the cost of some storage. As we visit each node q, we store the list of targets it manages to reach, and on a subsequent depth-first search, if we visit q, we add the visited subset $Q \subset M$ to the current node's visited-targets list. This greatly reduces the running time of the search.

4 Possibility and impossibility results

Recall that we would like to decompile arbitrary classfiles. Arbitrary classfiles can be very bad; one can imagine that a classfile might not correspond to any Java code (after all, a classfile has arbitrary goto's and Java code doesn't.)

Arbitrary classfiles cannot be, however, arbitrarily bad: Java code must be verifiable. Verification imposes some constraints on the structure of valid classfiles. The only constraint that affects control-flow structuring is that verifiable classfiles may only contain jumps to valid instructions.

A search of the literature reveals that [PKT73] answers many questions about possibility and impossibility. This paper deals with control-flow graphs and proves a spectrum of results concerning exactly how much power is required to restructure arbitrary graphs.

Consider the following theorem:

Theorem 1 *There exist control-flow graphs that cannot be expressed in terms of* `if` *and* `while` *without increasing their length or changing their execution sequence.*

For the proof of this and subsequent theorems, the reader is referred to the original paper.

Clearly the graph in figure 21 can be expressed if `break` is allowed. However, we can see that `break` is not sufficient:

Theorem 2 *There exist control-flow graphs that cannot be expressed in terms of* `if` *and* `while`*,*

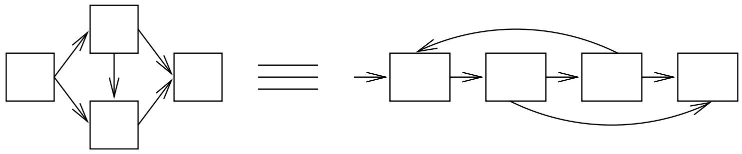


Figure 21: A control-flow graph that requires `break`.

allowing even multi-level `break`*, without increasing length or changing the execution sequence.*

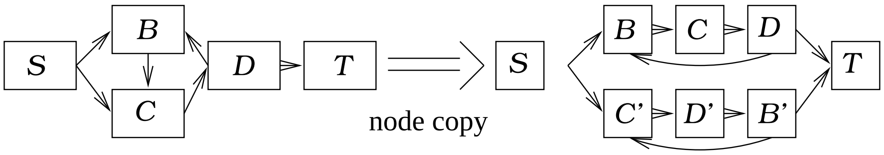


Figure 22: A control-flow graph that requires node copying.

Furthermore, simply copying nodes is not sufficient. (We do not permit new code, nor do we permit extra variables. This result is not true with extra variables allowed.)

Theorem 3 *There exist control-flow graphs that cannot be expressed using* `if`*,* `while` *and single-level* `break`*, even if node copying is allowed.*

It is intuitively clear that the graph in Figure 23, which uses nested loops, makes node-copying with single-level break insufficient.

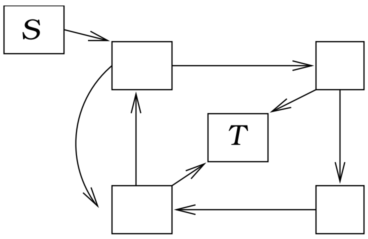


Figure 23: A control-flow graph that requires multi-level break.

The next theorem, however, provides a possibility result, which is very encouraging.

Recall that with every node $n \in N$ is associated a cyclic region $C(n) = \bigcup$ cycles rooted at n. We define a *well-formed control-flow graph* as one where every cyclic region has exactly one entry point.

Theorem 4 *Every control-flow graph can be transformed into an equivalent well-formed graph using node copying.*

Also, we can produce a constructive proof of:

Theorem 5 *It is possible to construct a* `goto`*-less program from a well-formed control-flow graph.*

Hence restructuring is always possible. In Sculpt, we intend to use techniques from [Ero95] rather than from [PKT73], but knowing that restructuring is possible is very helpful when attempting to write this part of the decompiler.

5 Future Work

Much work remains to be done in order to complete Sculpt. At the moment, some of the features which remain to be added are: multi-level `break`, the `continue` statement, and improving support for the `if` statement.

Also, Grimp, the aggregated form of Jimple, which allows complicated expressions, must be combined with the control-flow restructuring.

Finally, phase III of decompilation must be implemented; we must eliminate `goto` for arbitrary control-flow graphs. This will probably be done using work previously done with the Compilers and Concurrency group, described in [Ero95][EH94].

6 Summary

This paper discusses techniques used in the control-flow structuring phase of the Sculpt decompiler. In particular, the three main phases of control-flow structuring were described.

Stage I recognized simple structures in the control-flow graph which could be locally reduced; among these structures were `if-then-else` conditionals, `while` loops, and basic blocks. These reductions require that the substructures involved have been previously reduced to single nodes.

Stage II of the structuring handled some structured impediments to Stage I reductions; in particular, `continue` and `break` become reducible after Stage II transformations. `break` is handled for loop structures as well as `switch`.

Stage III is a final attempt to restructure any remaining irregularities in the control-flow graph that were not previously handled in Stages I and II. Once a control-flow graph requires a Stage III transformation, we have abandoned any attempts at niceness and simply try to give an equivalent Java program: that is, one with the same execution sequence as the original.

Finally, proofs of possibility and impossibility from [PKT73] are quoted. This culminates in the result that, given multi-level break and node copying, it is possible to restructure an arbitrary control-flow graph.

7 Sponsors

The author would like to thank the following organizations for their sponsorship of the McGill contingent to the Canadian Undergraduate Math Conference: CUMC for travel assistance, the McGill Faculty of Science, the McGill Department of Mathematics and Statistics, and the Institut des Sciences Mathematiques.

The author also thanks Laurie Hendren for advice and constructive criticism of earlier drafts of this article.

The work on this project was supported by an NSERC grant.

References

[AG98] Ken Arnold and James Gosling. *The Java Programming Language Second Edition.* Addison-Wesley Longman, 1998.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[EH94] Ana M. Erosa and Laurie 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.

[Ero95] Ana Maria Erosa. A goto-elimination method and its implementation for the McCAT C compiler. Master's thesis, McGill University, January 1995.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, California, 1997.

[PKT73] W. W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat and exit statements. *Communications of the ACM*, pages 503–512, August 1973.

[VRH98] Raja Vallée-Rai and Laurie J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations. Sable Technical Report 4, Sable Research Group, McGill University, July 1998.