NEW ALGORITHMS FOR A JAVA DECOMPILER AND THEIR IMPLEMENTATION IN SOOT

by
Jerome Miecznikowski

School of Computer Science
McGill University, Montreal

February 2003

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Copyright © 2003 by Jerome Miecznikowski

Abstract

This thesis presents Dava, a Java bytecode to Java source code decompiler built on top of the Soot framework.

The Java Virtual Machine Specification of valid bytecode is much less restrictive than the Java Language Specification of valid Java source programs. For example, bytecode has unstructured control flow, loose local typing, and few restrictions on method modifiers. By contrast, the Java language has highly structured control flow, strong local typing, and many restrictions on its method modifiers. The goal of this thesis was to build a tool that could correctly decompile the widest range of verifiable Java bytecode back into compilable Java source. This includes bytecode coming from other source languages, bytecode that has undergone certain types of obfuscation, and optimized bytecode. To accomplish this goal we created the Structure Encapsulation Tree data structure and a set of new decompiling algorithms.

The algorithms fall into three categories: regular control flow, exceptional control flow, and idioms. Regular control flow includes finding constructs such as loops, `if-else` statements, and labeled blocks. Exceptional control flow refers strictly to `try-catch` statements. Idioms is a collection of miscellaneous restructuring enhancements and techniques for ensuring that we produce syntactically correct Java.

The Structure Encapsulation Tree allows us to address various decompiling problems one at a time, in order of their difficulty. For example, we find all loops before we find any `if-else` statements. The result is a very robust algorithm that will restructure any control flow graph *without* resorting to tricks such as encoding the control flow graph as a finite state machine.

We test our implementation of the decompiler on a wide range of inputs and compare the results to the output of four leading Java decompilers. Our results show that Dava always produces correct source code and that it far outperforms the competition on all of our more advanced tests.

Résumé

Cette thèse présente Dava, un décompilateur basé sur le cadre d'applications Soot et générant du code source Java à partir de code objet Java.

Les Spécifications de la Machine Virtuelle de Java sont beaucoup moins restrictives quant à la validité du code objet Java que ne le sont les Spécifications du Language Java pour la validité du programme source. À titre d'exemple, le code objet Java présente un flux de commande astructuré, un système de types minimaliste et peu de restrictions sur les attributs déclarés. À l'inverse, le langage Java présente un flot de contrôle hautement structuré, un système de types diversifié et de nombreuses restrictions sur les attributs déclarés. Mon objectif était de bâtir un outil pouvant décompiler la plus grande variété de code objet Java vérifiable en code source Java compilable. Ceci inclut du code objet provenant d'autres langages source, du code objet ayant subi certains types d'obfuscation, ainsi que du code objet optimisé. Afin d'accomplir cette tâche, j'ai créé l'Arbre d'Encapsulation de Structure (Structure Encapsulation Tree)(une structure de données), ainsi qu'un nouvel ensemble d'algorithmes de décompilation.

Les algorithmes appartiennent à l'une des trois catégories suivantes: flux de commande régulier, flux de commande exceptionnel et idiomes. Le flux de commande régulier inclut la recherche de structures telles que les boucles, les instructions 'if-else' et les blocs avec étiquettes. Le flux de commande exceptionnel fait uniquement référence aux instructions 'try-catch'. Les idiomes sont une collection d'améliorations structurelles et de techniques permettant de garantir la production de code Java correct.

L'Arbre d'Encapsulation de Structure (Structure Encapsulation Tree) nous permet de résoudre, un à la fois, une variété de problèmes relatifs à la décompilation, par ordre de difficulté. Par example, nous trouvons toutes les boucles avant de trouver les instructions 'if-else'. Cette technique résulte en un algorithme très robuste et capable

de restructurer n'importe quel graphe de flux de commande sans recourrir à des trucs tels que l'encodage du graphe sous forme de machine à états finis.

Nous vérifions la validité de notre implémentation sur une large sélection d'entrées et comparons les résultats obtenus à ceux de quatre décompilateurs populaires pour Java. Malgré le fait que Dava donne de moins bons resultats dans sa fa con de traiter le code objet provenant du code source Java, il génère toujours du code source correct et devance, et de loin, la compétition dans tous nos tests les plus avancés.

Acknowledgments

This thesis was made possible by a large number of people who I would like to thank. First, Laurie Hendren, my supervisor, who runs the first rate compiler courses and Sable research group at McGill. Besides introducing the world of compilers in a truly engaging manner, it was by her suggestion that I began to investigate the fruitful area of decompiling. From this study, I would like to thank her especially for the two co-authored published papers, the OOPSLA poster, and the chance to present my research at three different conferences. Lastly I would like to thank her for the research assistantship she gave me, without which it would have been impossible to pursue the master's degree.

Next, thanks must go to the authors of Soot, upon whose work I have built. Raja Vallée-Rai is the chief architect and implementer of Soot and his efforts at creating a simple and powerful object oriented framework are deeply appreciated. Many times, I was pleasantly surprised by the rich set of APIs that he provided. Patrick Lam built Grimp which is a very high level internal representation in Soot that I used almost exclusively. Etienne Gagnon built the typing routines that typed local variables. There are also the many members of the lab who have indirectly helped either by enduring my algorithmic ideas and/or by steering me in right directions. In no particular order they are: John Jorgensen, Marc Berndl, Ondřej Lhoták, Felix Kwok, Feng Qian, Patrice Pominville, and Fabien Deschodt. I would also like to thank Bruno Dufour for his translation of the abstract, and John for his help with editing my papers and parts of this thesis.

Finally I would like to thank my parents who have gave me the moral support and courage, sometimes on a daily basis, to be able to see this project through to completion.

Dedicated to

My Mother, Nadine A. Miecznikowski,

and

My Father, John J. Miecznikowski.

Contents

Abstract ii

Résumé iii

Acknowledgments v

1 Introduction 1

1.1 The Soot framework 2

1.2 Thesis Contributions 4

1.3 Thesis Organization 5

2 Overview: Data Structures and Algorithm 6

2.1 Introduction 6

2.2 Data Structures 7

2.2.1 Grimp 8

2.2.2 Control Flow Graph 11

2.2.3 Structure Encapsulation Tree 13

2.2.4 Abstract Syntax Tree 15

2.2.5 Java Source Code 16

2.3 Algorithm Construction 17

2.4 A simple illustrative example 20

3 Regular Control Flow **25**

3.1 `while`, `while(true)` and `do-while` Loops 26

3.1.1 Single entry point strongly connected components 27

3.1.2 `while` Loops 28

3.1.3 `do-while` Loops 29

3.1.4 `while(true)` Unconditional Loops 31

3.1.5 Multiple entry point components 32

3.1.6 Nested loops 36

3.1.7 Loop Bodies 37

3.1.8 Putting Loops in the SET 38

3.2 DAGs 39

3.2.1 Putting DAGs in the SET 39

3.2.2 A small transform that simplifies design 41

3.2.3 `if` statements 41

3.2.4 `if-else` statements 43

3.2.5 `switch` statements 43

3.3 Labeled Blocks 46

3.3.1 Labeled Breaks and Continues 50

3.3.2 Removing labels 52

4 Exceptions **56**

4.1 Introduction 56

4.2 Exception Table Entry Removal 59

4.3 Exception Preprocessing 63

4.3.1 Versioning on the Control Flow Graph 63

4.3.2 Basic Exception Preprocessing 67

4.3.3 Improving the Versioned Control Flow Graph 72

4.3.4 Putting it all together with Control Flow 76

4.4 Exception Handling 80
4.4.1 Spurious `catch` and `try` Removal 82

5 **Idioms** **84**

5.1 Converting Structured Grimp to Java 84
5.1.1 Simple Statements 84
5.1.2 Converting `invokespecial <init>` to Constructor Calls . . . 85
5.1.3 Converting the `clinit` Method to a `static` Initializer Block . 87
5.1.4 `throws` declarations 89
5.1.5 Throwing `null` 92
5.1.6 Class literals 92
5.2 Readability Transforms Performed on Grimp 93
5.2.1 Aggregated `ifs` 94
5.2.2 Class names 101
5.3 Readability Transforms performed on the SET 102
5.3.1 `synchronized()` blocks 102
5.3.2 `finally` blocks 107
5.3.3 `for` loops 108
5.3.4 `if-else` chains 110
5.3.5 Conditional assignments 111
5.3.6 `if-else / continue` substitutions 111

6 **Testing and Results** **113**

6.1 Introduction 113
6.1.1 Measures 114
6.2 Component Testing 116
6.2.1 Basic Loops 116
6.2.2 Multi-entry point Loops 118

6.2.3 if, if-else, and switch Statements 120
6.2.4 Labeled Blocks, breaks and continue Statements 120
6.2.5 Basic Exception Handling 121
6.2.6 Advanced Exception Handling 122
6.2.7 synchronized Statements 124
6.2.8 Class/Package Names Clashes 124
6.2.9 throws Declarations 125
6.3 Benchmarks Description and Testing 125
6.4 Comparison to other Decompilers 127
6.5 Conclusions 132

7 Conclusions **133**

List of Figures

1.1 A simplified layout of Soot. 3

2.1 Grimp representation of simple method `m()`. 8

2.2 Control Flow Graph of `m()`. 11

2.3 The control flow graph for an `if` statement. 12

2.4 An `if` statement without any "join" point. 13

2.5 Structure Encapsulation Tree of `m()`. 14

2.6 Abstract Syntax Tree of `m()`. 15

2.7 Resulting Java source code of `m()`. 16

2.8 The nine phases of the decompiling algorithm and their control flow categories. 18

2.9 Control Flow Graph of `m()`. 20

2.10 Structure Encapsulation Tree of `m()` after phase 1. 21

2.11 Structure Encapsulation Tree of `m()` after phase 2. 21

2.12 Structure Encapsulation Tree of `m()` after phase 5. 22

2.13 Structure Encapsulation Tree of `m()` after phase 6. 23

2.14 Structure Encapsulation Tree of `m()` after phase 8. 24

3.1 Regular control flow phases. 25

3.2 Pattern used to generate a `while` loop. 29

3.3 Pattern used to generate a `do-while` loop. 30

3.4 An example control flow graph that transforms to a `while(true)` loop, and its corresponding code. 31

3.5 A simple multi-entry point strongly connected component and its conversion to a single entry point strongly connected component. 33

3.6 A complex multi-entry point strongly connected component with two possible transforms. 34

3.7 The generated code from figure 3.6 35

3.8 Code and corresponding control flow graph. 38

3.9 Finding a DAG *bodySet* 40

3.10 Trimming a DAG SET node. 41

3.11 A small transform that guarantees that successors of conditionals are dominated. 42

3.12 A simple `switch` statement. 44

3.13 A `switch` statement with a case fall-through. 44

3.14 A `switch` statement with multiple case fall-throughs. 45

3.15 A `switch` statement that cannot employ case fall-throughs. 45

3.16 A second `switch` statement that cannot use case fall-throughs. 46

3.17 Summary SET from the top view. 47

3.18 Summary SET from the top view with labeled block solution. 48

3.19 An inter-child control flow graph. 49

3.20 Natural exit points from structured statements. 51

4.1 Exception based control flow phases. 56

4.2 Grimp representation of simple method `m()` with exception. 58

4.3 Code showing decompilation of Figure 4.2 59

4.4 A complex interaction of exception table entries. 60

4.5 Abstract Control Flow Graph of Figure 4.4. 61

4.6 Splitting an exception table entry. 62

4.7 An area of protection with two entry points and its appropriate transform. 64

4.8 A self targeting area of protection and its appropriate transform. . . . 65

4.9 Two non-nesting areas of protection and their appropriate transform. 66

4.10 Application algorithm 6 to figure 4.5. 68
4.11 A simple example of branch integration. 73
4.12 An more complex example of branch integration building on figure 4.11. 74
4.13 A simple example of stem integration. 76
4.14 Code corresponding to stem integration in figure 4.13. 77
4.15 A simple example of follower fusing. 78
5.1 Calling one constructor from another in Java and the corresponding Grimp. 86
5.2 An "impossible-to-decompile" constructor. 87
5.3 A class with an illegal static initializer. 88
5.4 An explicit `throw` statement causes a `throws` declaration. 90
5.5 Method invocation may cause a `throws` declaration. 90
5.6 Inheritance may cause a `throws` declaration. 91
5.7 Reduction and simplification of a compound `if`. 94
5.8 An Expression with potential side-effect and their ordering. 96
5.9 Removal of 2 negations from an expression tree. 99
5.10 Unstructured use of monitor instructions. 103
5.11 A simple `synchronized()` block in Java. 103
5.12 The Grimp equivalent of figure 5.11. 104
5.13 The structured code output before `synchronized()` blocks are found. 104
5.14 Exception preprocessing can resolve structuring problems involved with creating `synchronized()` blocks. 106
5.15 Pattern that is searched for to build `finally` blocks. 108
5.16 `for` loop pattern. 109
5.17 Pattern that "fixes" `breaks` in `for` loops. 109
5.18 Conversion of nested `if-else` to `if-else` chain. 110
5.19 A conditional assignment. 111
5.20 Using a `continue` to reduce level of nesting. 111

6.1 Development life cycle for Dava. 114
6.2 Overview of Implementation and Testing. 115
6.3 A basic set of tests for simple loops. 117
6.4 A basic set of `while` in `while` tests. 117
6.5 Control flow graphs for figure 6.4. 118
6.6 Control flow graph of a simple multi-entry point loop. 119
6.7 A "fun" multi-entry point loop. 119
6.8 Example program control flow graph and Dava output. 121
6.9 Decompiled code for method `foo1()` 122
6.10 Control flow graph and decompiled code for method `foo2()` 123
6.11 Description of Core Suite. 126
6.12 Original and decompiled code for method `foo3()` 129
6.13 Decompiled code for method `foo1()` 130
6.14 Decompiled code for method `foo2()` 131

Chapter 1

Introduction

Java is an increasingly popular development language [9] and platform [14]. The Java language has many attractive features, such as a simple multi-threading model, garbage collection, runtime checking, exceptions, and a highly object-oriented design. Applications written in Java are usually compiled with Sun Microsystem's [11, 25] `javac` compiler to a bytecode representation. The bytecode is then run on a Java platform, which is usually a Java virtual machine [14]. The virtual machine's runtime system then has to support the Java language's features.

This support has benefits and drawbacks. On the benefit side, bytecode is portable among many different hardware platforms, common types of memory leaks are eliminated, memory access is always "safe", multi-threading resource locks and unlocks are always balanced, and so on. On the drawback side, supporting these features imposes a runtime overhead. Optimization can reduce this overhead, but if the optimization process itself is done "on-the-fly" as the bytecode application is being run, it too imposes an overhead. Choosing good optimizations to reduce overall runtime can therefore be tricky.

Soot [22,26,27] is a tool written by McGill University's Sable Research Group [21] that can analyze and optimize Java bytecode before it is run on a Java virtual machine. This eliminates some of the overhead of optimization at runtime. However, this is also a useful tool for experimenting with new, radical types of optimizations and more general transforms, including bytecode obfuscation and watermarking. As a general purpose tool, then, Soot has many applications.

One of the aspects of Soot that makes it effective, is that it is relatively easy to use. Debugging one's transforms can be done by dumping the internal representation

of the code being transformed to a file and inspecting the resulting "snapshot" of your transform at work. Decompiling, then, is a logical continuation of this feature, such that one can now view what one's transform does in a pure Java sourced representation of the bytecode.

Once we were committed to the project of decompiling with Soot, however, we felt that there were other worthwhile goals to be pursued as well. There are quite a few Java decompilers already available, but very little published information as to how they worked. The Soot decompiler, then, is an exploratory work in which we hope to cover most of the important issues of decompiling Java. With this goal in mind, we expanded the scope of this project to trying to state what is and is not possible in decompiling Java and to provide high quality solutions. Along the way, we discovered numerous differences between the Java language specification and the Java virtual machine specifications which have limited our result [16]. However, because we have taken a comprehensive and aggressive approach, we have developed a surprisingly powerful decompiler.

In the rest of this introduction, we take a look at the Soot framework, clarify the contributions of our research, and review the organization of the rest of the thesis.

1.1 The Soot framework

Soot is a Java bytecode transformation and annotation framework. There are two ways to view Soot, first as a *user* tool, and second, as a *compiler writer's* tool. As a user tool, Soot will read in a Java class file, perform some optimizations on it and emit a new optimized class file. However, Soot has been built with the goal of making it easy for compilers developers to add and test new optimizations. As a compiler writer tool, then, it provides a rich API and set of internal representations to develop upon. A simplified layout of Soot is given in figure 1.1 (page 3).

Soot has three main parts, Baf, Jimple [28] and Grimp. Each part is is made up of a processing phase, an internal representation, and an associated API. For example in Baf, the processing phase converts a class file into an internal representation called *baf* and supplies an API for performing transforms on this internal representation (IR).

Briefly, baf is a stack based representation that closely resembles the disassembly of a class file. Its main benefit is that it allows one to directly manipulate the stack

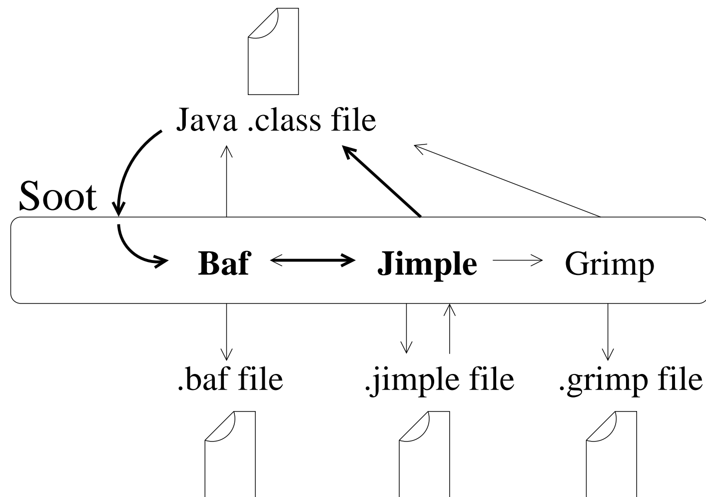


Figure 1.1: A simplified layout of Soot.

code that will be eventually translated into a class, without having to deal with the complications of Java bytecode. It is useful for operations such as peep-hole optimization.

Jimple is a stackless 3-address IR with typed locals. This is a powerful and convenient form suitable for high level optimizations such as copy-propagation or array bounds checking. This is the form which most Soot users and developers work with. A nice feature of Jimple is that its IR can be saved to a text file and reloaded at a later time. This allows one to write Jimple files directly in the Jimple IR and to assemble them with Soot into Java class files. Most people who use Soot as a user tool, however, will just be loading the class through Baf into Jimple where it is optimized and emitted back as a Java class file.

Grimp is an extension to Jimple. Its internal representation is identical to Jimple's except that it aggregates expressions across straight line code. It is noteworthy, however, that Grimp does not aggregate expressions across conditional statements. We use the Grimp IR as the starting point for Soot's decompiler.

1.2 Thesis Contributions

This thesis is concerned with extending Soot to allow it to function as a Java decompiler. The 3-address code of Jimple provides us with typed simple statements, and the expression aggregation done in Grimp makes the grimp IR look very much like unstructured Java source code. We have written a part of Soot called Dava, which has its own IR (dava), API, and conversion phase. Dava converts the Grimp IR into the Dava IR, which when printed to a text file, is recompilable Java source code.

There are two problems addressed in the Dava phase, first structuring Grimp, and second, modifying the structured Grimp into Java source.

The more difficult issue of the two is restructuring Grimp and is therefore the focus of this thesis. To restructure, we build a control flow graph (CFG) and then, using graph theoretic techniques from compiler optimization design, detect properties in the CFG that allow me to find a structured representation. Because we avoided simple pattern matching in favor of more general techniques, Dava is able to restructure *any* control flow graph. This is an interesting accomplishment because it is able to handle highly unstructured control flow without ever having to resort to tricks such as simulating control flow with a state machine.

There are three new contributions in our techniques.

1. We developed a new data structure called the Structure Encapsulation Tree (SET). This data structure is important because it allows us to find control flow statements in new ordered way based on their individual semantic properties rather than their locations relative to other control flow statements.

2. We built a collection of new structuring algorithms that work on an SET which can translate unstructured Grimp into structured Grimp. These algorithms work with both arbitrary regular control flow and arbitrary exceptional control flow.

3. As a sub-problem, we realized that any acyclic control flow graph, even if it includes arbitrary exceptional control flow, can be represented in pure Java. This allowed us to develop a new technique for building labeled blocks and using labeled `break` and `continue` statements for structuring complex control flow.

The remaining problem is to convert the structured Grimp into Java. Our ambitions were a little lower here, as there are many prohibitive rules in the Java language specification for which there is no counterpart in the Java bytecode specifications. For example, there is a long list of restrictions that are applied to `static` initialization blocks in Java source, but very few for the corresponding Java bytecode. For this reason, it is possible to "foil" Dava, but only in what could be called the "Javafication" of structured Grimp.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 gives a brief overview of the data structures and how the decompiling algorithm is put together. Then, the next three chapters present different parts of the algorithm. Chapter 3 examines non-exceptional control flow. This is broken into three sections, loops, DAGs, and labeled blocks. Chapter 4 looks at exceptional control flow, and chapter 5 examines a wide range of Javafication and idiomatic issues. In chapter 6 we test Dava and review the results. Finally, chapter 7 presents our conclusions on the Dava decompiler.

Chapter 2

Overview: Data Structures and Algorithm

2.1 Introduction

This thesis presents an algorithm that is used for decompiling Java bytecode into Java source code. The main goals of the algorithm are as follows.

1. Correctness. We should not produce incorrect output. In the rare cases when we cannot produce an output, we simply give up and send an error message to the user.

2. Understandable output. This requirement is broken into two sub-goals.

   (a) Efficient representation. For any given piece of control flow there may be many possible structured representations. For example, a `while` loop could equally be represented by an `if` and `do-while` statement combination. Our intuition is that a simple concise representation of control flow should be easier to understand than a more complex one.

   (b) Programmer idiom recreation. If there is an obvious pattern in the compiled code which corresponds to a programmer idiom and does not interfere with efficient representation, we will attempt to rebuild the idiom. An example is the construction of `for` loops out of `while` loops with initialization and incrementalization.

3. Speed. Once the first two criteria are satisfied, we make an effort to keep the restructuring algorithms reasonably fast. We judge this to be adequate as long as the decompilation phase runs in under five seconds for a method. Note that in the Java Virtual Machine Specification there is a limit of up to 65536 bytes per method, and therefore a limit of 65536 decompilable statements for our decompiler to produce.

This algorithm is very large and the parts are highly interdependent. To directly address the whole functioning together is to lose sight of the guiding design in a flood of details. This chapter presents an overview which walks through the data structures, briefly talks about what the parts of the algorithm are and how they are put together. Then afterwards, in chapters 3 through 5, we go back and examine each major issue on its own.

2.2 Data Structures

There are five data structures that we use to represent a Java method while decompiling.

1. Grimp representation
2. Control Flow Graph (CFG)
3. Structure Encapsulation Tree (SET)
4. Abstract Syntax Tree (AST)
5. Java source code

The Soot framework processes Java bytecode and produces Grimp, which is our starting point. The contribution of this thesis is to convert Grimp to a CFG, to build a SET from the CFG, and finally to produce an AST from the SET. A recompilable Java method is then emitted as source code from a pretty printer traversal of the AST.

To help understand how the decompiler works, we show, starting in section 2.2.1, an example method beginning in its Grimp representation, and how it gets transformed through each of these different representations.

2.2.1 Grimp

Label	Grimp Code
	`public void m(int )`
	`{`
	`    dso_1 r0;`
	`    int i0, $i1;`
	`    java.lang.RuntimeException r2, $r3;`
a	`    r0 := @this;`
b	`    i0 := @parameter0;`
	`    goto label3;`
	`label0:`
c	`    $i1 = i0;`
d	`    i0 = i0 - 1;`
e	`    java.lang.System.out.println($i1);`
	`label1:`
	`    goto label3;`
	`label2:`
f	`    $r3 := @caughtexception;`
g	`    r2 = $r3;`
h	`    java.lang.System.out.println("RuntimeException caught.");`
	`label3:`
j	`    if i0 > 0 goto label0;`
k	`    return;`
	`    catch java.lang.RuntimeException from label0 to label1 with label2;`
	`}`

Figure 2.1: Grimp representation of simple method `m()`.

Grimp is a high level unstructured intermediate representation suitable for traditional compiler optimizations. Figure 2.1 shows an example method `m()` in the Grimp format. 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.

Note that in figure 2.1 all program statements except simple `gotos` have been labeled with lower case letters. These letter labels are not part of Grimp, but have been added to aid with our later presentation of the CFG and SET representations.

Although Grimp was originally designed as part of a bytecode optimizer [27], 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 have been aggregated, and all variables have declarations with appropriate types [8]. 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. It is just the directives for control flow and exception handling that are unstructured.

Our first task is to restructure the control flow and exception handling and build a structured Grimp representation. Unfortunately, structured Grimp is not equivalent to Java source code. Some of the additional differences that we address are:

1. Although variables have been typed, integral and boolean constants have not. The `boolean`, `char`, `byte`, and `short` types are still all treated as `int`.

2. Method declarations reflect the looser constraints imposed by the Java Virtual Machine Specification rather than the stricter ones from the Java Language Specification. Two issues of note are that (1) in bytecode a method does not need to declare a `throws` clause for an uncaught exception, while in the Java language it must, and (2) the `throws` clauses must agree between classes and their sub-classes in the Java language, while it is not necessary in bytecode.

3. In bytecode, new objects are created in two steps. First, an empty object is created with a `new` instruction, and second, the object is initialized with a call to the class's `<init>` method. In the Java language, there is simply one call to the `new()` class instance creation expression. While the two matching bytecode instructions are usually folded into a single `new()` in Grimp, the fold is not performed if the two instructions are separated by intervening code. A more aggressive `new()` folder is needed for decompilation.

4. In Grimp, there may be multiple calls to the current or super-classes `<init>` method within a constructor. In Java there may only be one call to the corresponding `this()` or `super()` initialization expression per constructor.

5. Classes and packages may have the same name in Grimp since all `invoke` statements explicitly state the package, class and method names. The Java language, however, allows for implicit package use and disallows package/class name clashing.

6. Method parameters are assigned from special variables within the body of a Grimp method. In our example method `m()` in figure 2.1 (page 8) the integer parameter that `m()` receives is loaded into local `i0` in the statement `i0 := @parameter0;`. In Java, however, the local *is* the parameter which is implicitly assigned to in the invocation of `m()`.

7. Exception parameters are also assigned from special variables within the body of the Grimp method. Again, in our example method `m()` in figure 2.1 (page 8) the exception handler at `label2` assigns a reference to a caught exception object to a local reference: `$r3 := @caughtexception;`. In Java, they are implicitly assign to in the declaration of the `catch()` clause as shown in figure 2.7 (page 16), statement **f**.

8. The syntax of `<invoke>`, negation, length, and the various `cmp` expressions may need to be converted to conform to the Java language specification.

9. The various constraints on static initializers within the Java bytecode are much looser than those specified in the Java language specification. Two important issues are (1) while there may be several exit points within the bytecode (`return void` instructions), there must only be one in the language version, and (2) while the bytecode may contain arbitrary normal control flow, the language version is intended to be a straight line of assignments.

These are just a few of the issues that will need to be addressed for a complete transform between structured Grimp and Java. However, our results will show that addressing all differences is neither possible nor a practical necessity. We support this claim by comparing our results with the results of several leading free and commercial Java decompilers in chapter 6 and see that our approach is both sufficient and more thorough than other leading decompilers.

2.2.2 Control Flow Graph

The Grimp representation is a list of simple statements, so our first step in structuring is to build a control flow graph (CFG) representing potential control flow between statements. The CFG is a directed graph where each node wraps a single Grimp statement with predecessor, successor, dominator, and reachability information.

Definition 1 (Predecessor) *Given some program point* p*, a predecessor of* p *is any statement* q *that may have been immediately executed prior to the execution of* p.

Definition 2 (Successor) *Given some program point* p*, a successor of* p *is any statement* r *that may be immediately executed subsequent to the execution of* p.

Definition 3 (Dominators) *Given a program control flow graph* G *with entry point* s*, for program point* p*, a dominator for* p *is any program point* q*, such that if we start at* s *we must pass* q *to reach* p.

Definition 4 (Reachability) *Given program points* p *and* q*,* q *is reachable by* p *if there exists some path from* p *to* q.

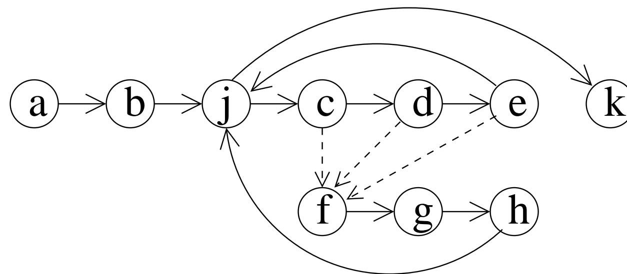


Figure 2.2: Control Flow Graph of `m()`.

Figure 2.2 shows the CFG for our example method `m()`. The labeled nodes correspond to the label letters from figure 2.1 (page 8). There are two types of control flow edges in this graph, those that correspond to regular control flow (solid arrows) and those that correspond to control flow generated by the throwing of an exception

(dotted arrows). 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, such as the successors to a loop condition, may consider only regular paths of execution.

There are two interesting features that all methods will produce in their CFG representations.

1. There will be exactly one entry point to the method's control flow.

2. The graph need not be reducible.

The chief consequence of point 2 is that a reduction based approach to decompiling is insufficient. Compiling is performed by using grammar *productions* and it is intuitively appealing that decompiling should simply be a set of *reductions* on a CFG, perhaps making slight graph transformations where necessary. To understand the difficulties a reduction based approach presents, and why the next data structure we introduce (the SET) is necessary, we must take a brief look at how reduction works.

Given an unstructured digraph G representing the control flow of a program, G is scanned for subgraphs that are isomorphic to the control flow graphs of the target language's compositional constructs. For example, the graph of a simple `if` would seem to be well defined, figure 2.3.

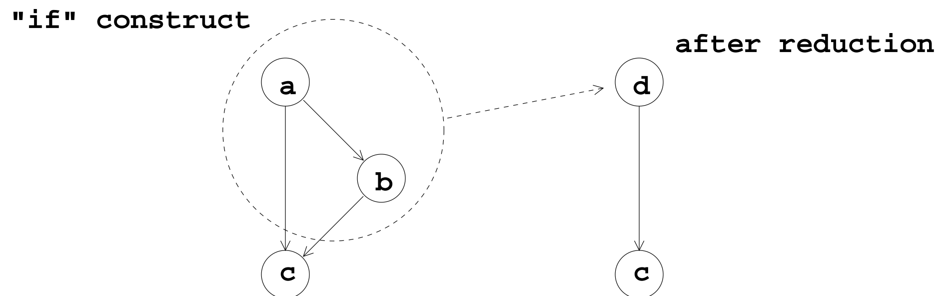


Figure 2.3: The control flow graph for an `if` statement.

If, while scanning G, we ever find a graph that is isomorphic to the left-hand side of figure 2.3 (page 12) we can *reduce* it to the right hand side. Unfortunately, an `if`

```
if (a)
{
    b
    exit program
}
c
```

Figure 2.4: An `if` statement without any "join" point.

may take many other forms. For example, the control flow edge from `b` to `c` need not exist, as seen in figure 2.4.

The subgraph for a compositional `if` statement, then, is not isomorphic to any one graph, but rather one of potentially a very large set of graphs. Also, not all conditionals should be represented by `if` statements, some may better represented as conditions on loops. Choosing to match or reject a pattern is not straight forward. Practically speaking, we need more information than a simple pattern match to decide which `ifs` match, and when it is appropriate to perform the reduction. A *pure* reduction based restructurer, then, is not practical.

It turns out, however, that there is enough extra information available in the CFG, that a reduction based restructuring is not necessary at all. The Structure Encapsulation Tree is built from the CFG and represents just such an accumulation of extra information.

2.2.3 Structure Encapsulation Tree

Figure 2.5 shows the Structure Encapsulation Tree (SET) for our example method `m()`. The important feature of the SET is that each node of the tree contains a set CFG nodes. These sets of CFG nodes determine where any particular SET node fits in the tree.

The SET nodes themselves, represent structured Java constructs, such as `if-else` and `while` statements, and present information about the constructs' characteristics. For example, all constructs have some type of body, so every SET node has the set of references to the CFG nodes that make up its body. Nodes corresponding to conditional statements must also carry a reference to the necessary conditional expression.

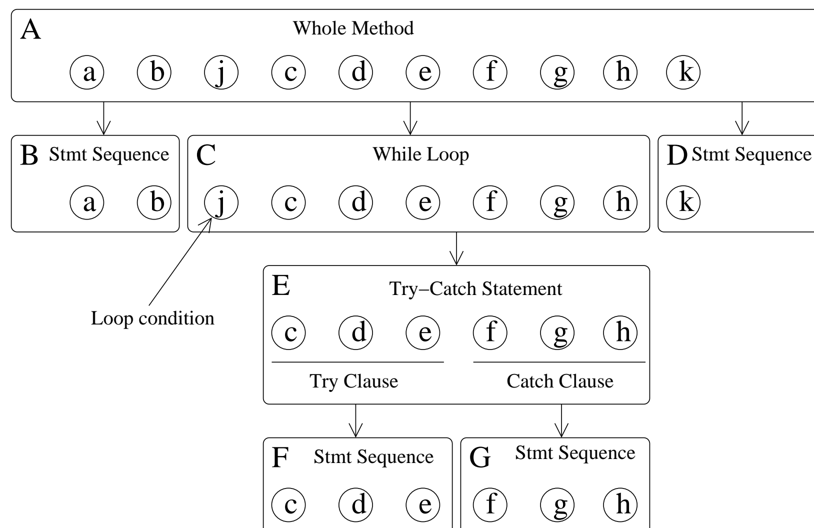


Figure 2.5: Structure Encapsulation Tree of `m()`.

As with the Grimp representation, we have added extra labels to the new representation. This time the upper case letters have been added to the upper left corner of each node.

Definition 5 (Body) *A body is the set of program statements* S *from a control flow graph* G *that are used to position (nest) a control flow construct* C *within an SET. When looking at Java source, the body is only the set of statements within the curly braces of a statement.*

Edges in the SET represent the strict subset relations between the body sets of each SET node. Assume we have nodes A and B that respectively contain body sets x and y. If $x \supset y$, then A will be an ancestor of B. This property is strictly enforced.

The second important property is that SET sibling nodes must have disjoint body sets. In figure 2.5 the body sets of B, C, and D do not share any of their Grimp statements. The reason for this is that SET nodes are used to directly generate an abstract syntax tree which represents the structured control flow statements in Java.

The final key feature of the SET is that it is built in whatever order is suitable for the algorithm. We begin with just the "Whole Method" node and insert all other nodes one by one. Because the subset relation is transitive, there is no restriction on

the order in which we can add these nodes. For instance, in figure 2.5, a valid order of insertion could have been `A E C B F G D`. We will show in section 2.3 why this property is so useful.

2.2.4 Abstract Syntax Tree

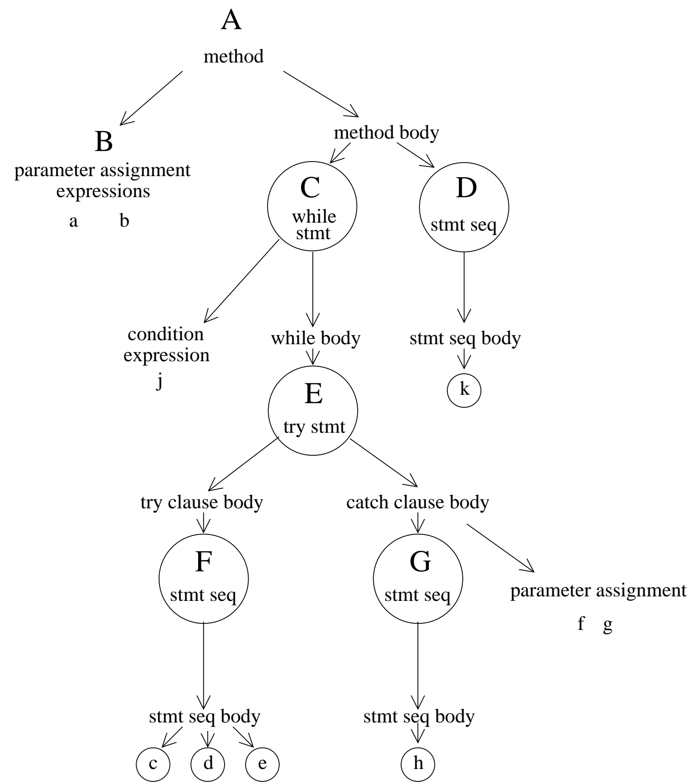


Figure 2.6: Abstract Syntax Tree of `m()`.

The Abstract Syntax Tree (AST) is generated by a single pass over the SET. Since Grimp simple statements can house structured expressions, the AST need only express structure down to the level of the Grimp statement and use the Grimp representation to express structure at the sub-statement level.

Each Dava statement in the AST may, then, optionally house a Grimp expression and a list of other statements. The list is called the statement's *body*. Note that this is a subset of the corresponding SET node's body.

Figure 2.6 (page 15) illustrates the AST resulting from our SET of method `m()` in figure 2.5 (page 14). Circles represent statements; large ones represent Dava's control flow based statements, small ones represent Grimp's simple statements. Note that the condition expression for the `while` loop and the parameter assignment expressions for the method have been stripped from their Grimp statements and are directly held by their appropriate grammar productions.

2.2.5 Java Source Code

Grimp Stmt	SET Node	Decompiled Java Code
a,b	**A,B**	`public void m(int i0)`
		`{`
		`   int $i1;`
j	**C**	`   while (i0 > 0)`
		`   {`
	E	`      try`
	F	`      {`
c		`         $i1 = i0;`
d		`         i0 = i0 - 1;`
e		`         System.out.println($i1);`
		`      }`
f,g		`      catch (RuntimeException $r3)`
	G	`      {`
h		`         System.out.println("RuntimeException caught");`
		`      }`
		`   }`
k	**D**	`   return;`
		`}`

Figure 2.7: Resulting Java source code of `m()`.

The final representation for a method is the resulting Java source code. This code should be recompilable with Sun's `javac` Java compiler. Two issues we take care of in the output of the source code are:

1. When compiling a class which contains the use of a class literal, all current versions of `javac` will generate a method called `class$`. This means that the

source code already containing a `class$` method will cause a compile time error, as the special method name is no longer free for the compiler to generate. This is a bug in `javac` but should be accommodated to generate classes that are recompilable today.

2. To promote readability, `import` statements and implicit package dereferencing should be used whenever possible. Explicit package dereferencing should only be used as a last resort when resolving class naming conflicts.

2.3 Algorithm Construction

We turn now from a data structure centric point of view to an algorithmic one. Here we have three main issues: regular control flow, exceptional control flow and idioms. Regular control flow refers to a *minimal* set of Java control flow constructs: loops, `if`s and `switch` statements and does not include the `try`, `catch` or `finally` constructs. Exceptional control flow is made of `try` and `catch` statements. Idioms is everything else we do, including finding `synchronized` blocks, `finally` blocks, `for` statements (which are a syntactic sugaring of `while`), and so on.

Although we have three simple categories for explaining issues, the actual construction of the decompiler does not follow these nice divisions. The decompiler implementation is in fact made up of nine major phases.

1.	Exception Preprocessing	section 4.2, 4.3
2.	`while` and `do-while` statement restructuring .	section 3.1
3.	`if` statement restructuring	section 3.2.3, 3.2.4
4.	`switch` statement restructuring	section 3.2.5
5.	`try-catch` statement creation	section 4.4
6.	Statement sequences	section 3.2
7.	Labeled block creation	section 3.3
8.	`break` and `continue` control flow identification	section 3.3.1
9.	`synchronized()` block simplification	section 5.3.1

The reason that these issues are split up into small parts and, in effect, inter-mixed with each other is that we found specific design and implementation advantages in the above ordering. Figure 2.8 shows how these nine phases fit into the three main issues.

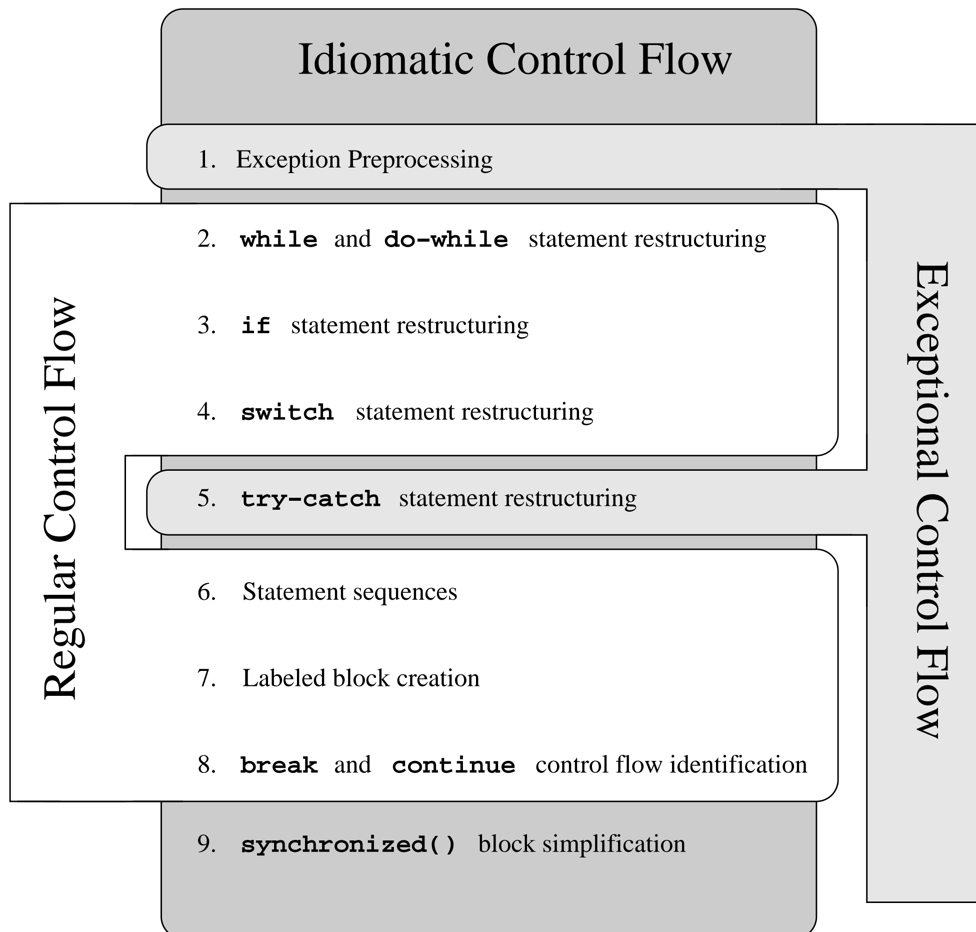


Figure 2.8: The nine phases of the decompiling algorithm and their control flow categories.

At the start of decompiling, we are given a `GrimpBody` which holds a list of `Grimp` program statements. These phases then incrementally build the SET in a specially ordered way so that each phase supports and builds on the others. For example, loop restructuring provides vital information for the `if` statement restructuring phase.

The first action is to create a CFG from the `Grimp` statements. Since all branches in `Grimp` are direct, building the control flow graph is merely a matter of traversing the list, and for each statement noting which statements are its successors.

Phase 1, exception preprocessing, checks the control flow graph to make sure that

certain impossibilities do not exist for the construction of Java `try` statements. If they do exist the graph is transformed to a close-to semantically equivalent form (we do not change the program behavior) where the impossibilities have been eliminated.

At this point we start building the Structure Encapsulation Tree (SET) by creating a SET node that represents the whole method and houses every statement in the control flow graph.

Phase 2, `while` and `do-while` statement restructuring, looks for strongly connected components (SCC) [4] in the control flow graph. For each SCC it finds, it creates a SET node that correspond to a `while`, `do-while`, or `while(true)` statement that best represents the SCC, and embeds it in the SET. Nested loops are found by strategically removing certain statements from the SCC and performing a re-evaluation.

Phase 3, `if` statement restructuring, looks for the conditionals that have not been used in the creation of looping SET nodes, and builds `if` SET nodes for each of these remaining conditionals. The new node is then embedded in the SET.

Phase 4, `switch` statement restructuring, finds `Grimp` switch statements in the control flow graph and building a SET node for each. The new node is then embedded in the SET.

Phase 5, `try-catch` statement creation, runs through the set of exception candidates building SET nodes for each and seeing if the new SET nodes can nest properly in the SET. If they nest, they are inserted into the SET. Otherwise, the control graph is modified slightly to allow the nesting and the algorithm goes back to phase 2.

By phase 6 we have identified all SET nodes that employ some type of conditional or exception generated control flow. At this point we can examine each SET node and determine where we need to establish statement sequences. By construction, the new statement sequences will be leaves in the SET.

In phase 7, labeled block creation, we form directed acyclic graphs of our SET nodes and topologically sort them [4]. From this sorting we can determine if we need to create any labeled block to resolve any issues of non-exceptional control flow. If so, we create the appropriate SET nodes for the necessary labeled blocks and nest them in the SET.

Phase 8, `break` and `continue` control flow identification, examines the control flow edges between the members of the topological sorting and determines if the control flow needs to represented with the addition of `break` and `continue` statements.

If it is found to be necessary, this phase will then append these statements into their appropriate positions in the already existing SET nodes representing statement sequences.

Phase 9, `synchronized()` block simplification, scans the SET for a usage pattern of `monitor` statements coupled with the presence of `try-catch` statements that cover the `java.lang.Throwable` exception type. If this pattern is found in the SET, it is replaced with a SET node representing a `synchronized()` block.

Finally, we traverse the completed SET and emit an abstract syntax tree (AST). The AST presents a simplified view of the structured method and conforms to the necessary `Soot` specifications for emitting a printout. The printout from the AST is pure Java source.

2.4 A simple illustrative example

In section 2.2 (Data Structures) we saw an example method `m()` in each of its representations. We will now walk through each of the phases from section 2.3 (Algorithm Construction) and see how it builds the SET for `m()`.

We begin by taking the Grimp representation shown in figure 2.1 (page 8) and building a control flow graph representing both the regular and exceptional control flow.

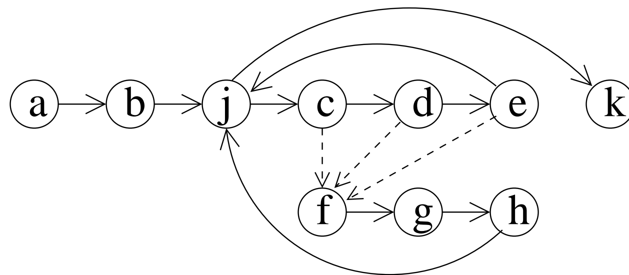


Figure 2.9: Control Flow Graph of `m()`.

Phase 1, exception preprocessing, has nothing to do to the control flow graph since `m()` represents already structured Java. This is explored in sections 4.2 and 4.3.

At the end of phase 1 we build the SET. The first form of the SET contains only one node which represents all the statements from the control flow graph.

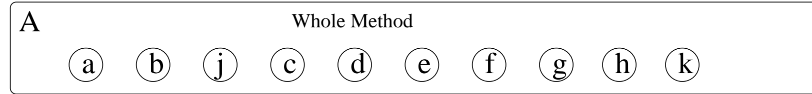


Figure 2.10: Structure Encapsulation Tree of `m()` after phase 1.

Phase 2, `while` and `do-while` statement restructuring, finds the SCCs in the control flow graph of `m()` and builds a SET node for each SCC and inserts it into the SET. These SET nodes are then inspected to determine what type of loop best represents them.

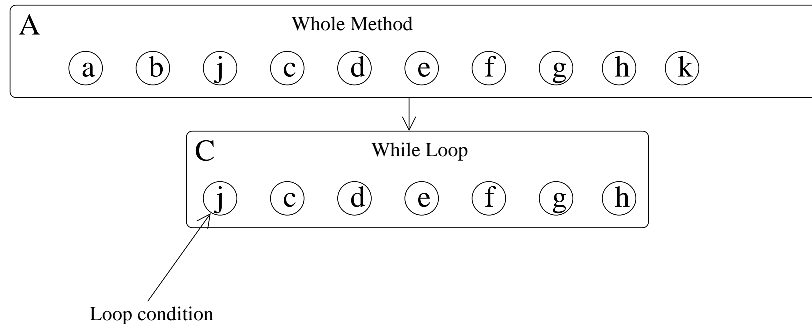


Figure 2.11: Structure Encapsulation Tree of `m()` after phase 2.

Phase 3, `if` statement restructuring, examines the conditional control flow statements in the CFG that have not yet been marked as the conditions controlling loops. Since there are no such instances in `m()` this phase does not put any new nodes in the SET.

Phase 4, `switch` statement restructuring, will build SET nodes for each switch statement found in the CFG. Again, because there are no `switch` statements in `m()`, this phase does not add to the SET.

Phase 5, `try-catch` statement creation, looks at the exceptional edges in the CFG, builds a set of `try-catch` SET nodes and inserts them into the SET. In `m()` we

find that one `try-catch` SET node can represent all exceptional control flow. Once built, the exceptional SET node is inspected to determine the `try` and `catch` clauses.

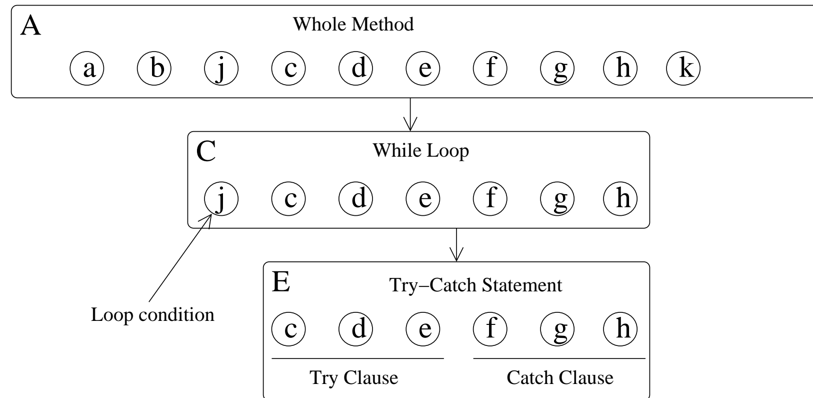


Figure 2.12: Structure Encapsulation Tree of `m()` after phase 5.

Phase 6 finds all the statement sequences in `m()` by examining the nodes in the SET. For any SET node, if it contains CFG statements which are *not* contained in a child SET node (for example `a` in SET node A), then those CFG statements are put in statement sequences. The result from phase 6 is shown in figure 2.13 (page 23).

In phase 7, we create labeled blocks and insert them into the SET. For clarity, our example `m()` does not contain any labeled blocks, but this issues is thoroughly explored in section 3.3.

Phase 8, `break` and `continue` control flow identification, examines *inter*-SET node control flow where control flow does *not* target an ancestor SET node. Figure 2.14 (page 24) shows us this control flow for `m()`. We explore how these edges are examined in section 3.3.1 and will find that no `break` or `continue` statements are necessary in `m()`.

Phase 9 identifies and restructures `synchronized()` blocks. Again, none are found in `m()`, but our techniques for this are explained in section 5.3.1. At this point we have finished building the SET and have identified which control flow edges (if any) should be represented with `break` or `continue` statements. The AST is build with a single traversal of the SET, and source code is emitted as pretty printing of the AST.

The focus of this thesis is not how the implemented parts fit together, but how

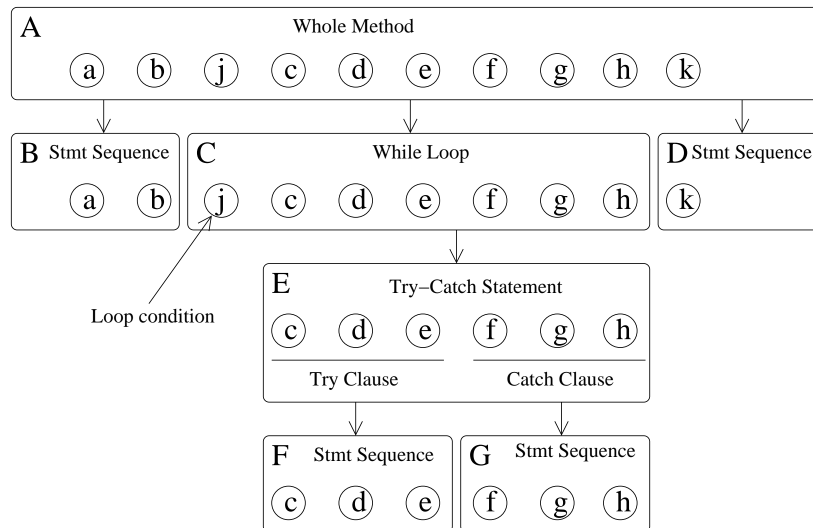


Figure 2.13: Structure Encapsulation Tree of `m()` after phase 6.

each conceptual issue in restructuring is solved. While the nine phases of the implementation interleave the three categories of issues for practical reason, we will examine the decompiler by looking at each issue (for example, exceptional control flow) on its own. In this way, we will try to keep a clear view of the strategy behind the decompiler without getting lost in a maze of details.

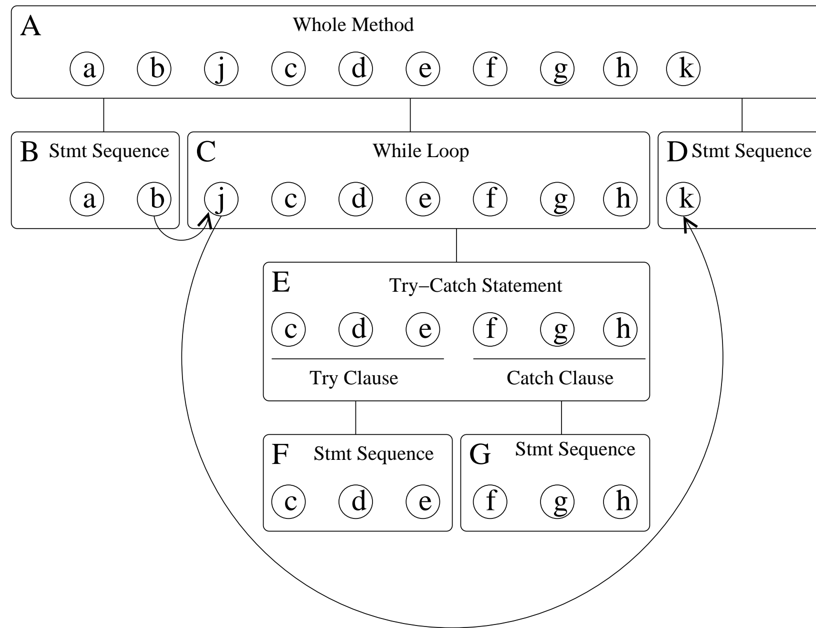


Figure 2.14: Structure Encapsulation Tree of `m()` after phase 8.

Chapter 3

Regular Control Flow

Regular control flow forms the core of Dava's decompiling algorithms. This includes `while` and `do-while` loops, `if` and `if-else` statements, `switch` statements, and labeled blocks, `break`, and `continue` statements. These form six of the nine phases of our algorithm.

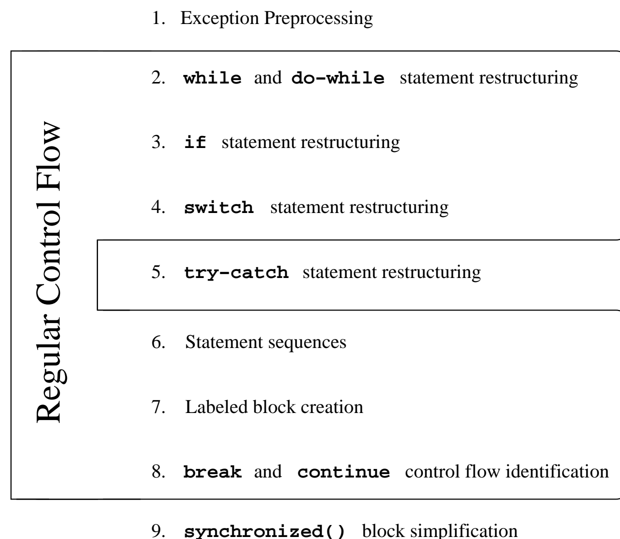


Figure 3.1: Regular control flow phases.

In this chapter, we present regular control in 3 main sections.

3.1 Loops. We find strongly connected components in section 3.1, resolve which

types of Java loops best represent them, and determine which conditional statements should be used to control the iteration of these loops.

3.2 DAGs. Once all loops have been resolved and inserted into the SET, we can determine which conditional branches are not being used as conditions on loops. In sections 3.2.3 through 3.2.5 we determine how to represent these remaining conditionals with `if`, `if-else` and `switch` statements.

3.3 Labeled Blocks. Finally, we look for labeled blocks in section 3.3 and put in any necessary `break` and `continue` statements in section 3.3.1.

Note that since the creation of statement sequences is trivial and we do not devote a section to examining it. A working intuition is given in the overview in section 2.4.

3.1 `while`, `while(true)` and `do-while` Loops

Our first task is to find Java loops in the control flow graph. Every strongly connected component [4] in a control flow digraph G must contain at least one back-edge. The *only* way we can represent back-edges from G in the Java language is with a `while`, `for` (which is a syntactic sugaring of `while`) `while(true)` or `do-while` loop. Because there is no transform, apart from introducing recursive methods, that can be done on the graph to eliminate back-edges, we must ensure that we can represent strongly connected components regardless of any other Java language constructs we build from G. For this reason, Java loops are the first construct we build and place in the SET. Later constructs will then be modified to accommodate already built loops.

A fundamental problem is choosing which cycles in G will contribute to which Java loops in the final representation. A common approach is to look for smallest cycles with an entry point that is the tail of some back-edge [1, 3, 12, 29]. This cycle is then reduced to a single node in the control flow graph and the process is reiterated. Because there is no proven direct correlation between smallest cycles in the control flow graph and most-deeply nested Java loops, this approach can yield bizarre restructurings.

The approach shown in this section (3.1) is to crudely restructure every strongly connected component, and then refine the restructuring with successive passes of the algorithm. To begin with, we simply create one Java loop per strongly connected component. Once created, we use the properties of the Java grammar associated

with the loop to *remove* certain statements from G, producing a new graph G'. The removal of these key statements from G is designed to alter the strongly connected components in G' in such a way that they can correctly be represented as Java loops that nest in the ones that came from G. This process is repeated until we arrive at some G^n which contains no strongly connected components at all. At that point we claim that we have found all the loops and their proper nestings for the method being restructured.

This approach cannot directly deal with strongly connected components that have more than one entry point, since no Java loop has more than one entry point. However, no direct restructuring approach can work because there always must be some cycle in such a strongly connected component that has more than one entry point. A direct Java representation of this cycle is impossible.

In Dava, we transform any multi-entry point strongly connected components into single entry point strongly connected components by creating a dispatch statement that can direct control flow to any of the multiple entry points. Key branches of control flow are then directed to this dispatch statement such that it now acts as the single entry point to the strongly connected component. Section 3.1.5 (page 32) gives a complete description of this process.

The rest of this section is organized as follows. First, in section 3.1.1 we lay out some definitions for simple single entry point strongly connected components. Section 3.1.2 explores when we should represent the SCC with a `while` loop, section 3.1.3 examines when we should use a `do-while` loop, and section 3.1.4 presents the `while(true)` loop as an effective fall-back mechanism when the previous two loop types cannot effectively represent the SCC. Then, in sections 3.1.5 and 3.1.6, respectively, we generalize our scheme to handle multi-entry point loops and nested loops. Finally in sections 3.1.7 and 3.1.8 we explore how to refine what the contents of the Java loop should be and how their corresponding SET nodes should be put in the SET.

3.1.1 Single entry point strongly connected components

We begin by assuming that every strongly connected component has exactly one entry point, and generalize later. Given some strongly connected component, we initially create a Java loop of unknown type, that is, without determining whether it is a `while`, `while(true)`, or `do-while` loop. Specifically, we record:

- The *entryPoint* to the strongly connected component. This is the member of the strongly connected component that has a predecessor which is not a member of the strongly connected component.

- The set of exit points of the strongly connected component. This is the set of members of the strongly connected component that have successors which are not members of the strongly connected component. We call this set the *exitPointSet*

- The set statements in the strongly connected component. We call this set the *firstBodyApproximationSet*

The next step is to determine the type of Java loop, which in turn will give us the loop's *naturalExit*.

3.1.2 `while` Loops

If the *entryPoint* is a Grimp `IfStmt`, in which only one of its successors is a member of the *firstBodyApproximationSet*, then we can represent the strongly connected component with a `while` loop, and set the *naturalExit* to be the successor that is not a member of the *firstBodyApproximationSet*. Figure 3.2 shows this pattern.

Greedily choosing to represent this strongly connected component as a `while` loop is good choice. Our overall goal is not to find the restructuring that is guaranteed to match the original source code, but just one that is understandable. From this point of view, each structured Java statement should try to encapsulate as much information from the control flow graph as possible. In our choice of loop type, we want to represent predecessor and successor information from the *entryPoint* statement. In this first example, a `while` statement captures the fact that there is a successor to the *entryPoint* that is not a member of the strongly connected component, and therefore fulfills our information requirement. No other type of Java loop will do this, and we make this choice greedily knowing that it saves the introduction of an extra Java `break` statement to exit the loop from the *entryPoint*, later on.

Finally, note that for a `while` statement, the *entryPoint* should be a Grimp `IfStmt`. The entry point must have more than one successor, so at least either an `if` or a `switch` is required. However, if the *entryPoint* is a `SwitchStmt` with more than two targets, we *may* build a conditional expression from the `switch` that evaluates

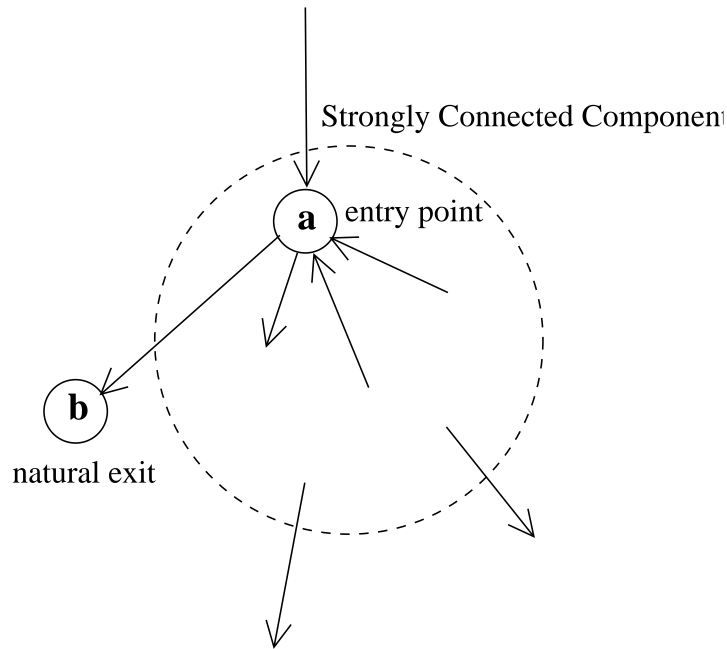


Figure 3.2: Pattern used to generate a `while` loop.

the `switch`'s local and create a new `IfStmt` before evaluating a reduced version of the `switch`. The result will be a `while` loop whose first statement is the reduced `switch`, both of which will be evaluating the same local. While correct, this could be more compactly represented by a `while(true)` loop whose first statement is the original `switch`. For compactness and expediency we choose this second option and ignore `switch` based conditional loops.

3.1.3 `do-while` Loops

If we did not find a `while` loop, the next step is to see if we should represent the strongly connected component with a `do-while` loop. If there's a predecessor to the *entryPoint* that is a member of the *firstBodyApproximationSet* which has a successor that is not a member of the *firstBodyApproximationSet*, then we can represent the strongly connected component with a `do-while` loop, and set the *naturalExit* to this successor. Figure 3.3 (page 30) shows this pattern.

This predecessor is what will eventually act as the condition for the `do-while`

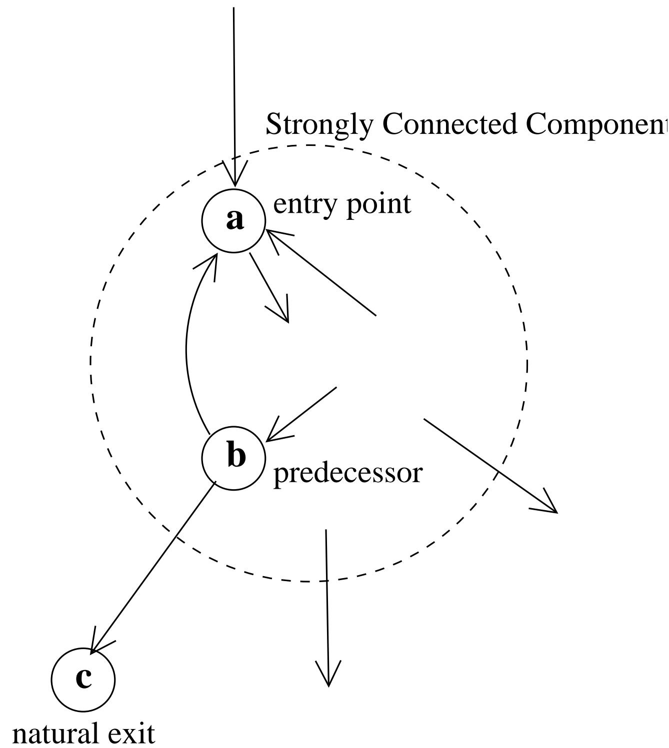


Figure 3.3: Pattern used to generate a `do-while` loop.

loop. If there is more than one candidate that qualifies, we simply abandon trying to restructure a `do-while` loop and mark the strongly connected component as an unconditional loop.

The intuition behind this choice is that when trying to represent the case where there are multiple `do-while` condition candidates, we note that there are n back edges to the *entryPoint*, where $n > 1$, and only one of these will be accounted for by the `do-while` condition. There are then $n - 1$ remaining back edges which must be represented by at least one nested loop. The choice of which candidate condition is used in the `do-while` will radically affect what embedded loops are created. Unfortunately, we have not found any heuristics which can reliably help with the selection of a good candidate. A better solution is to produce just a single unconditional `while(true)` which handles all the back edges at once, rather than 2 or more nested conditional loops.

3.1.4 `while(true)` Unconditional Loops

If neither a `while` nor a `do-while` loop are created, we build an unconditional `while(true)` loop. Edges that come out of this type of loop are all interpreted as `break` statements, or if they target some encapsulating loop's condition, as `continue` statements. Figure 3.4 (page 31) illustrates such a strongly connected component and the resulting structured code.

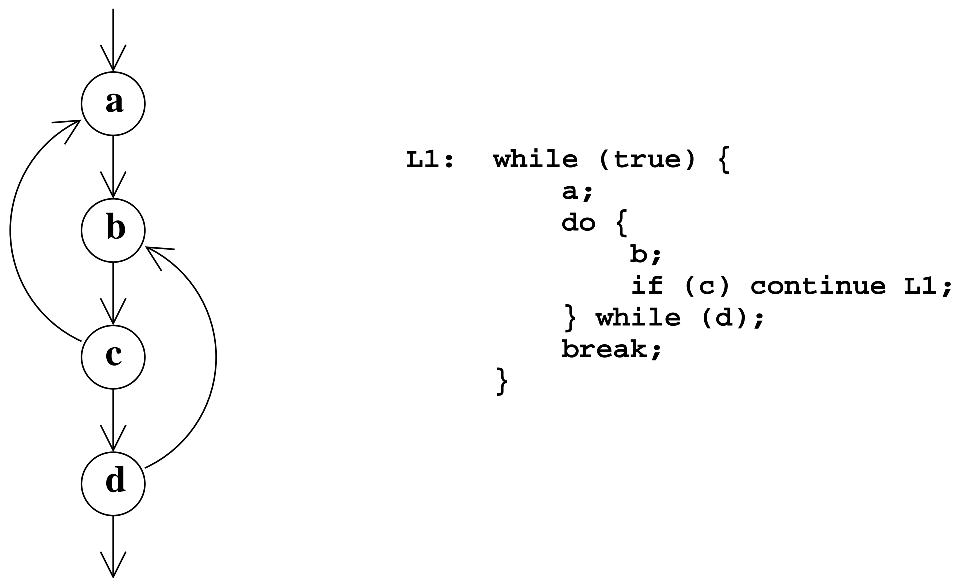


Figure 3.4: An example control flow graph that transforms to a `while(true)` loop, and its corresponding code.

In choosing the loop type we do the following. First we see that the *entryPoint* to the SCC does not have a successor that is not a member of the SCC. Therefore a `while` loop is not a good choice for representing this SCC. Next, the *entryPoint* does not have a predecessor in the SCC which has a successor which is not a member of the SCC. Therefore, a `do-while` loop is not appropriate either. Our fall back choice is an unconditional `while(true)` loop. The main consequence is that any edge out of the loop must now be a `break` statement. Accordingly the edge from `d` which leaves the SCC is now represented with a `break`.

3.1.5 Multiple entry point components

All nodes within a strongly connected component can reach all other nodes. Because of this, if the strongly connected component has more than one entry point, then there is no way to partition or reduce a strongly connected component into smaller parts, such that every part will have only one entry point. Since no Java statement has more than one entry point, we cannot directly represent a multi-entry point strongly connected component in Java. To proceed with restructuring, we are forced to perform a transform on the control flow graph.

There are two types of transform that could be applied: 1) versioning the strongly connected component for each entry point, or 2) the introduction of an artificial dispatch statement that will act as the single entry point.

Although versioning introduces no new semantic to the code, there are two arguments against it: 1) it multiplies the size of the code within the strongly connected component by the number of entry points, which obfuscates the code's purpose, and 2) the strongly connected component does not represent any structured construct, so attempts to view parts of it as such will likely be nonsensical. We chose to introduce artificial control flow dispatch statements in Dava because it preserves the general intention of the strongly connected component, yet only adds a set of assignments to the new flag variables in the code.

An example of this second transform is given in figure 3.5. Here we convert a "`do-while`-ish" loop that can be entered directly into it's body into a `while(true)` loop. The new "primed" statements direct control flow. For example, *c'* is an assignment that sets some flag to indicate that we want control flow to be directed to *c*, *b'* set the flag to indicate *b*. The statement *D* is a `switch` or `if` statement that dispatches control flow based on the value of the control flow flag.

Although the above figure is easy to understand, it hides an important issue. In the transform we had to redirect only *some* but not all of the edges that target the entry points. Here are the rules for which edges are redirected to the dispatch statement.

1. Choose one of the entry points to be the *natural entry point*. Let all other entry points be call *synthetic entry points*

2. For the natural entry point, redirect all control flow that targets it to the dispatch statement, via a flag setting statement.

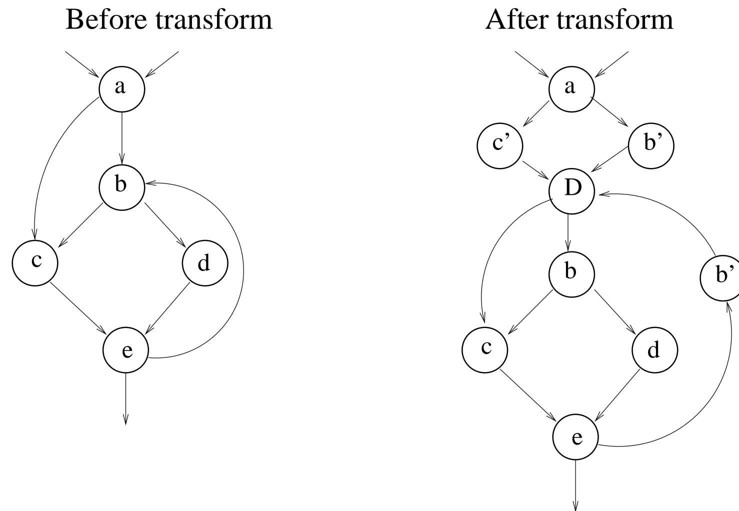


Figure 3.5: A simple multi-entry point strongly connected component and its conversion to a single entry point strongly connected component.

3. For the synthetic entry points, redirect only control flow sourcing from outside the strongly connected component to the dispatch statement, via a flag setting statement.

The consequent issue, then, is how to choose which entry point is to be the natural one. In general, we do not know how many Java loops it will take to represent a strongly connected component until we have actually done the restructuring. Unfortunately, the number of loops needed may be changed by selecting different entry points as the natural one.

There are two possible approaches to this problem. The first is to do an in depth search of all possible restructurings for the strongly connected component and then to select the one that generates the fewest Java loops. This will be both complex and slow. The second alternative is to use some heuristic to guess which entry point should be selected.

There are several heuristics that could be used, including intervals [3] and smallest cycle finding [12]. The heuristic we currently use is to perform a depth first search within the strongly connected component starting from each entry point. For each DFS, we count the number of targets of back edges, realizing that in the future restructuring, these targets are *likely* to become the entry points of embedded loops.

Wanting to minimize the number of loops needed to represent the component, we choose the DFS from the entry point that produces a minimal number of targets of back edges. Figure 3.6 shows an example graph with two possible entry points.

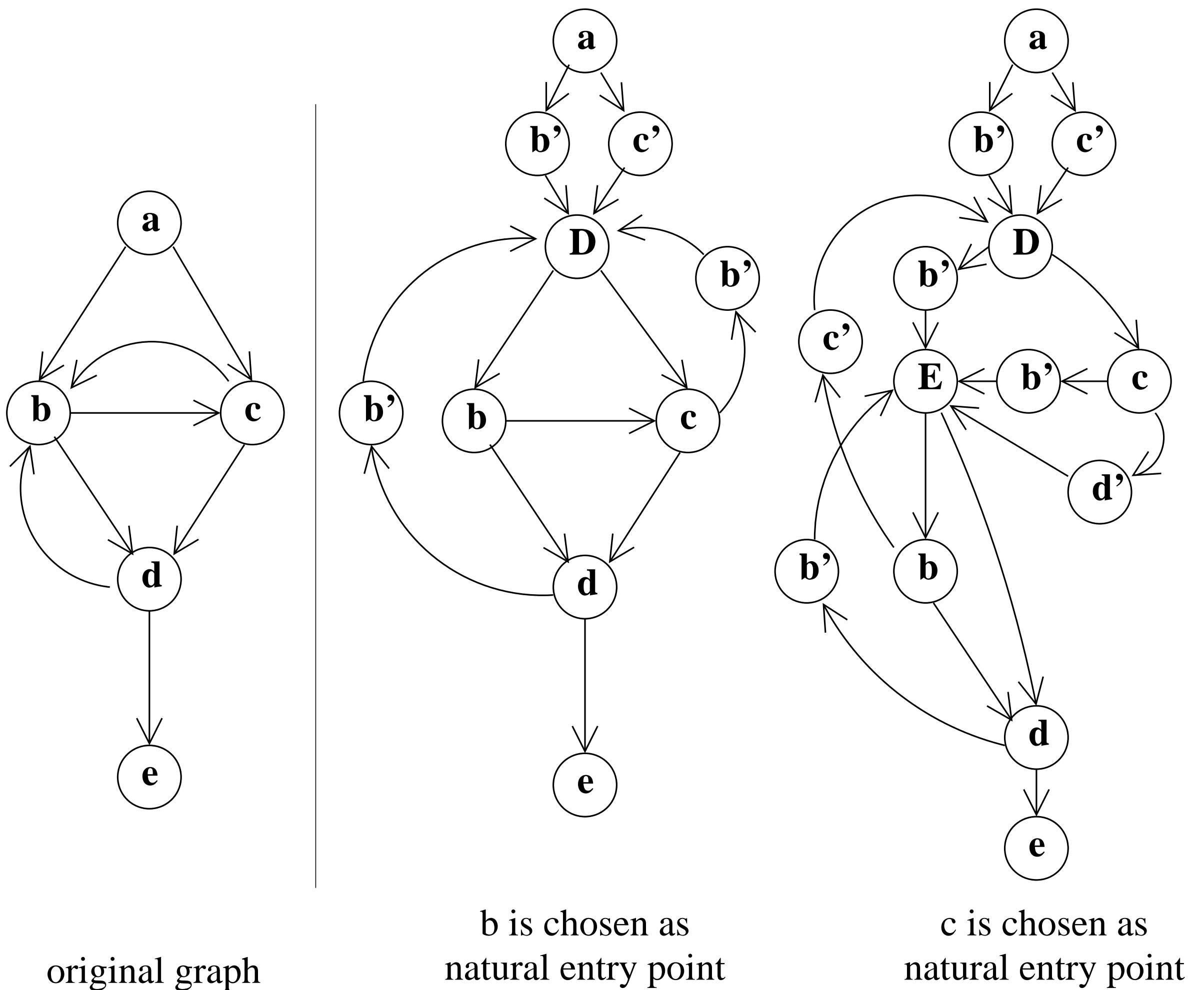


Figure 3.6: A complex multi-entry point strongly connected component with two possible transforms.

If statement `b` is chosen as the entry point there is just 1 target of back edges, while if statement `c` is chosen, there are 2. The corresponding code in figure 3.7 is ugly no matter which we choose but it is worth noting that the code from choosing `b` is substantially smaller and simpler.

```
if (a)
    b';
else;
    c';

while (true) {
L1: {
        switch (D) {
            case b:
                if (b) break L1;
            default:
                break;
        }
        if (c) {
            b';
            continue;
        }
    }
    if (d) break;
    b';
}
e;
```

```
if (a)
    b';
else
    c';

L1: while (true) {
    switch (D) {
        case b:
            b';
            break;
        default:
            if (c)
                b';
            else
                d';
            break;
    }
    while (true) {
        switch (E) {
            case b:
                if (b) {
                    c';
                    continue L1;
                }
            default:
                break;
        }
        if (d) break L1;
        b';
    }
}
e;
```

Figure 3.7: The generated code from figure 3.6

When placing the new flag-setting and dispatch statements in the control flow graph, we have to be careful that they are placed in the appropriate exceptional zones as defined in the method's exception handler table. For example, if all the entry points are within a certain zone, then the new dispatching entry point should also be placed in the zone. If only some of the entry points are in a zone, then the

new entry point should *not* be placed in the zone. Flag setting statements should always be placed in the same zones as their predecessors.

3.1.6 Nested loops

A strongly connected component may be made up of more than one Java loop. Until now, we have only looked at finding what should be the outer-most of these loops. To find nested loops in control flow graph G we remove a single statement from G producing a new control flow graph G'. This statement must obey the following two properties: 1) it *cannot* be part of a nested loop, and 2) it must always be run for every loop iteration. If these conditions are met, then when this key part of the loop is removed, we have effectively removed the outer Java loop from G'. We can then evaluate G' for its strongly connected components to find loops that nest in those found from G. As stated earlier, this process is repeated until we reach some G^n that contains no strongly connected components and all nested loops have been found.

Fortunately, there always exists at least one statement for every Java loop that has these properties. Dava generates three types of Java loops: `while`, `do-while`, and `while(true)`. We begin looking at the two conditional loops, `while` and `do-while`. In a conditional loop, we remove the statement that houses the loop's conditional expression. The reasoning is that 1) for every complete iteration of a conditional loop, the condition must be evaluated, (note that `continue`s target the conditional) and 2) that no nested loop can contain the current loop's conditional expression. In other words, for every time we pass this conditional, we may assume we are iterating on our current and not a nested loop. We remove it from G' and remove the outer Java loop.

Unconditional loops are similar: we remove the entry point of the strongly connected component. First note that for every complete iteration of the loop the entry point must be executed. Although the Java language form of the unconditional loop (`while(true)`) has a constant conditional expression in it, the resulting bytecode never includes an evaluation of `true`. Instead, we automatically execute the first statement of the loop, which happens to be the entry point of the strongly connected component. As such it is roughly correct to state that in unconditional loops, `continue`s target the entry point.

Second, we show by contradiction that we can always safely interpret G such that the first statement is not a member of a nested loop. Consider that it is a

member of nested loop. Since our target statement is the entry point to the strongly connected component, we know that it must be the entry point to the nested loop. Now, the nested loop must be one of the three types of loops. If it is a `while` loop, we know that one of the successors of the entry point is not a member of the strongly connected component. However, this cannot be the case, because if it were we would have restructured the current encapsulating loop as a `while` loop and not as a `while(true)`. Next consider that the nested loop is either a `while(true)` or `do-while` loop. In this case every back edge that targets the entry point need not be represented by normal control flow in the nested loop, but can be represented as a `continue` in the encapsulating loop. Therefore there is no control flow graph that will necessitate that we add nested loops that use the unconditional loop's entry point and hence we remove it.

3.1.7 Loop Bodies

The body of a Java loop may contain more than just the statements in the strongly connected component. The code fragment in figure 3.8 shows that the body can also hold some of the statements that members of the strongly connected component dominates. Loop bodies are found by the following algorithm.

1. The *escapingSuccessor* is the successor to the loop condition statement, which is itself not a member of the loop. By construction, `while` and `do-while` loops have a naturally occurring *escapingSuccessor* (`e` in figure 3.8) and the `while(true)` loop does not. For the `while(true)` loop we take the set of successors to loop members, which themselves are not in the loop, and select the one that has the longest shortest path to any exit of the method.

2. The *escapersReachingSet* is made up of the *escapingSuccessor* plus all statements that are reachable from the *escapingSuccessor* without passing through the entry point of the strongly connected component.

3. The *bodySet* made up of the strongly connected component's entry point plus all statements in the control flow graph that are dominated by the entry point, minus the *escapersReachingSet*.

The statements of the *bodySet* are then used to place the loop into its appropriate position in the SET.

```
while (x < 10) {                    // a
    if (x % 5 == 1) {               // b
        System.out.println(x);      // c
        break;
    }
    x += y;                         // d
}
System.out.println("done");         // e
```

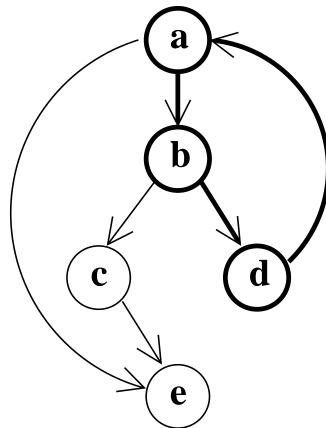


Figure 3.8: Code and corresponding control flow graph.

3.1.8 Putting Loops in the SET

Loops are nested in the SET according to the *bodySet* set relations that they have with constructs already in the SET. A loop will be a descendant of every construct that has a *bodySet* which is a superset of its own, and an ancestor of every construct that has a *bodySet* which is a subset. Fortunately, we know that the *bodySet* for a newly constructed loop will always be a subset or superset of any *bodySet* already in the SET. We claim this with the following two points in mind.

1. Any strongly connected component in any of the G^n control flow graphs will either be disjoint, a proper subset, or a proper superset to any other strongly connected component.

2. The non-cyclic control flow is found by dominance and reachability of the loop's

entry and escaping successor points. Since the entry point is a member of the strongly connected component, we know that every ancestor loop in the SET already contains this statement. Because both dominance and reachability are both transitive, we know that all ancestor loops' *bodySets* non-cyclic control flow must therefore be a superset of the current loop's non-cyclic control flow.

3.2 DAGs

Once we have identified all the conditional loops, the remaining unaccounted-for conditionals in the control flow graph should be represented by DAG (`if` and `switch`) control flow statements. This section is divided five subsections which are primarily concerned with correctly finding *bodySets* for DAG statements. Section 3.2.1 discusses why *bodySets* are so important and introduces a problem and solution which is common to finding *bodySets* for any type of DAG statement. Next, section 3.2.2 presents a small transform that allows us to make an important assumption about the control flow graph. Then, in the last three sections (3.2.3 through 3.2.5) we examine how `if`, `if-else` and `switch` statements and their *bodySets* are found.

3.2.1 Putting DAGs in the SET

SET nodes are placed in the SET according to their *bodySets*. The basic condition for finding DAG *bodySets* is that each *bodySet* should be dominated by its entry point. For example in an `if-else statement`, the set of statements in the `else` clause should be dominated by the first statement of that `else` clause. While necessary, dominance is not sufficient for determining membership in a *bodySet*.

Consider figure 3.9 (page 40). Focus on the conditional `b` in the left hand side of the diagram. Intuitively, we have a simple `if-else` statement using the conditional expression from statement `b`, and having an empty `else` clause. If `b` then do `c` and `d`, else nothing, then do `e`. On the right side of the diagram, however, simply by moving a target of `d`, we are given a `do-while` loop with its condition held in statement `d`. The `if-else` statement at `b` now performs `c`, else a `break` from the loop to statement `e`. The key issue is that while it would be correct to include `d` in the `if`'s *bodySet* on the left hand side, it would be wrong to include it in the `if`'s *bodySet* on the right hand example.

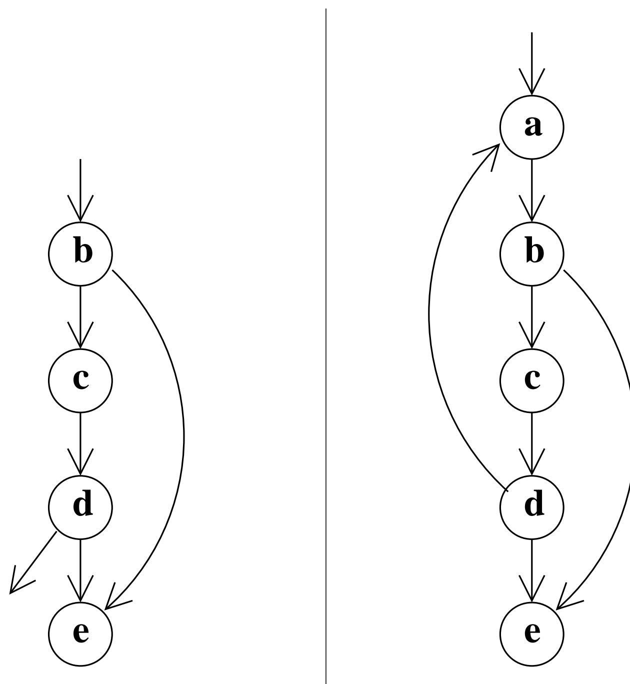


Figure 3.9: Finding a DAG *bodySet*

The solution is to optimistically use dominance and create *bodySet*s that are potentially too large and then trim it when nesting its SET node into the SET. Assume that we are in the middle of trying to nest the new node in the SET. That is, we have found that the new node's *bodySet* is a descendant of several nodes already in the SET, and we are performing the evaluation for some *current* SET node. Examine the children of the current SET node and determine which contains the *entryPoint* of the new node. If it exists, call this child the *targetNode*. If the *targetNode* has a *bodySet* that is a not a superset of the current node's *bodySet*, trim the new node's *bodySet* so that it is. We then recurse down the SET with the newly trimmed node.

Consider the left side of figure 3.10 (page 41). Here we have an SET. Our current node is **A**. We already have properly nested node **B** in **A** and now want to nest node **C**. However, because we are finding *bodySet*s optimistically we can neither put **C** as a sibling, nor as a child of **B**. However, we note that the *entryPoint* of **C** is in **B**'s *bodySet*. We then *trim* **C** by removing all statements from **C**'s *bodySet* that are not in **B**'s.

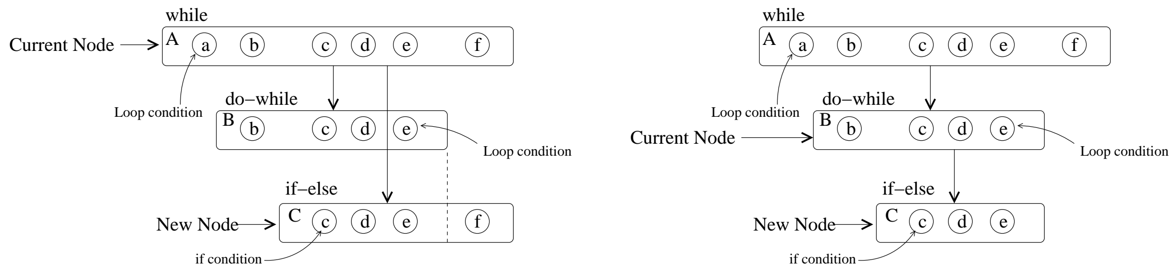


Figure 3.10: Trimming a DAG SET node.

The effect of this is that the new node's *bodySet* will nest properly in the SET. Because the *bodySet* is determined by the *entryPoint*, and the operation of trimming will never remove the *entryPoint* we can be certain that we will never somehow destroy the new DAG node.

3.2.2 A small transform that simplifies design

In finding the *bodySet* we are constantly using dominance. It may easily occur that a conditional doesn't dominate anything, and so should have an empty *bodySet*. We will see, however, that it is convenient not to use the conditional's dominance, but its successors' dominance to determine the *bodySet*. Unfortunately, if the successor is not dominated by the conditional and we blindly apply successors' dominance we will get incorrect *bodySet*s.

Consider the example on the left hand side of figure 3.11. If we use the dominance of `b` and `c` to find *bodySet*s, we will incorrectly include statements that are not dominated by `a`. A simple transform that solves this problem is to insert an unconditional direct jump between every conditional and its targets. In the example, these jumps are statements `x` and `y`. This introduces no new semantics to the program and guarantees us that that every conditional does indeed dominate its successors.

3.2.3 `if` statements

An `if` statement is detected when the following conditions are met.

1. The current statement is a `Grimp` `if` statement.

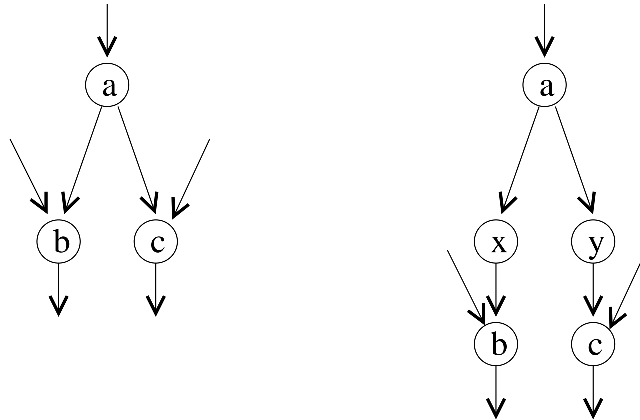


Figure 3.11: A small transform that guarantees that successors of conditionals are dominated.

2. The `if` will have two successors here called `a` and `b`. Without loss of generality, if `a` is reachable from `b` without passing through the `if` statement, we will call `b` the *branchSuccessor*. For an `if` statement to be detected, the branchSuccessor must exist.

The *bodySet* for the `if` statement is the set of the *branchSuccessor* plus all statements that it dominates. This *bodySet* will then be trimmed to accommodate any constructs already existing in the SET. It is important to note that the trimming here can only be done when nesting into a `do-while` or `while(true)` loop. Consider the following impossible cases.

1. We are nesting into a conditional `while` loop. The *entryPoint* of the `if` will be a member of the `while`'s *bodySet*, but cannot be the *entryPoint* of the `while` loop (if it were it would be the condition of the loop and not the condition of a DAG node). Because of this we know that if the loop's *bodySet* were solely defined by dominance of the loop's *entryPoint* (and since dominance is a transitive property) the `if` statement's *bodySet* would properly nest in the `while` loop's. However, statements reachable from the `while` loop's *escapingSuccessor* are removed from its *bodySet*. But note that any path from `while` loop's `bodySet` to the loop's *escapingSuccessor* must pass through its *entryPoint*, and so our DAG's conditional can not dominate the *escapingSuccessor*. Accordingly, the

DAG's *bodySet* can not contain anything reachable by the *escapingSuccessor* and we know that the `if` must nest properly in the `while`.

2. We are nesting into another DAG construct. This is easier than the previous case. Again, the *entryPoint* of the `if` will be a member of the DAG's *bodySet*, but not the DAG's *entryPoint*. Since all DAGs' *bodySet*s are defined by the dominance of their *entryPoint*s, and because dominance is a transitive property, the `if` statement must nest properly in the DAG construct. Note that if the DAG construct has been trimmed by a `do-while` (for example), we must have already experienced the same trimming from the same `do-while` on the `if`'s *bodySet* during the nesting process.

3.2.4 `if-else` statements

An `if-else` statement is detected when the following conditions are met.

1. The current statement is a `Grimp if` statement.
2. An `if` statement as described in section 3.2.3 is *not* detected.

The *bodySet* for the `if-else` statement is found in a similar manner to the `if` statement except that both successors are used to contribute, rather than just the *branchSuccessor*. Additionally, we partition the *bodySet* into the *ifBodySet* and the *elseBodySet*. These two sets are then used for performing nesting of sub-nodes in the SET.

3.2.5 `switch` statements

Multi-way conditional branches are provided with Grimp `switch` statements. These hold a conditional expression, plus a series of direct targets based on the evaluation of the expression, and a default target.

Our basic treatment of `switch` statements is just a generalization of `if-else` statements. The *bodySet* for each `case` is simply the target statement for that case plus everything that it dominates. The *bodySet* for the `switch` as a whole is the union of the `case` *bodySet*s. Figure 3.12 illustrates this.

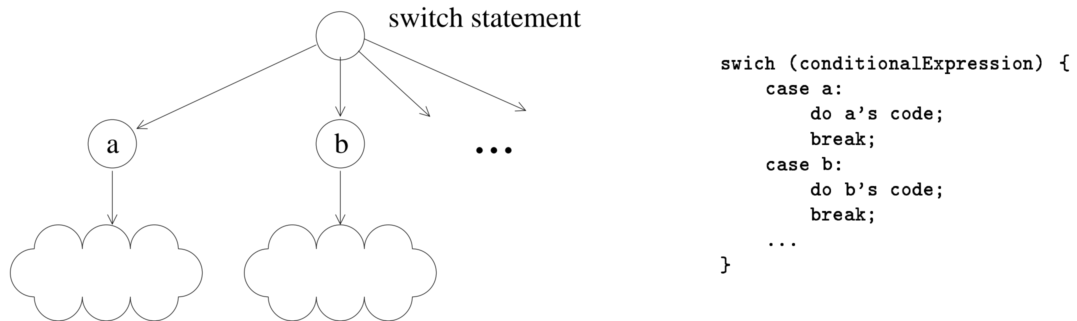


Figure 3.12: A simple `switch` statement.

Although this will give a correct answer, it may be too conservative. Case fall-throughs allow us to include more in a case's *bodySet* than would be otherwise possible. Consider figure 3.13 (page 44). Here, statement `b` does not dominate the control flow that follows it, and hence `b` will have an empty case. However, we would prefer to put this control flow into `b`'s case and simply have a fall-through from `a`'s case to `b`. A simple transform that will allow us to do this is with our original dominance based algorithm is to remove the path from `a`'s case to `b`'s follower. Now `b` will dominate its follower and the appropriate *bodySet* will be found.

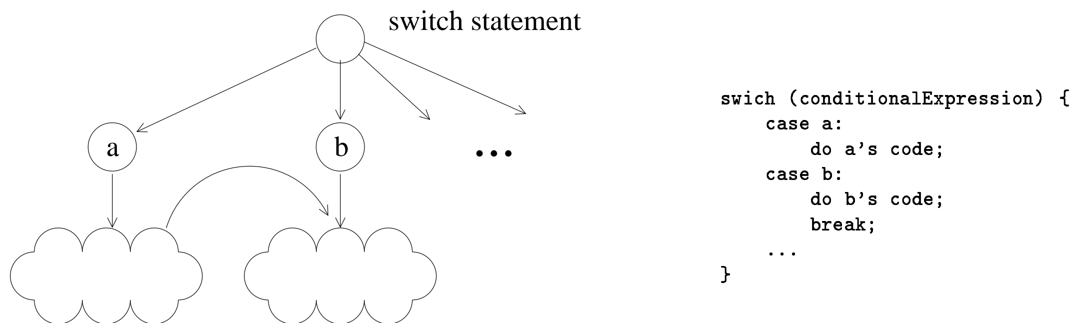


Figure 3.13: A `switch` statement with a case fall-through.

As well, if there are several `case`s that essentially target the same code, such as `b`, `c` and `d` do in figure 3.14 (page 45), we can perform the same edge removal trick to find the appropriate *bodySet*s.

There are, however, two situations where the edge removal trick should *not* be

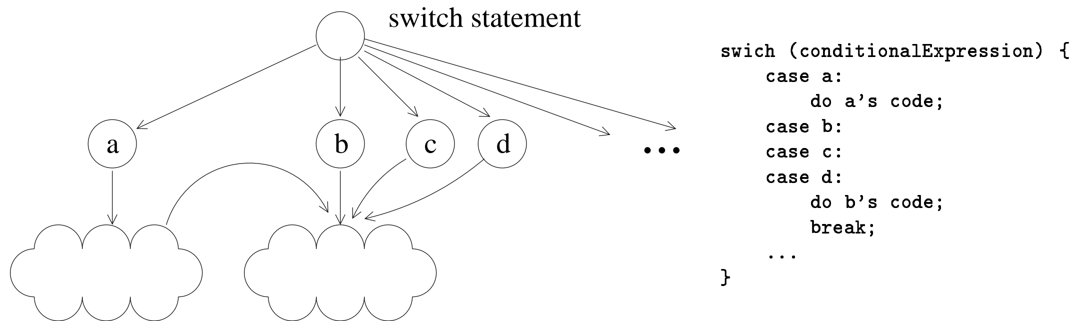


Figure 3.14: A `switch` statement with multiple case fall-throughs.

applied. First, as shown in figure 3.15 (page 45), there may be more than one fall through predecessor that does not directly target the `case` body. Intuitively, a graph of the `case`s in a switch will form a forest of linked lists, but it is impossible that these lists should merge. Second, as shown in figure 3.16 (page 46), there can be no cycle in the reachability of `case` bodies. If there is, we are faced with the absurdity that there is no "first" case in the `switch`

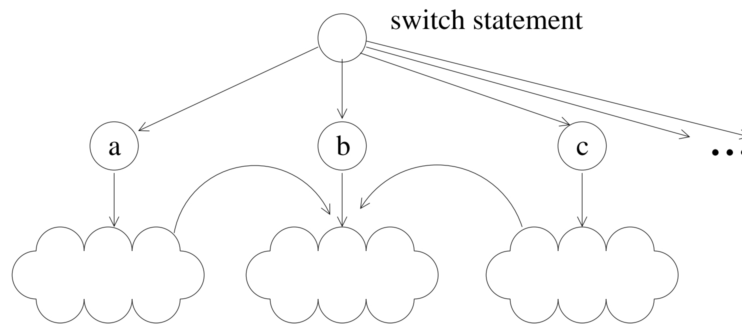


Figure 3.15: A `switch` statement that cannot employ case fall-throughs.

Algorithm 1 **findStructuredSwitchBodySets** (page 53) finds the case *bodySet*s for a structured `switch` with these facts in mind. Finally, we must order the cases. In algorithm 1 we define the graph *G'* to represent reachability between the cases of the `switch` statement. We note that the cases which have non-empty *bodySet*s will form a forest of linked lists in *G'*. The only necessary ordering is to make sure that members of these linked lists are placed in the `switch` contiguously in the order that they appear in their respective lists.

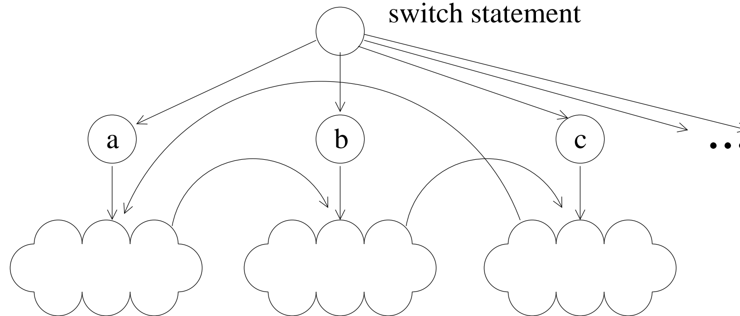


Figure 3.16: A second `switch` statement that cannot use case fall-throughs.

3.3 Labeled Blocks

Imagine that we have created a SET that contains nodes that fully represent both the regular and exceptional control flow, except that it does not yet have labeled blocks. The following example will build up such a case.

Consider the pseudocode and the corresponding control flow graph in figure 3.17 (page 47). This expresses regular control flow, and for simplicity, the extra unconditional jumps from section 3.2.2 have been left out. We are are also presented with an SET in side view. This SET has been augmented with inter-child control flow. The only difference this makes over a normal SET is that control flow between sibling SET nodes has been added to the diagram. The new feature is the SET in summary top view. Here, we have dropped the CFG statements from the diagram and summarized the control flow between SET siblings.

In our example we would say that an inter-child control flow graph is made up of nodes **B**, **C** and **D**.

We can now notice several properties for any SET node.

1. All SET nodes, with the exception of statement sequences, will have at least one child SET node.
2. With the exception of the unconditional loop, the graph of control flow between the children of a SET Node will form a DAG. Proof by contradiction: Assume that the control flow between children of some SET node forms some cycle. A cycle is a strongly connected component, which will be represented by SET

```
label_a:   if (a) then goto label_c;
label_b:   if (b) then goto label_d;
                else goto label_e;
label_c:   if (c) then goto label_e;
label_d:   d;
label_e:   e;
```

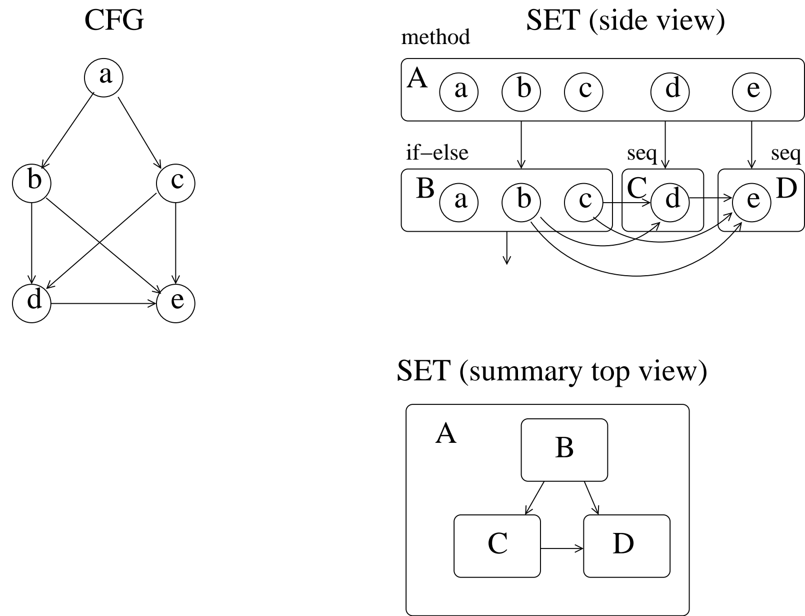


Figure 3.17: Summary SET from the top view.

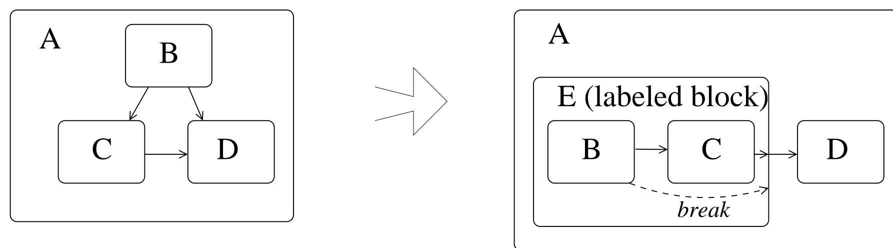
looping node. Therefore the *bodySet* of this SET loop node must be equivalent to the union of the *bodySet*s of *its* children that form the cycle. For every conditional loop, we know that the condition is not a member of any child SET node. Because we have disallowed unconditional loops, we know that these *bodySet*s cannot be equivalent, and hence that the inter child control flow forms a DAG.

3. With an unconditional loop SET node, removing the control flow that targets the *entryPoint* of the loop will result in the control flow between the children forming a DAG. By construction we know that removing the *entryPoint* from an unconditional loop breaks the loop, and from the proof above that inter-child

control flow must pass through every child, including the one that contains the *entryPoint*. Therefore breaking control flow to the *entryPoint*, must break the inter-child control flow cycle, and produce a DAG.

4. By construction there is only one entry point to each inter-child control flow graph. For example, while an `if-else` will have two entry points, each of the `if` and `else` clauses', while the `if-else` as whole DAGs will have only have one entry point.

We can now see that the inter child graph of any SET node forms a DAG. The Java grammar, however, dictates that this graph should be a linked list of statements. We transform the DAG into a linked list by introducing labeled blocks. Figure 3.18 (page 48) shows the DAG from our previous example being transformed into a linked list and the resulting structured code.




```
child_E:  {
          child_B:  if (a) {
                        if (b)
                            break child_E;
                    }
                    else {
                        if (c)
                            break child_E;
                    }
          child_C:  d;
          }
child_D:  e;
```

Figure 3.18: Summary SET from the top view with labeled block solution.

One can trivially transform any DAG to a linked list by performing algorithm 2 **TrivialLabeledBlocks**.

While correct, this solution is only a proof of possibility, and will produce far too many labeled blocks. The challenge is to find an algorithm and a topological sorting of SET node children that requires the introduction of a minimal number of labeled blocks.

To begin with, consider figure 3.19 (a) (page 49). This represents the the inter-child control flow graph of the children of a SET node. The important feature of labeled blocks is that we can `break` from them and immediately go to the statement following them. Our first goal will be to place just the labeled block *closings* within the graph to allow direct control flow to the statements that follow the closings.

Now, for every join point in the graph there will have to be a closing in front of it. The reason is that only one of the join points' predecessors can have the natural fall through to the join point, and therefore the other must be some control flow path that reaches the join point by breaking from a labeled block. We can therefore claim that the number of labeled blocks must be at least the number of join points. This yields us figure 3.19 (b) with labeled block closings in front of the nodes `(e,g)`.

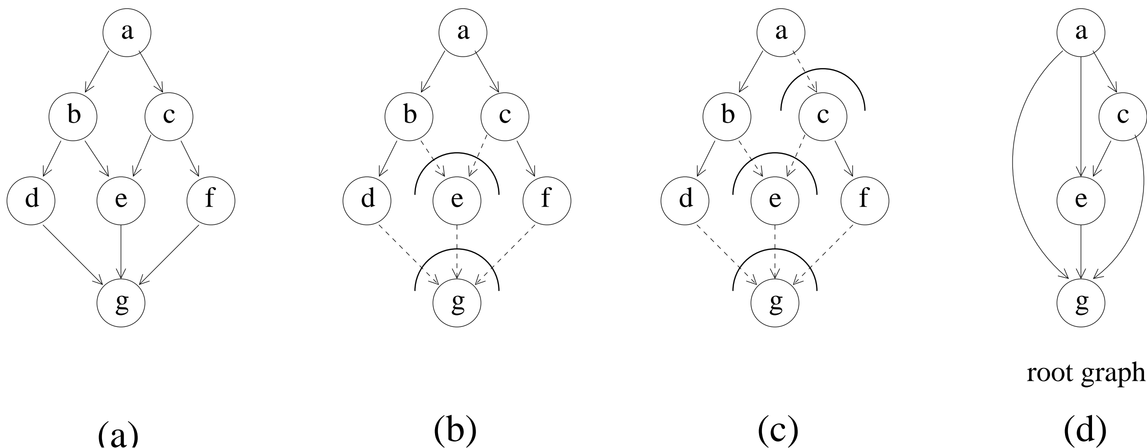


Figure 3.19: An inter-child control flow graph.

Now, for every node within each tree in 3.19 (b), we know that it can only have one natural successor, so the other successors must be reached again by some break from a labeled block. Accordingly, for each node, we must place closings in front of *all*

but one of its successors. This will yield a forest of linked lists which we see in figure 3.19 (c) with labeled block closings in front of nodes `(c,e,g)`. Note that since we are performing this second round of closing insertions into trees rather than DAGs, the selection of which successors we choose to place closings in front of will make no difference to the total number of closings that are inserted in each tree. Therefore, given a number of join nodes j in the DAG, the number of branching nodes b in the resulting trees and the number of successors s to these branching nodes, the minimal number of labeled block closings is $j + s - b$.

To simplify matters we next build a *root graph* which expresses connectivity between the roots of linked lists. Since all the necessary labeled blocks have been already accounted for, any topological sorting of the root graph will yield the same minimal number of labeled blocks. Figure 3.19 (d) shows the root graph and yields us a topological sorting `(a,c,e,g)`. This sorting is then expanded into the vertex sorting `(a,b,d,c,f,e,g)` where each member of the root graph is replaced by its linked list.

We now present our topological sorting that induces a minimum of labeled blocks in algorithm 3 **EnhancedTopologicalSort** (page 54).

Once we have our topological sorting S of the children of an SET node, inserting the labeled blocks is straight forward. We traverse S from head to tail checking if the current node has any predecessor which is not immediately previous to it in S. If so, we find the immediately previous node, p_1, and the predecessor that has the earliest position in S, p_0. We then create a labeled block b that encapsulates the sequence of $p_0..p_1$, and give it a summary of the control flow associated with the members of $p_0..p_1$. The sequence $p_0..p_1$ is then removed from S and replaced by b.

The net effect is to reduce each inter-child control flow graph to a singly linked list through a minimal insertion of labeled blocks.

3.3.1 Labeled Breaks and Continues

So far, this algorithm has focused only on the contents of structured statements, and ignored categorizing which edges are to represented by abrupt control flow. There are two basic types of abrupt control flow: labeled `breaks` and labeled `continues`. The unlabeled versions of these can be equivalently represented by labeled versions and we begin by representing all abrupt control flow as labeled, and later remove the labels when possible.

Each Java language control flow statement that we have detected so far has well defined natural exit points as shown in figure 3.20. Note that in many cases the exit point is defined as the exit point of an encapsulated control flow statement. This means that we will have to perform a recursive search of the SET if we want to find the exact simple statements that defines the exit point(s) for any one structured statement.

Statement	Natural exit point
simple statement	itself unless this is a `return` or `throw` statement
statement sequence	the exit point of the last statement in the sequence
labeled block	the exit point of its statement sequence
`if`	the condition of the if and the exit point of the if clause's statement sequence
`if-else`	the exits points of both the if and else clauses' statement sequences
`try-...-catch`	the exit point of the try and catch clauses' statement sequences
`while`	the condition of the while
`do-while`	the condition of the do-while
`while(true)`	none
`switch`	none

Figure 3.20: Natural exit points from structured statements.

To determine if we have abrupt control flow, we simply test whether control flow is following these conditions, and if not, mark it as either a `break` or `continue`.

Continues

We first search the SET for situations that demand `continue` statements. Although the `continue` statement can many times be equally represented by a `break`, the `continue` keyword conveys extra information that the abrupt control flow is loop related, and we will choose to use it whenever applicable. The algorithm for finding `continues` is given in algorithm 4, **FindContinues** (page 55).

The intuition is simple: any control that comes out of a loop's body and targets the loop's condition may be a `continue`. We then check to see if the source of this

control flow is one the natural exits of the loop body's statement sequence, and if it isn't, mark it as a `continue`.

Breaks

We now search the SET for `break` statements in algorithm 5, **FindBreaks** (page 55). Again, the intuition is straight forward. A SET node's children form a control flow linked list. Any one of the control flow edges coming out of the source child and going to the entry point of the destination child may be a `break` from the source child. We check to see if the source statement is a natural exit point of the source child, and if not, mark the edge as a `break`.

3.3.2 Removing labels

If for either a `break` or `continue` the following conditions are met, then the label is removed from the statement.

1. The abrupt statement is targeting either a loop or a `switch` statement. For example, we might be `break`ing from a `while` loop.
2. There is no labeled statement, labeled block, loop or `switch` that is both a parent of the abrupt statement and a child of the statement that is being broken from or continued on.

Finally, if for some labeled statement s there are no abrupt statements that carry the same label, the label is dropped from s.

Algorithm 1 findStructuredSwitchBodySets(GrimpSwitchStatement *rootStmt*, ControlFlowGraph G)

1. Define the ***targetSet*** as the set of statements that are control flow successors to the *rootStmt*. From section 3.2.2 we know by construction that each of member of the *targetSet* is dominated by the *rootStmt*.
2. Define the ***originalTargetSet*** as the set of statements that are targeted by the members of the *targetSet*.
3. Define the ***targetGroupSet*** as the empty set.
4. For each statement $s \in originalTargetSet$ do
   (a) Define $\boldsymbol{m}$ to be the set of members of the *targetSet* that target s
   (b) $targetGroupSet \leftarrow targetGroupSet \cup m$ (note: $targetGroupSet$ is a set of sets)
   (c) If there is more than one edge from the members of m to s, remove all but one of these edges.
5. Build a graph ***G'*** of reachability between the members of the *targetGroupSet*. Note that the nodes of *G'* will be sets (m) of members of the *targetSet*.
6. Define $\boldsymbol{C}$ to be the set of cycles in *G'*
7. For all cycles $c \in C$
   For each set $m \in c$
   build an empty case *bodySet* for m
   remove m from *G'*
8. Define $\boldsymbol{B}$ to be the set of basic blocks in *G'*
9. For every basic block $b \in B$
   For each set $m \in b$
   if m is a join point in *G'*
   give m an empty case *bodySet*
   else
   remove all edges from m's predecessor to m's target
   build m's case *bodySet* as the set of statements that
   any member of m dominates

Algorithm 2 TrivialLabeledBlocks(DAG G)

1. Find any topological sorting T of G.
2. While the number of vertices $v \in T > 2$
   (a) Remove the first two vertices (v_1, v_2) from T. Call the new head of T v_3
   (b) Embed v_1 and v_2 in a new vertex v_4 representing a labeled block.
   (c) Place v_3 at the head of the T. Now all control flow from v_3 to v_4 will be represented with a break from v_3.

Algorithm 3 EnhancedTopologicalSort(DAG G)

1. Define the ***rootSet*** to be the empty set.
2. For each vertex $v \in G$ do
   if the in-degree of $v \neq 1$
   $rootSet \leftarrow rootSet \cup \{v\}$
3. For each vertex $v \in G$ do
   (a) Get a clone ***S*** of the set of successors of v
   (b) Remove any members of S which are members also members of the *rootSet*
   (c) Place all but one of the remaining members of S in the *rootSet*
4. For each vertex $r \in rootSet$ do
   - Find the linked list associated with r.
   - Find the *rootSet* successors to r.
5. Define the ***topologicalSorting*** to be the topological sort of the *rootSet*.
6. Replace each vertex $v \in topologicalSorting$ by the linked list associated with v.
7. **return** *topologicalSorting*

Algorithm 4 FindContinues(SETNode *S*)

1. For each child $c \in S$ do
   if *c* is not a sequence of simple statements
   FindContinues(*c*)
2. **if** *S* is one of `while`, `while(true)` or `do-while`
   (a) The *target* is defined as the condition of *S*, if *S* is a conditional loop, or the entry point of *S*, if *S* is a `while(true)` loop. As with the natural exit point of a Java language control flow statement, the entry point is also well defined, and may require a recursive search of the SET to find it.
   (b) The *naturalExitSet* is defined as the natural exit points of the statement sequence that forms the body of the loop.
   (c) The *target* has a set of control flow predecessors *P*. *P'* is the subset of *P* that are members of *S*'s *bodySet*.
   (d) For each $p \in P'$ where $p \notin naturalExitSet$ do
   Edge (*p*,*target*) is marked as a `continue`.
   (*p*,*target*) and *S* are labeled appropriately.

Algorithm 5 FindBreaks(SETNode *S*)

1. For each child $c \in S$ do
   if *c* is not a sequence of simple statements
   FindBreaks(*c*)
2. For each child $c \in S$ do
   (a) Find the next child c_{next} in *S*. By construction, control flow from *c* will only go to c_{next}. If there is no c_{next}, return.
   (b) The *target* is defined as the entry point of c_{next}.
   (c) The *naturalExitSet* is defined as the set of natural exit points of *c*.
   (d) The *target* has a set of control flow predecessors *P*. *P'* is the subset of *P* that are members of *c*'s *bodySet*.
   (e) For each $p \in P'$ where $p \notin naturalExitSet$ do
   Edge (*p*,*target*) is marked as a `break`.
   (*p*,*target*) and *c* are labeled appropriately.

Chapter 4

Exceptions

4.1 Introduction

Restructuring exceptional control flow into Java `try-catch` statements is one of the more difficult problems of decompiling Java bytecode into Java source. To simplify this task, it is broken into three main parts which are located in three areas of the of the decompiling algorithm.

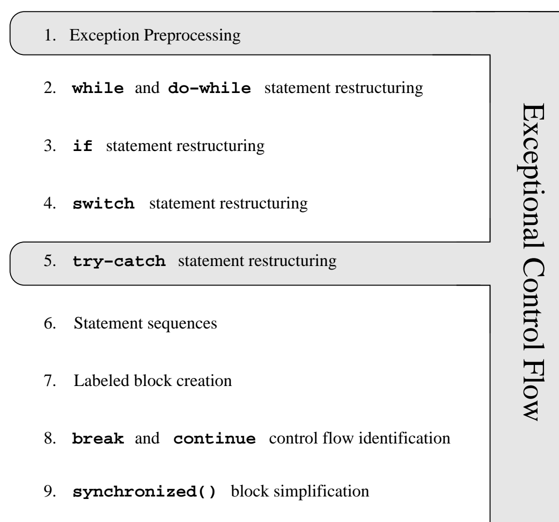


Figure 4.1: Exception based control flow phases.

Exception table entry removal (section 4.2) and exception pre-processing (section 4.3) are placed in the phase 1 of the algorithm, and perform several transforms to the control flow graph that ensure structured exceptional control flow. Exception handling (section 4.4) performs the restructuring and building of `try-catch` statements in phase 5, and finally section 4.4.1 removes any obviously spurious `try-catch` blocks at the very end of the algorithm. This last step is fairly minor and does not rate as a full phase in the decompiling algorithm.

We begin in this introduction, however, by examining why restructuring exceptions is hard. Exceptions are represented in Java bytecode and Grimp by the use of an exception handler table. The table specifies a relation made up of a range of instructions, the type of the exception being caught and the target instruction to jump to, if the exception is thrown. Figure 4.2 (page 58) shows a simple method in Grimp format that has an exception table with just one entry.

The interesting line in figure 4.2 is the one beginning with the `catch` statement. This is not part of the method's program text, but rather is the single entry of this method's exception handler table. The entry states that if an exception of type `java.lang.RuntimeException` is thrown between `label0` (inclusive) and `label1` (exclusive), then the current instruction should be aborted and execution should begin immediately at `label2`. We say that the statements between these two labels are in an *area of protection*.

Our exception table entry means that if, for example, `i1` has the value 0, the divide by 0 at `label0` will trigger a `java.lang.ArithmeticException`, which is a subclass of `java.lang.RuntimeException`, the exception table entry will be activated, the current instruction will be aborted, `i0` will not be modified, and `r2 := @caughtexception;` will be immediately started.

Figure 4.2 comes from a compiled Java program, and hence exhibits an easy to decompile pattern. Obviously, the statement `i0 = i0 / i1;` should be put in a `try` statement and the statements between `label2` and `label3` would go in the `catch` statement as figure 4.3 (page 59) illustrates.

Unfortunately, the specification of the exception table does not place any constraint on where the handler target is located. The handler target may proceed the area of protection or even be located inside the area of protection. Consider the contrived, hand coded Grimp method in figure 4.4 (page 60) and the abstract control flow graph for this program in figure 4.5 (page 61).

```
public int m(int int )
{
    mException r0;
    int i0, i1;
    java.lang.RuntimeException r1, r2;

    r0 := @this;
    i0 := @parameter0;
    i1 := @parameter1;

 label0:
    i0 = i0 / i1;

 label1:
    goto label3;

 label2:
    r2 := @caughtexception;
    r1 = r2;
    java.lang.System.out.println("RuntimeException thrown");

 label3:
    return i0;

    catch java.lang.RuntimeException from label0 to label1 with label2;
}
```

Figure 4.2: Grimp representation of simple method `m()` with exception.

There are two reasons why the two areas of protection cannot simply be represented by two `try - catch` statements.

1. When considering the exception handled by `b` we see that `b` is protected by itself, and therefore that some of the exceptional edges targeting it are forward edges while others are back-edges. Since all the exceptional edges in a `try - catch` statement are forward edges, a single `try - catch` statement cannot represent this area of protection. An identical argument can be made for the exception handled by `d`.

2. The set of statements that are in `b`'s area of protection {`a,b,c`} and the set of statements that are in `d`'s area of protection {`b,c,d,e`} intersect, but neither is a subset of the other. If we were to try to represent this with only two `try -`

```
public int m(int i0, int i1)
{
    try
    {
        i0 = i0 / i1;
    }
    catch (java.lang.RuntimeException r2)
    {
        r1 = r2;
        java.lang.System.out.println("RuntimeException thrown");
    }
    return i0;
}
```

Figure 4.3: Code showing decompilation of Figure 4.2

`catch` constructs, we would be left with the impossibility that one of the `try - catch`s would be nested in the other to accommodate the non-empty statement set intersection, and yet not nested in the other to accommodate the fact that its statement set isn't a subset of the other's.

Therefore only way to proceed is to perform a transform on the graph that will guarantee that we can take any valid exception handling and transform it to a form that can be later be structured with `try - catch` statements.

4.2 Exception Table Entry Removal

The Java language specification states that for a `try` block to catch a certain exception, it must be possible to statically prove that that exception *may* be thrown. Although we will remove spurious `catch` and `try` blocks as a last step, we begin with an optional preprocessing step which splits certain types of areas of protection and removes provably useless exception handling to reduce the number of effects due to off-by-one errors in the exception table[1], and simple obfuscation by the addition of extra areas of protection.

The first step is to examine every area of protection to see if it is *self-targeting*,

[1]A compiler writer may misinterpret the inclusive/exclusive properties of the boundary statements for the area of protection.

```
public void m()
{
    mException r0;
    java.lang.RuntimeException r1;
    java.lang.Throwable r2;

    r0 := @this;

  label_a:
    java.lang.System.out.println("a");
    goto label_c;

  label_b:
    r1 := @caughtexception;
    java.lang.System.out.println("b");

  label_c:
    java.lang.System.out.println("c");
    goto label_e;

  label_d:
    r2 := @caughtexception;
    java.lang.System.out.println("d");

  label_e:
    java.lang.System.out.println("e");

  label_f:
    java.lang.System.out.println("f");

  catch java.lang.RuntimeException from label_a to label_d with label_b;
  catch java.lang.Throwable from label_b to label_f with label_d;
}
```

Figure 4.4: A complex interaction of exception table entries.

and if it is, whether it is *easily divisible*. Self targeting means that the handler for the area of protection is inside the area of protection itself. Easily divisible simply refers to whether the handler dominates all statements that it can reach in the area of protection, without having to traverse statements outside that area of protection. If the area of protection is self targeting and easily divisible, we split it into two areas of protection that share the same handler. The split is done in three sub-steps.

1. We create a duplicate of the area of protection. This corresponds to creating a

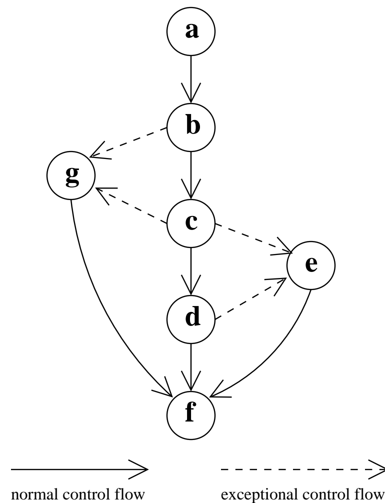


Figure 4.5: Abstract Control Flow Graph of Figure 4.4.

new entry in the bytecode's exception table, and does not entail the duplication of any method statements.

2. We remove the handler statement and all statements dominated by the handler from the original area of protection.

3. We remove all the statements remaining in the original area of protection from the duplicate. In other words, the duplicate now just contains the handler and those statements it dominates. Figure 4.6 shows an abstract statement sequence being split.

The second step is to check each area of protection to see if it is provably unnecessary. For area of protection A with exception type t, if t is not a super or sub-class of `java.lang.RuntimeException`, then we check if we can remove A. To do this we scan all the `throw` and `invoke` statements in the body of A to see if they can be the source of an exception of type t. If there are no such statements, then we remove A from the method. Again, as with splitting, this corresponds to removing an entry from the exception handling table, and does not otherwise affect the body of statements in the method.

In an aside one should note that removing exceptional control flow edges may produce dead code. This is easily checked for by giving each handler an *entry count*.

```
L0:  a;
     b;
L1:  c;
     d;
L2:  e;

catch from L0 to L2 with L1
```

(a) Original code

```
L0:  a;
     b;
L1:  c;
     d;
L2:  e;

catch from L0 to L1 with L1
catch from L1 to L2 with L1
```

(b) After split.

Figure 4.6: Splitting an exception table entry.

This is just the number of areas of protection that use the handler instruction as their exception target. For example, if two areas of protection use the same instruction in the bytecode as their handler, that handler will have an entry count of two. A split will increment a handler's entry count while a removal decrements it. When a handler reaches a zero entry count it is checked to see if it has any predecessors from normal non-exceptional control flow, and if not, is declared to be dead code and is removed from the method body, along with all statements that it dominates.

The intuition behind this preprocessing comes from two issues. First, there are off-by-one errors in the bytecode as a result of buggy compilers which may include the exception handler in the area of protection by accident. This is often a benign bug that doesn't alter the program's behavior because the first instruction of the handler usually is incapable of throwing the covered exception type. This is, however, a specific instance of a more general intuition which is that it is not likely that the handler of an exception will cause that same exception to be thrown. *If* the handler's area of protection can be removed, we may well modify the area of protection to resemble a `try` block and reduce the amount of complexity in restructuring later on.

The second issue come from obfuscation whereby extraneous areas of protection are randomly injected into a method's exception table. Since the program's control flow depends on these exceptions *not* being thrown, the obfuscater must be able to prove that this is the case. For example, a runtime exception such as `java.awt.image.RasterFormatException` could be inserted anywhere so long as there were no references to `java.awt.image.Raster` objects within the program. The goal of our decompiler is to weed these areas of protection out as soon as possible so that they do not impact on program structuring in building the SET. The level of sophistication that one employs in the proof of not throwing will allow for more

or less successful decompilation. Current Dava uses the fairly naive rule of thumb that the exception is only removed if it would cause `javac` to fail compilation of the resulting source code. In other words, we only remove areas of protection if it is statically provable that the area of protection would cause recompilation to fail.

4.3 Exception Preprocessing

We now turn to the core of phase 1 in the overall decompiling algorithm. There are two problems in converting exception handling in Grimp to `try - catch` statements in Java. First, the `try - catch` statements must nest each other properly, and second, they must nest properly within the SET. We tackle these nesting issues by breaking exception handling into two phases: preprocessing and `try-catch` statement creation. Preprocessing deals with making sure `try - catch` statements nest each other properly.

4.3.1 Versioning on the Control Flow Graph

Although it may occasionally produce visually unappealing output, this solution is guaranteed to produce areas of protection and eventually `try-catch` blocks, that will nest each other properly. We begin by noting the following.

1. Like every Java language statement, every `try` statement has exactly one entry point.

2. The entirety of every `try` body is covered by any number of `catch` exception handlers, and at most one `finally` handler statement. These handlers must immediately follow the `try` statement.

3. `try - catch - finally` statement groups must either be nesting or non-intersecting with other `try - catch - finally` statement groups. That is, given a `try - catch - finally` statement group, the body of encapsulated statements from that group must either be disjoint or a subset or superset with respect to every other `try - catch - finally` group.

To accommodate these observations we built three simple transforms to the control flow graph. In each case we transform an unstructurable graph into a structurable one.

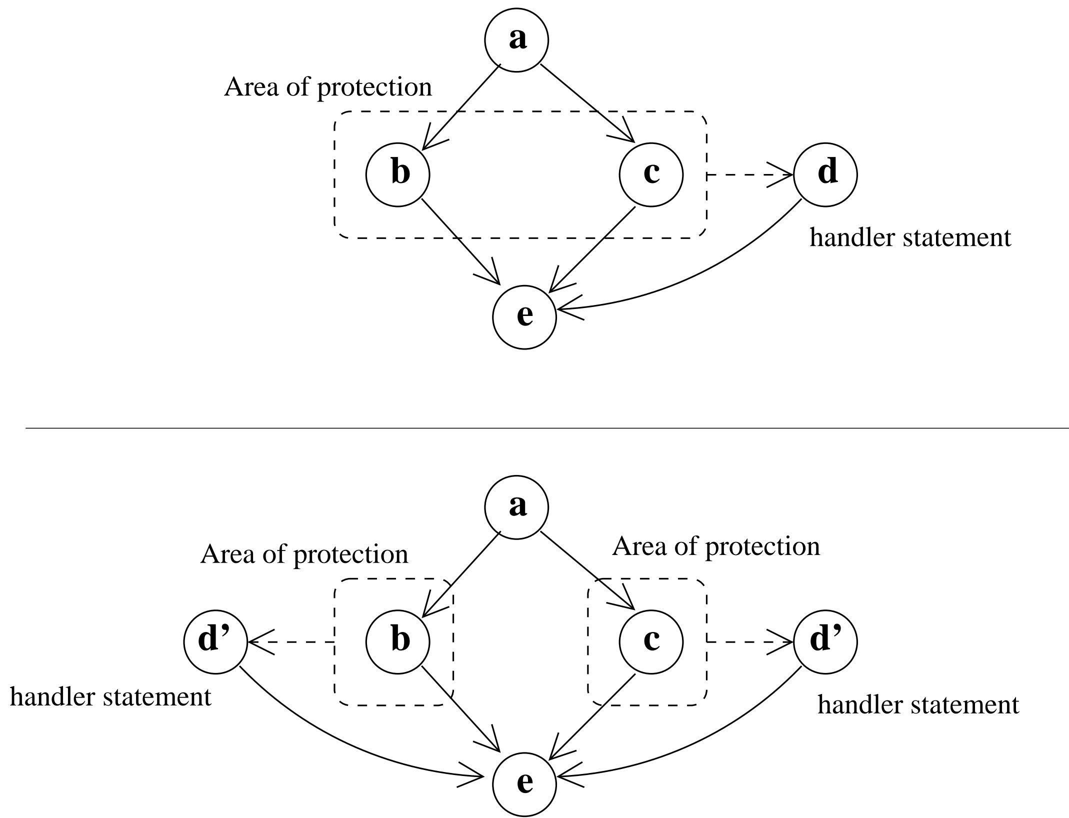
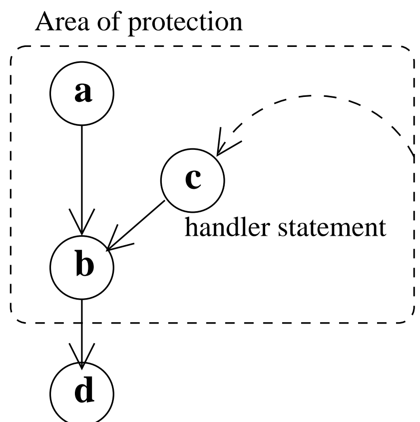


Figure 4.7: An area of protection with two entry points and its appropriate transform.

1. Given an area of protection with two entry points, we simply create a new area of protection for each entry point, and duplicate the exception handler for each new area of protection.

   Figure 4.7 (page 64) illustrates this transform. Here the area of protection has two entry points (b and c). To resolve this problem we create a version of the area of protection for each entry point. Because after this operation there are two distinct areas of protection, we have to clone the exception handler (d) so that each area of protection has its own handler (d’).

2. Given an area of protection where the handler statement happens to be within the area of protection, such as in figure 4.8 (page 65), we break the area of protection into two segments where in one segment the exceptional edges are forward edges and in the other segment they are back-edges. We then create a



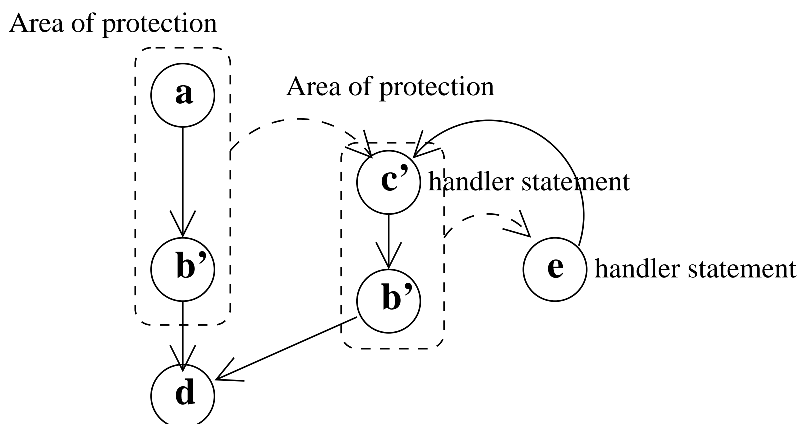


Figure 4.8: A self targeting area of protection and its appropriate transform.

new dummy exception handler for back-edged segment and use it convert the back-edges to forward edges. As we will later see, although this does not exactly match the original semantics of the program, the resulting Java program is still guaranteed to simulate the original correctly.

In figure 4.8 we are shown an area of protection where the handler is within the area of protection. First we create two versions of the area of protection for which the exceptional edges will either be forward edges or back-edges. Note that since the exceptional edge from `b` could be *either* forward or backwards, that two copies of `b` are created (`b'`), one for each direction. Next, we create a dummy handler statement `e` that has the original handler as its regular control

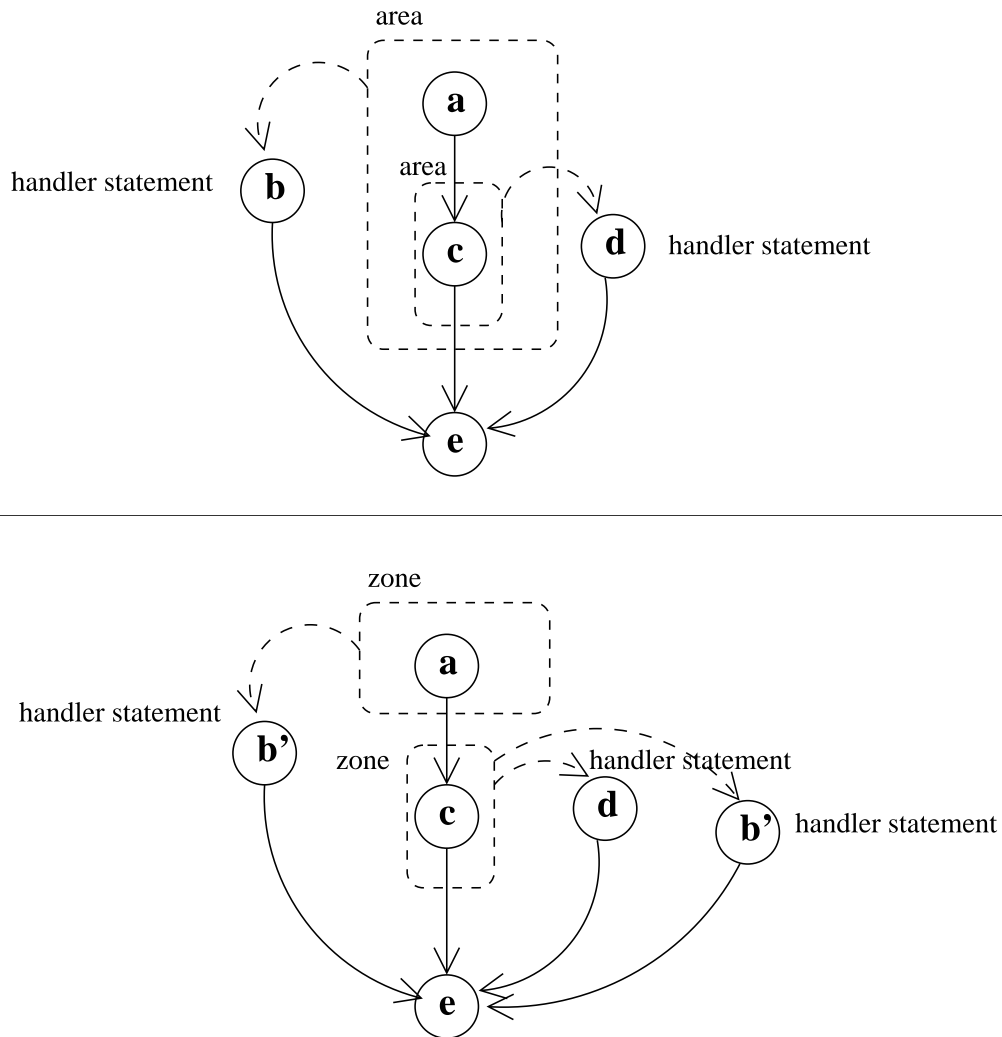


Figure 4.9: Two non-nesting areas of protection and their appropriate transform.

flow successor. Finally we put in the exceptional control flow edges. The result will now structure as a `try-catch` with an unconditional loop in the `catch` clause, which itself contains a `try-catch` statement.

3. Given one or more areas of protection, we define a set of protection zones. A zone is just like an area of protection, except that (like a `try` statement) it can handle more than one exception. In figure 4.9 (page 66), we show the creation of two zones from two areas of protection. Note that because zones can handle

more than one exception, we construct zones so that each zone is disjoint from all the other zones.

In figure 4.9 we have two non-disjoint areas of protection. We create protection zones so that statements in contiguous control flow have the same set of exception handlers protecting them. For example, `a` is only covered by the handler `b` while `c` is covered by two handlers, `b` and `d`. For this reason, `b` and `c` are placed in separate zones. These zones will be used to eventually create `try-catch` blocks, so once the zones have been constructed, the exception handler statements are cloned so that each zone has its own proper handlers. In this case `b` was cloned into `b'`.

These examples only outline a conceptual approach. Let's now examine see how these three transforms apply to the example from figure 4.5 (page 61). Figure 4.10 (page 68) shows the state of the graph after the application of Algorithm 6 (page 69) which employs these three transforms. Although the resultant may appear somewhat bewildering at first, it's construction is fairly simple and is discussed next.

4.3.2 Basic Exception Preprocessing

Basic exception preprocessing is done in algorithm 6, **exceptionPreprocess** (page 69). We begin by looking at lines 1 to 4 which form the basis of the versioning approach. Lines 5 through 7 (inclusive) improve the structure of some correct but awkward artifacts in the output, and lines 8 through 11 deal with patching the newly created graph back into the original control flow graph. The task of the first four lines is to build *zones* which have two important properties. First, a zone holds a set of statements that will eventually be made into the body of a Java `try` statement, and second, it has a set of one or more `catch`s. Algorithm 7 **findTryZone** (page 70) show the initialization of a zone.

The first line of Algorithm 6 **exceptionPreprocess** sets up the self targeting set. The members of this set act as stopping conditions for when we find the bodies of try blocks. There are other stopping conditions, that we add later, but this one is important because it eliminates the possibility that we generate a `try` block with more than one entry point. The second line defines the entry set. In general, the sets of statements under different areas of protection may overlap in complicated ways. To deal with this we version these statements for each control flow path they

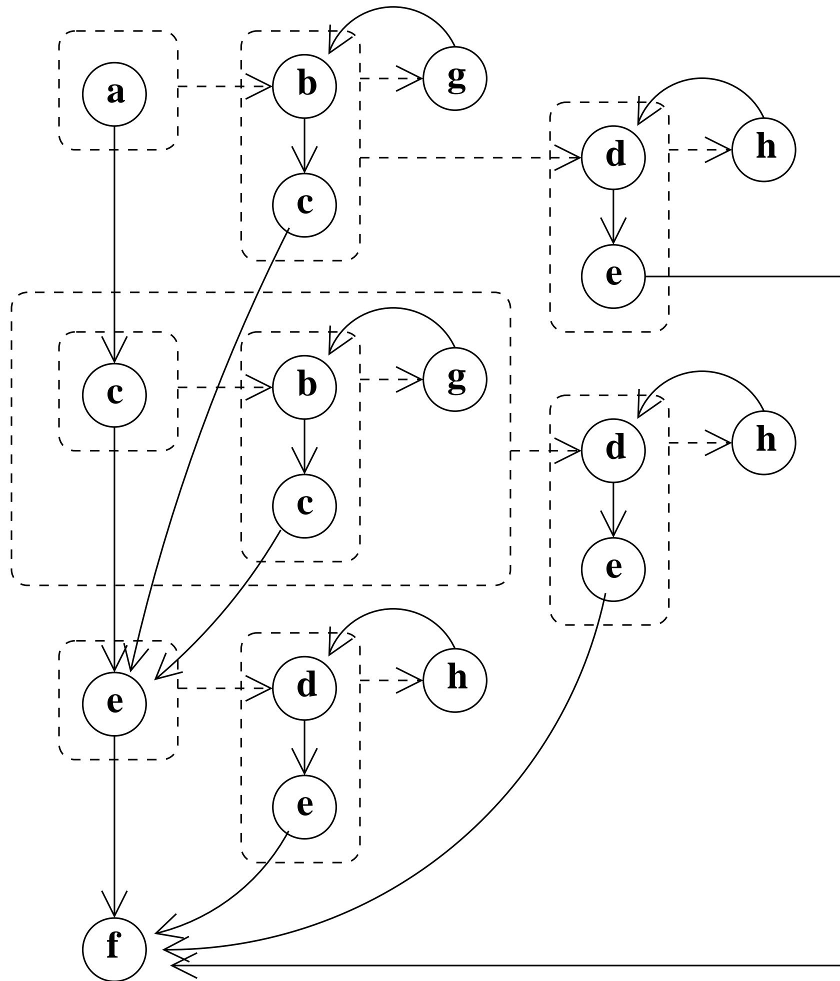


Figure 4.10: Application algorithm 6 to figure 4.5.

may occur on while under some area of protection. The entry point set, therefore, represents the start of each possible versioning control flow path. The third line simply initializes a *set* of zones of protection, while the fourth creates the zones with calls to **findTryZone**. We will cover the subsequent lines in sections 4.3.3 and 4.3.4.

Line 4 involves a call to Algorithm 7 **findTryZone** which we now turn to. Algorithm 7 finds zones, which are a first approximation to Java `try` statements. Line 1 is just initialization of several fields, the real work starts in line 2. Here we define an exception type to be the cross of the class of exception being caught and the target

Algorithm 6 exceptionPreprocess(Set *methodStatementSet*, Set *trapSet*)

1. Define the ***selfTargettingSet*** as the set of statements that are self targetting under exceptional control flow. Specifically, for each $t \in trapSet$ if t's handler statement is in t's area of protection, then t's handler statement is added to the *selfTargettingSet*.

2. Define the ***entrySet*** as the set of method statements that are in some trap's area of protection **and** are not dominated by any trap's handler statement **and** either 1) have no predecessors, or 2) have some predecessor that is not under any area of protection.

3. Define the ***entryZoneSet*** as the empty set.

4. For each statement $s \in entrySet$ do
   $entryZoneSet \leftarrow entryZoneSet \cup \{$ **findTryZone**(s, *trapSet*) $\}$

5. For each zone $z \in entryZoneSet$ do **integrateBranches**(z)

6. For each zone $z \in entryZoneSet$ do **integrateStem**(z)

7. For each zone $z \in entryZoneSet$ do **fuseFollowers**(z)

8. Define the ***exitSet*** as the empty set.

9. For each zone $z \in entryZoneSet$ do $exitSet \leftarrow exitSet \cup \{$ **wireTryZone**(z) $\}$

10. For each edge $e \in exitSet$ set the appropriate statement successor and predecessor information for the clones that are involved in e.

11. For each statement $s \in entrySet$ set the appropriate control flow edges from the original graph to s

Notes
findTryZone is Algorithm 7 (page 70)
wireTryZone is Algorithm 12 (page 80)

statement that does the catching. We then find all the exception types that cover the entry point to the currently created zone. These correspond to the catch statements that will handle the future try block. In line 3 we find the bodies of what will become `catch` statements in the Java representation. Note that since a Java `catch` statement can embed further `try` statements, the call to **findCatchZone** must recurse back

Algorithm 7 findTryZone(Statement *entryPointStatement*, Set *trapSet*)

1. Create a new ***tryZone***.
   $tryZone.entryPoint \leftarrow entryPointStatement$
   $tryZone.exceptionSet \leftarrow \emptyset$
   $tryZone.body \leftarrow \emptyset$
   $tryZone.catchSet \leftarrow \emptyset$
   $tryZone.followerSet \leftarrow \emptyset$
   $tryZone.embeddedTryZoneSet \leftarrow \emptyset$

2. From the *trapSet* find the set of exception handlers that protect the *entryPointStatement*. An exception handler is defined to be the pair of the type of exception being caught and the handler statement. Assign the set of exception handlers to the *tryZone.exceptionSet*.

3. For each Statement $s \in tryZone.exceptionSet.handlerStatement$ do
   $tryZone.catchSet \leftarrow tryZone.catchSet \cup \{$ **findCatchZone**(s) $\}$

4. Find the body of the *tryZone*. To do this, traverse the control flow graph by following statements' control flow successors starting from the *tryZone*'s entry point statement. Do not traverse 1) successors that are protected by a different set of exception handlers as the entry point, 2) members of the self targeting set, or 3) statements that have already been traversed for the current *tryZone*. For each statement that is traversed, make a new clone of that statement and put it in the *tryZone.body*. For each non-traversed statement due to reasons 1 or 2, s, do $tryZone.followerSet \leftarrow tryZone.followerSet \cup \{$ **findTryZone**(s) $\}$.

5. Return the *tryZone*.

note: **findCatchZone** is Algorithm 8 (page 71)

into **findTryZone** to find embedded `try`s. Line 4 finds the body of the future `try` statement. Since a `try` is wholly under protection from all of its `catch` statements, each statement in the corresponding zone must be under protection by the same set of exception types as is the zone's entry point. At the same time we find a zone's followers. A follower is simply a zone that is only accessible from the current zone, and since we are producing versioned control flow, any zone that is immediately entered after the current zone is one of the current zone's followers.

Algorithm 8 **findCatchZone** (page 71) finds catch zones which correspond to

Algorithm 8 findCatchZone(Statement *entryPointStatement*, ExceptionType *exType*, Set *trapSet*)

1. Create a new ***catchZone***.
   $catchZone.entryPoint \leftarrow entryPointStatement$
   $catchZone.exType \leftarrow exType$
   $catchZone.embeddedTryZoneSet \leftarrow \emptyset$

2. Find body of the *catchZone*.

   (a) If the *entryPointStatement* is the entry point for a previously encountered *tryZone* create a dummy handler statement which loops back to the *tryZone* and return. This will take care of exception generated loops in the control flow graph.

   (b) Take the *S*, the set of statements that are dominated by the *entryPointStatement*. Let set $T \leftarrow S$

   (c) For each $s \in S$, find the set of statements *EP* that are in some trap's area of protection **and** either 1) have no predecessors, or 2) have some predecessor that is not under any area of protection.

   (d) If *entryPointStatement* is in some area of protection, then add *entryPointStatement* to *EP*.

   (e) For each $s \in EP$, find the set of statements U using the same mechanism *w.r.t.* s as a *tryZone* body is found. $T \leftarrow T \cup U$. $catchZone.embeddedTryZoneSet \leftarrow catchZone.embeddedTryZoneSet \cup$ { **findTryZone**(s, *trapSet*) }

   (f) $catchZone.body \leftarrow T$

3. Return the *catchZone*.

Notes
findTryZone is Algorithm 7 (page 70)

future `catch` statements. Line 1 initializes a few basic bookkeeping fields. Step 2 finds the body for a catch zone and combines two intuitions to do its job. The first in 2a, is that if we have already encountered the exception handler statement as an entry point to a previously created try zone, instead of finding a new body for the `catch`, we can simply make a dummy handler that loops back to the previously created zone with regular control flow. In Java, this will end up looking like a `continue` on a loop

from an empty `catch` statement. The second is that the body of the `catch` should be those statements that are dominated by the exception handler. We have to be careful, though, because with versioning, a statement that is not dominated by the handler in the original graph may be dominated in the versioned graph. Line 2e provides the specification for how this is determined. Note that we also call **findTryZone** to find any `try` statements that may exist inside the `catch` that we are creating.

4.3.3 Improving the Versioned Control Flow Graph

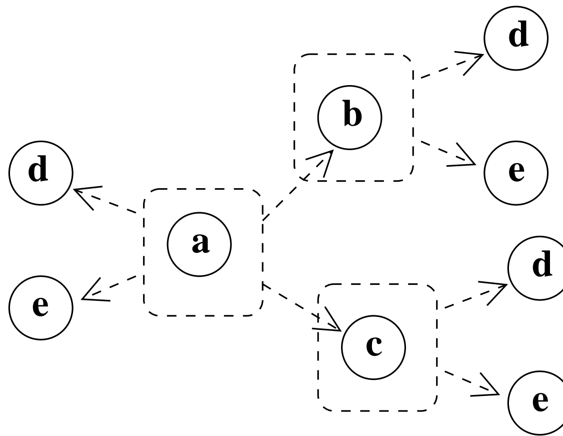
Although we now have built a *zone* based representation for the control flow graph that is guaranteed to be representable with `try` statements, many common idioms of exception use will produce ugly results. Simply nesting one `try` statement in another will induce the restructurer to produce up to four separate `try` statements where only two were necessary. Worse still these four can greatly obfuscate the output. To reduce this effect we have introduced three simplifying procedures, branch integration, stem integration and condition fusing, that recover much of the correct natural form of the original bytecode.

When viewing figure 4.10 (page 68), you can see that the exception preprocessor produces a plant-like structure with stems of regular control flow, for example edge `(a,c)`, and branches of exceptional control flow, for example edge `(a,b)`.

Branch Integration

Branch integration allows us to get simple nested `try` statements from certain types of `try-catch` chains.

Figure 4.11 (page 73) illustrates branch integration. The operation is done on the zone for node `a`. The handlers for this zone are divided into two sets, one which holds those exception handlers which will be put into an encapsulating zone (this is the set of outer handlers) zone and one which will be put into an encapsulated zone (this is the set of inner handlers). In our example, the outer set is {`d,e`} and the inner set is {`b,c`}. The important property is that all members of the inner set have exception handlers for all members of the outer set. If it is not possible (because of the exception handling properties of each handler) to divide a zone's handlers into these two sets, then branch integration is abandoned.



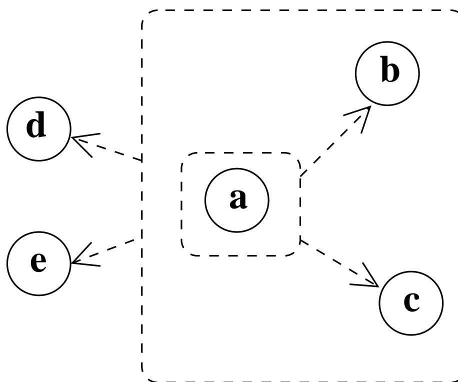
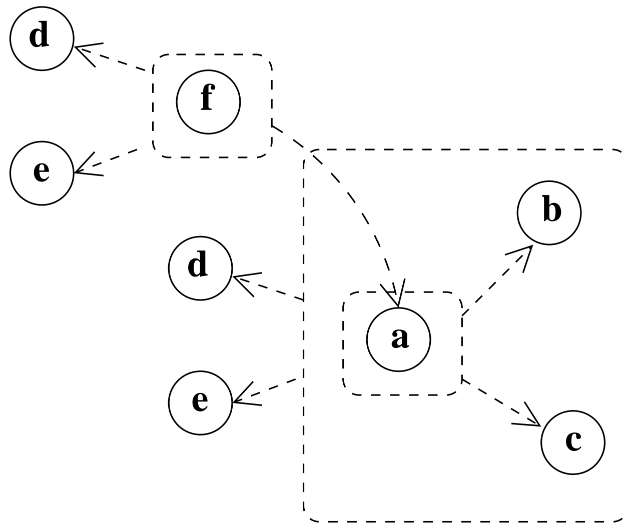


Figure 4.11: A simple example of branch integration.

Figure 4.11 shows an example that will yield one level of zone nesting once integration is performed. It is possible that there could be more than one level of zone nesting. Consider figure 4.12 (page 74) as an extension to figure 4.11.

Here we have simply added on a predecessor (statement `f`) to statement `a`. Since the handlers of `f` can be split into the necessary sets, we can perform the branch integration. This means that branch integration is best done as a bottom up procedure from the ends of the exceptional branches back to the stem. A high level algorithm follows in Algorithm 9 **integrateBranches** (page 75).



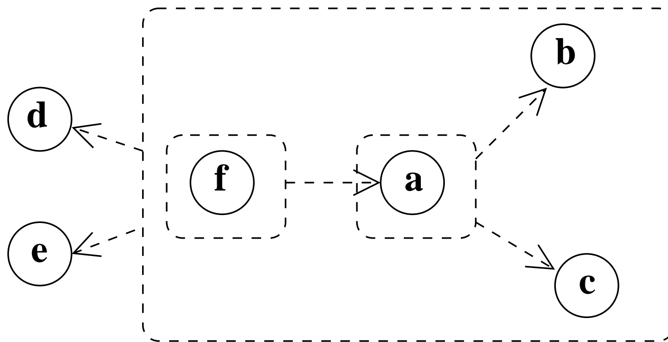


Figure 4.12: An more complex example of branch integration building on figure 4.11.

Stem Integration

Although we will recover all proper nestings of zones with branch integration, direct translation to `try` statements from zones may still produce awkward code from common motifs in the bytecode. Consider the top half of figure 4.13 (page 76); this will decompile to the code given on the left of in figure 4.14 (page 77). Intuitively, it is easy to see that code on the right is equivalent, and more natural and readable. Stem integration, as illustrated in figure 4.13 will produce this second version from the first. The high level algorithm for stem integration is given in Algorithm 10 **integrateStem** (page 77).

Algorithm 9 integrateBranches(TryZone *currentZone*)

1. For each zone $z \in currentZone.followerSet$ do
   integrateBranches(z)

2. For each zone $z \in currentZone.catchSet$ do
   integrateBranches(z)

3. If possible, partition *zone.catchSet* into the *innerSet* and the *outerSet*. The *outerSet* will provide exception handling for each member of the *innerSet*. All members of the *innerSet* must have exception handlers for all members of the *outerSet*. If it is not possible (because of the exception handling properties of each handler) to divide a zone's handlers into these two sets, then branch integration is abandoned for the *currentZone*

4. Perform the integration. The *currentZone* is expanded to encapsulate the *innerSet* and the *innerSet*'s handlers are removed from it. Next the handlers from the members of the *innerSet* are removed. Finally, a ***newZone*** is created that holds just the original *currentZone* and hooks it to the handlers from the *innerSet*.

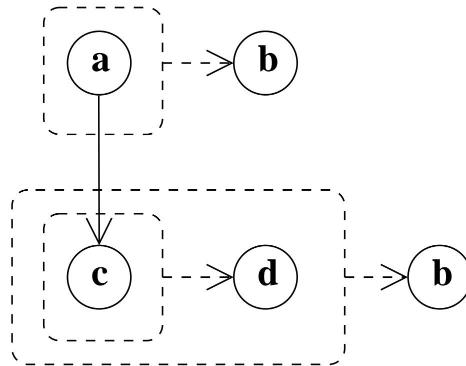
5. Call **integrateBranches**(*newZone*)

Follower Fusing

One of the chief strengths of the exception pre-processor's algorithm is that versioning control flow traces allow us to ignore statements' predecessor information. While this makes our procedure simple and robust, an unfortunate side effect is that we may produce more versions of code than we actually need. Consider figure 4.15 (page 78). Here we have produced two versions of `g`'s zone where only one version was necessary.

A similar motif will also commonly occur from `if - else` statements that are nested within a `try` statement where each clause contains it's own nested `try` statement.

This is easily remedied by Algorithm 11 **fuseFollowers** (page 79). The intuition behind this algorithm is to define a zone by its body. We then keep a set of zones, such that if we find two or more zones with equivalent bodies, only one is kept in the set, and the others are removed.



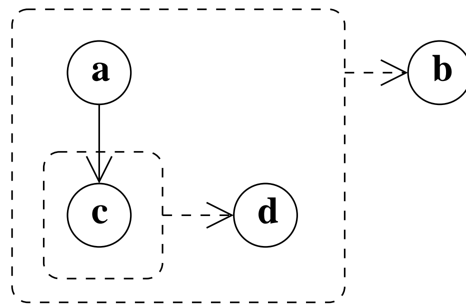


Figure 4.13: A simple example of stem integration.

4.3.4 Putting it all together with Control Flow

So far, all operations we have dealt with are performed on sets of statement clones. Throughout we have seen control flow between the *original* statements but only now do we build the control flow graph between the clones. The overall graph building strategy is to recursively build the follower set zone graphs, then the embedded try zone graphs, then the graph of statements that is not embedded in any try zone and then to link the three together. Algorithm 12 **wireTryZone** (page 80) starts off this process by establishing the control flow within a *zone*.

By construction, in any zone body, there will be at most one non-embedded clone statement for any given original statement. That is, given that we restrict ourselves to looking only at those statements in a zone which do not belong to a nested zone, there will never be two statement clones with the same original statement. Because of

```
try {                       try {
    a;                          a;
}                               try {
catch() {                           c;
    b;                          }
}                               catch() {
try {                               d;
    try {                       }
        c;                  }
    }                       catch() {
    catch() {                   b;
        d;                  }
    }
}
catch() {
    b;
}
```

Figure 4.14: Code corresponding to stem integration in figure 4.13.

Algorithm 10 integrateStem(TryZone *currentZone*)

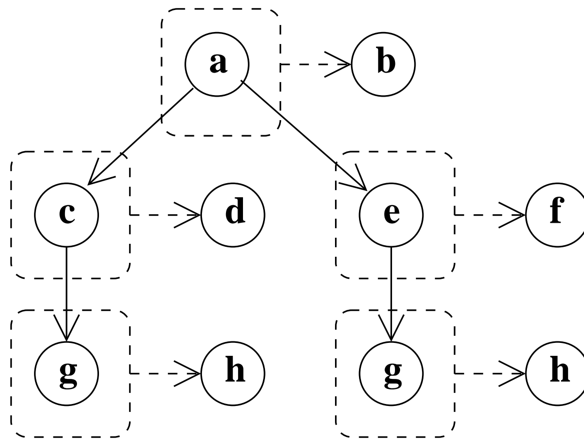
1. For each zone $z \in currentZone.followerSet$ do
   integrateStem(z)

2. For each zone $z \in currentZone.catchSet$ do
   integrateStem(z)

3. If all members of the *currentZone.followerSet* have the same exception handling as the *currentZone* does, then perform the stem integration. Figure 4.13 (page 76) shows the intuition. Extend the *currentZone* to encompass all of the *currentZone.followerSet*'s members and remove the *currentZone*'s handlers from them.

this we can simply mirror the original control flow graph within this set of statements.

The successors to these statements may be the entry points to follower zones. If so, we know again by construction, that among the set of follower zones there will never be two with entry points that have the same original statement. We therefore set up edges between the current zone and whatever follower zones exist. As well by



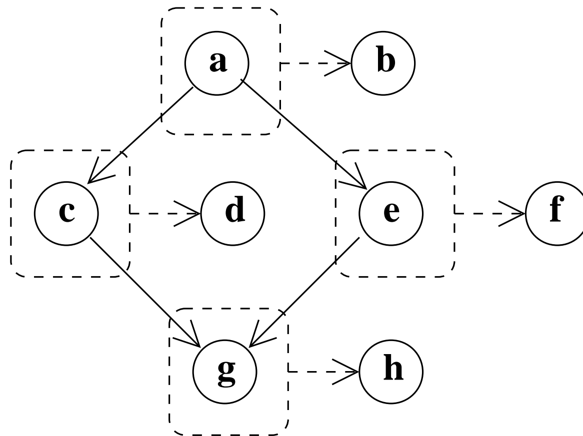


Figure 4.15: A simple example of follower fusing.

construction, we know that there will not be two embedded zones that have the same entry point. Since each embedded zone has only one entry point we therefore can link the current zone's statements to the embedded zone graph.

Finally, there may be successors which are not entry points to either follower zones or embedded try zones. These successors may be members of some parent try zone in the embedding hierarchy, or a member of the original graph that is not under any area of protection. These unknown edges are recursively passed back, evaluated, and eventually used to link into the modified control flow graph.

Algorithm 11 fuseFollowers(TryZone *currentZone*)

1. Define the identity of a *zone* as the set of original statements that correspond to the *zone.body*.

2. Define the ***curFollowerSet*** as the empty set.

3. For each zone $z \in currentZone.followerSet$
   $tmpSet \leftarrow$ **fuseFollowers**$(z) \cup \{ z \}$
   For each zone $fz \in tmpSet$
   if $fz \notin curFollowerSet$
   $curFollowerSet \leftarrow curFollowerSet \cup \{ fz \}$
   else
   remove *fz*

4. Return the *curFollowerSet*

Briefly for Algorithm 12 **wireTryZone** (page 80): Lines 1 - 3 initialize a few empty sets. Lines 4 and 5 set up the control flow edges in the `catch` and following `try` bodies, while also recording edges from these that had not been set up. Line 6 finds the set of statements that are not encapsulated by a nested `try`. Line 7 sets up the *testSet*. This is the set of all statements that *may* target some statement in the current *tryZone*. Line 8 then sets up the appropriate edge, if possible. Finally, line 9 gathers the set of all edges that could not be set up, which is then passed back to the caller.

Algorithm 13 **wireCatchZone** (page 81) sets up the control flow for a catch zone which, in fact, is just a simpler case of building the control flow for a try zone. The difference is that catch zones do not have a follower set, and we can avoid some of the extra complication.

Once this is algorithm is complete, it will have set up the control flow graph for the zones and their catch zones. The remaining task is to integrate the zones back into the original control flow graph. To do this we simply remove the original statements that have been used to generate the zones from the graph and put the zones in their place. Algorithm 6 **exceptionPreprocess** (page 69), lines 8 to 11, does this final step.

Algorithm 12 wireTryZone(Zone *currentZone*)

1. Define the ***exitSet*** as the empty set.

2. Define the ***followerExitSet*** as the empty set.

3. Define the ***catchExitSet*** as the empty set.

4. For each zone $z \in currentZone.followerSet$ do
   $followerExitSet \leftarrow followerExitSet \cup$ **wireTryZone**(z)

5. For each catch zone $cz \in currentZone.catchSet$ do
   $catchExitSet \leftarrow catchExitSet \cup$ **wireCatchZone**(cz)

6. Define the ***nonEmbeddedStmtSet*** as the set of statements in the *currentZone.body* which are not in any of the *currentZone.embeddedTryZoneSet*'s zones' bodies.

7. Define the ***testSet*** as the union of the *catchExitSet* and the *nonEmbeddedStmtSet*.

8. For each Statement $s \in testSet$ find the appropriate successor c. There are four possibilities.

   (a) c is in the *nonEmbeddedStmtSet*

   (b) c in an entry point to a member of the *currentZone.embeddedTryZoneSet*

   (c) c is an entry point to a member of the *currentZone.followerSet*

   (d) c escapes the *currentZone* and we have not yet seen it.

   If c is either of the first three cases, we simply set up the directed edge between s and c. Otherwise we place s in the *exitSet*.

9. $exitSet \leftarrow exitSet \cup (followerExitSet$ - { statements found in step 8c })

10. Return the *exitSet*

4.4 Exception Handling

Because we have already done heavy pre-processing of exception handling in the control flow graph, our task of representing exceptions with `try-catch` blocks is simplified. Since there are no inter or intra-exception handling representation problems,

Algorithm 13 wireCatchZone(CatchZone *currentZone*)

1. Define the ***exitSet*** as the empty set.

2. For each zone $z \in currentZone.embeddedTryZoneSet$ do
   $exitSet \leftarrow exitSet \cup$ **wireTryZone**(z)

3. Define the ***nonEmbeddedStmtSet*** as the set of statements in the *currentZone.body* which are not in any of the *currentZone.embeddedTryZoneSet*'s zones' bodies.

4. For each Statement $s \in$ *nonEmbeddedStmtSet* find the appropriate successor c. There are four possibilities.

   (a) c is in the *nonEmbeddedStmtSet*

   (b) c in an entry point to a member of the *currentZone.embeddedTryZoneSet*

   (c) c escapes the *currentZone* and we have not yet seen it.

   If c is either of the first two cases, we simply set up the directed edge between s and c. Otherwise we place s in the *exitSet*.

5. Return the *exitSet*

---

we only have to consider reconciling the newly created `try-catch` blocks with the existing constructs in the SET.

To begin with, a `try-catch` block is directly created from a single try zone and its corresponding catch zones. When nesting the `try-catch` block in the SET, we use the entry point of the `try` clause to decide which SET node we should be considering as either child or ancestor. While nesting the new block into the SET, we may have one of two nesting problems.

1. One or more of the `catch` clauses' *bodySet*s may not nest. The solution to this is to simply remove the non-nesting statements from the offending catch clause. These statements will then end up in some statement sequence following the `catch` clause.

2. The *bodySet* of the `try` clause may not nest. In this case we have to split the `try` clause into as many parts as necessary as to perform the nesting.

This second possibility carries a few complications.

1. Each newly split off `try` clause will need its own set of `catch` clauses. Accordingly, a full set catch clauses are cloned from the original `try` clause for each of the new `try` clauses.

2. If any of these cloned `catch` clauses contains any exception handling that exception handling must also be cloned.

3. The SET may indicate that some already found constructs nest within the cloned `catch` clauses. These constructs must also be cloned so that the contents of the newly created `catch` clauses gets properly filled in.

4.4.1 Spurious `catch` and `try` Removal

As stated in section 4.2, if we can statically prove that the exception will *not* be thrown, the presence of the spurious `catch` will be flagged by `javac` as dead code and generate a compile time error. Although the preprocessing described in section 4.2 will aid in the restructuring, it is strictly optional. The phase described in this section works on the completed SET and acts as a fall-back mechanism that removes all `catch` and `try` blocks from the SET that would halt recompilation.

Our goal is to find out which `catch` clauses cause trouble. To do this we first put all the `catch` clauses' exception types in a set C. Second, we remove from C any subclass or superclass of `java.lang.RuntimeException`. These exception types can always be caught. Third, we scan the body B of the `try` block. For every statement $s \in B$ we obtain the set of exceptions E that could be thrown from s. For every exception type $e \in E$ if $e \in C$ then e is removed from C. When this has been done for all the statements in the `try` block, C will contain those exception types which cannot be thrown, but never the less have `catch` clauses. These `catch` clauses are now removed from the `try` statement. Finally, if the `try` has no remaining `catch` clauses, then it is removed from the SET and replaced with the statement sequence from the body of the `try` block.

The remaining issue is to determine what the exception types can be thrown from a statement. This is a recursive process that runs on each type statement in the SET. We break statements into two types: simple statements and control flow statements. Within simple statements, only `invoke` statements and `throw` statements

are examined, all others are ignored as they can only produce runtime exceptions. For `invoke` statements we look at the invoked method's `throws` attribute, and for the `throw` statement we simply take the thrown exception type.

Within control flow statements, the `try` statement gets special treatment. The exceptions that can be thrown from a `try` statement is the union of throwable exception types from the body of the `try` block, minus the union of the caught exception types, plus the union of the throwable exception types from all of the `catch` blocks. For all other control flow statements, the throwable exception types are the union of the throwable exceptions in any of their nested statements.

In summary, this analysis finds and removes all `catch` clauses that could be flagged by `javac` as dead code. If the resulting `try` statement has no remaining `catch` clauses, it too is removed and replaced with the statement sequence from the body of the `try` block.

Chapter 5

Idioms

The chapter breaks down into three sections: 1) converting structured Grimp to Java, to the production of recompilable code, 2) readability transforms performed on Grimp which are optional but help produce code that exhibits human programmer idioms, and 3) readability transforms performed on the SET, which are also optional but helpful. The individual phases within these sections are scattered throughout the decompiling algorithm. For simplicity, and the fact that the idiomatic phases do not impact on the structure of the algorithm, they are not explicitly shown in the algorithm's outline. Rather, we state where the idiomatic phases are placed when describing them.

5.1 Converting Structured Grimp to Java

There are many differences between structured Grimp and Java and we need to make several analyses and transformations throughout the various decompilation phases to produce Java. This section describes six different issues.

5.1.1 Simple Statements

While some Grimp statements print out as syntactically correct Java, many do not. To correct this we go back to the very start of our decompiling procedure and transform the Grimp statement list l before doing anything else. For every Grimp statement $s \in l$ we perform algorithm 14 (page 85) **convertStmts**().

For example, an identity statement in Grimp will print the ':=' assignment which does not fit the Java syntax. To remedy this we simply replace the Grimp identity statement with a Dava identity statement that prints the '=' assignment instead.

Algorithm 14 convertStmts(Statement s)

---

1. For every expression $e \in s$ do
   convertExpr(e)

2. If **syntaxCorrectJava**(s) is *false*
   Replace s with an object whose type is a subclass of s's Grimp statement. This subclass overrides the printing method with one that yields syntactically correct Java.

Algorithm 15 convertExpr(Expression e)

---

1. For every subexpression $e' \in e$
   convertExpr(e')

2. If **syntaxCorrectJava**(e) is *false*
   Replace e with an object whose type is a subclass of e's Grimp expression. This subclass overrides the printing method with one that yields syntactically correct Java.

With the exception of control flow statements that are still using `goto`s, the list of Grimp statements that we feed our restructurer will now exhibit correct Java syntax.

5.1.2 Converting `invokespecial <init>` to Constructor Calls

Java language constructors are represented in Java bytecode by `<init>` methods. In our entire decompilation algorithm, there are few situations where we can have legal bytecode that we can not decompile. One of these occurs in the conversion of `invokespecial <init>` statements in Java bytecode to explicit calls to a constructor in the Java language.

In the Java language, one can overload the constructor and this is often done to offer the object creator varying amounts of default initialization. A common idiom,

then, is to have overloaded constructors call one another as shown on the left of figure 5.1 (page 86). The default constructor (the one having no parameters) calls the second constructor and passes it a default value of 5.

```
class A extends B
{
    private int x;

    public A()
    {
        this( 5);
    }

    public A( int i)
    {
        x = i;
    }

    ...
}
```

(a) First Java constructor calls the second constructor

```
...

public void <init>()
{
    A r0;

    r0 := @this;
    r0.<init>(5);
    return;
}

...
```

(b) Grimp representation for first constructor

Figure 5.1: Calling one constructor from another in Java and the corresponding Grimp.

The complication is that the Java language specification states that only the first statement in the class's constructor can be a call to another constructor, whereas *any* statement in a bytecode method can be an `invokespecial <init>`. Furthermore, if the first statement of a Java language constructor is *not* a call to another constructor, then there is an *implicit* call to the super class's default constructor. This holds for all classes except for `java.lang.Object` which has no super class.

Because of these restrictions, it may not always be possible to represent all legal `invokespecial <init>` as the first statement in the constructor that we are decompiling.

Consider the legal bytecode given in figure 5.2(a). The key feature to this code is that there are two control flow paths where along one path the object is initialized with its own default constructor while on another it is initialized by `java.lang.Object`'s

```
Method A(int)
   0 aload_0
   1 iload_1
   2 iconst_5
   3 if_icmpge 12
   6 invokespecial #1 <Method A()>
   9 goto 15
  12 invokespecial #2 <Method java.lang.Object()>
  15 return;
```

(a) bytecode

```
public A( int i)
{
    if (i < 5)
        this();
    else
        super();
}
```

(b) decompiled code, invalid Java

Figure 5.2: An "impossible-to-decompile" constructor.

default constructor. The corresponding decompiled code, however, is not valid Java because the calls to the constructors are not the first statement.

There are solutions to this problem, but they are invasive and require whole program knowledge. An example would be to create a object factory for objects of class `A` and replace all instances of new object `A` creation with calls to factory. Next, `A`'s constructors are modified in such a way that all code prefixing their calls to other constructors is moved out to the object factory. Aside from requiring whole program knowledge, potential difficulties arise with this scheme in maintaining the proper order in field initializations.

Instead, we assume that we do not have whole program knowledge and search for a specific pattern of instructions to rebuild constructor calls. Algorithm 16 **isConstructorSafe** illustrates the search pattern. If this algorithm returns true we convert the appropriate `invokespecial <init>` to a constructor call, but if false we abort the entire decompilation. Throughout our tests, however, this simple algorithm has always returned true.

5.1.3 Converting the `clinit` Method to a `static` Initializer Block

There are many problems with converting a `clinit` method into a class `static` initializer block, and as with the conversion of constructors, our attempt to decompile can again be thwarted.

Algorithm 16 isConstructorSafe(Method m)

1. If m does *not* contain an `invokespecial <init>`
   Return **true**
2. If m is *not* an `<init>`
   Return **false**
3. If there is more than one `invokespecial <init>`
   Return **false**
4. If the `invokespecial <init>` is under an area of protection
   Return **false**
5. If the `invokespecial <init>` is the first statement which is *not* a definition statement
   (a) Mark the `invokespecial <init>` as a call to another constructor
   (b) Return **true**
6. Return **false**

```
class BadStaticInitializer
{
   static {
      f = this.getClass().getField( "nonExistantField");
      if (f == null) return;
      System.out.println();
   }

   static Field f;
}
```

Figure 5.3: A class with an illegal static initializer.

Consider the decompiled code in figure 5.3. There are three reasons why this is illegal Java sourcecode, which could have been produced by perfectly legal bytecode.

1. The `this` and `super` method references must not appear in the Java language

static block, while there are no such restrictions on object references in bytecode. Accordingly, the use of this in the first line would cause compile time error.

2. It is a compile time error if a non-Runtime exception can cause the static block to exit abruptly, while there is no such restriction on the <clinit>. A Runtime exception is an exception of type java.lang.RuntimeException or any subclass of it. Non-Runtime refers to any other type of exception. In our example, trying to retrieve the nonExistantField, so named because it doesn't exist, will trigger a NoSuchFieldException. The javac compiler will statically determine that it is possible to throw such an exception and produce a compile time error.

3. A return statement must not appear in the static block, while every method in bytecode is exited by a return, including the clinit.

Dava addresses two of the problems.

1. We make sure that every throw which throws a non-Runtime exception is caught, and that for every invoke any possible non-Runtime exceptions that could cause the invoked method to exit abruptly are caught.

2. All return statements are removed from the Grimp statement list representation.

Although the remaining issues will generate compile time errors, they are simply ignored due to the facts that 1) we have never encountered them in our test suites, and 2) that they would be very difficult to solve.

5.1.4 throws declarations

If a non-Runtime exception can cause the abrupt exit of a method, the Java language specification [9] states that the method must declare that it throws that exception. Unfortunately, the Java virtual machine specification [13] does not make any requirement about such declarations. As a result compiled class files may leave this attribute out. This turns out to be a problem especially for applications originally written in source languages other than Java.

```
class A
{
    public void foo( int x) throws MyException
    {
        if (x < 10)
            throw new MyException();
    }
}
```

Figure 5.4: An explicit `throw` statement causes a `throws` declaration.

Figure 5.4 above shows a class `A` with single method `foo`.

Here `foo` must declare that it `throws MyException` because of the `throw new MyException()` statement, assuming that `MyException` is not a subclass of `RuntimeException`. Since `foo` is now the potential source of a `MyException` any method that calls `foo` must either `catch` it or also declare that it too `throws MyException`. Class `B` (figure 5.5 below) illustrates this by constructing a new object of type `A` and calling its `foo`.

```
class B extends C
{
    public void goo( int x) throws MyException
    {
        A a = new A();
        a.foo( x);
    }
}
```

Figure 5.5: Method invocation may cause a `throws` declaration.

Now note that class `B` extends class `C`. If method `goo` in `B` is overriding a method `goo` from class `C`, the Java language specification requires that the signatures of the two methods must match. Although `goo` in `C` will never really throw a `MyException`, it must declare that it `throws MyException` for method signature agreement as shown in figure 5.6.

Accordingly, if `C`'s `goo` is called, whoever calls it must either handle `MyException` or declare it as well. This pollution of potentially thrown exceptions and their `throws` declarations can also carry through interfaces. For the purposes of the rest of this section, we include interfaces whenever we mention super and sub classes.

```
class C
{
    public void goo( int x) throws MyException
    {
        System.out.println( x);
    }
}
```

Figure 5.6: Inheritance may cause a `throws` declaration.

Since bytecode may carry incomplete or spurious `throws` attributes, we pessimistically discard all `throws` attribute information that comes from the application we are decompiling, and rebuild it from scratch. Library class files, such as those in the Java language APIs, and native methods are optimistically treated as correct because we are not decompiling them and therefore have no analysis of the behavior.

Rebuilding the `throws` declarations is done as a fixed point inter-procedural analysis which builds summary information for every application class' methods. The starting state for each method is to declare only those exceptions for which there are `throw` statements in that method. This is done is two steps, first by discarding the `throws` attributes for each application class method so that they declare no thrown exceptions, and second, by scanning each method for `throw` statements. If a `throw` statement is found, the type of exception thrown is added to the method's `throws` attribute.

The fixed point analysis is then performed on these methods. Basically, we begin with every method in an unsafe state, and flow the thrown exception types until all methods reach a safe state. To update a method's `throws` attribute we perform two confluence functions.

1. We scan all the super and sub classes' versions of the current method. The new `throws` attribute will be the union of the old `throws` and all the `throws` attributes of the super and sub classes.

2. We scan the body of the current method for invoke statements. Again, the new `throws` attribute will be the union of the old `throws` and all the `throws` attributes of every method that is invoked. There is one catch to this step, and that is that we have to check to make sure that invoked method is not in an

area of protection that catches the potentially thrown exception. If it is, the exception type that is caught is *not* added to the union.

If the attribute changes from these two functions, we mark every application super and sub class version of the current method, and all application methods that directly invoke the current method as unsafe. The current method is then optimistically marked as safe.

Because confluence is the union operator, we know that when a method is processed, its `throws` attributes will either remain the same, or be increased. Since there is a fixed, limited number of exception types, we know that this analysis must terminate with all methods' `throws` attributes reaching a stable state. Finally, for correctness we note that confluence expresses exactly the requirements for recompilablilty, and hence know that the resulting `throws` attributes represent the *minimal* set of declared exceptions for each method.

5.1.5 Throwing `null`

In some cases we are presented with a `throws` statement that throws a local that has a `null` type. What happens in the VM is that on dereferencing the local we will throw a `java.lang.NullPointerException` and never run the `throws`. The net effect is that we throw a new `NullPointerException`.

Unfortunately, we will generate a compile time error if we try to compile the statement `throw null;`. The solution is to replace the `null` local with a new `NullPointerException`.

5.1.6 Class literals

The `javac` compiler contains a strategy for dealing with class literals that does not conform to the Java language specification. Whenever a class literal is used, for example for class A, `javac` will automatically generate a method called `class$` in class A. Now if, for example in class B, you explicitly write a method called `class$`, `javac` will abort compilation presumably because you effectively interfere with the potential automatic creation of this method. From a decompilation point of view, the problem is that if the original Java source code used a class literal, the `class$` method will have been generated by `javac`. Consequently, our decompiler will pick

up the generated method and blindly produce source code with a `class$` method in it. Then when we try to recompile the decompiled code, `javac` will claim that the decompiled program is not compilable. There are two possible solutions to this problem.

- Recoginize that the method is part of a class literal and:
  1. Remove the `class$` method from the class.
  2. Perform a high level analysis to remove invokes to the method, and any other code introduced by `javac`.
  3. Replace the removed code with the class literal.
- Rename `class$` to some new unique name within the class.

Both solutions share the problem that they require whole program knowledge to change all references to the `class$` method. Although producing a more elegant result, the first solution also imposes the overhead that we have to analyze the method to make sure that it is exactly equivalent to the `javac` generated method. This may very difficult to perform since the method may have undergone optimization and obfuscation.

The potential also exists that the `class$` method was *not* generated by `javac`. For this reason the second solution must always be implemented, at least as fall back mechanism when the first solution fails. In Dava we have only implemented the second solution so far.

5.2 Readability Transforms Performed on Grimp

We present two algorithms performed on the Grimp representation that improve the readability of the output. The first, aggregating `if` statements, allows for complex conditions controlling loops and `if` statements, and aids in the formation of `for` loops. The second introduces `import` statements to the decompiled output, and allows classes to be referenced without explicit package names being given at every reference.

5.2.1 Aggregated `ifs`

Although Grimp aggregates expressions between simple statements, it is not able to aggregate expressions across conditional control flow. In figure 5.7(a) we are shown a control flow graph with three conditional control flow statements, `a`, `b` and `c`. The purpose of `if` aggregation is to reduce this conditional subgraph to a single conditional control flow statement with a simplified aggregated conditional expression `(a && b) || c` as shown in 5.7(d).

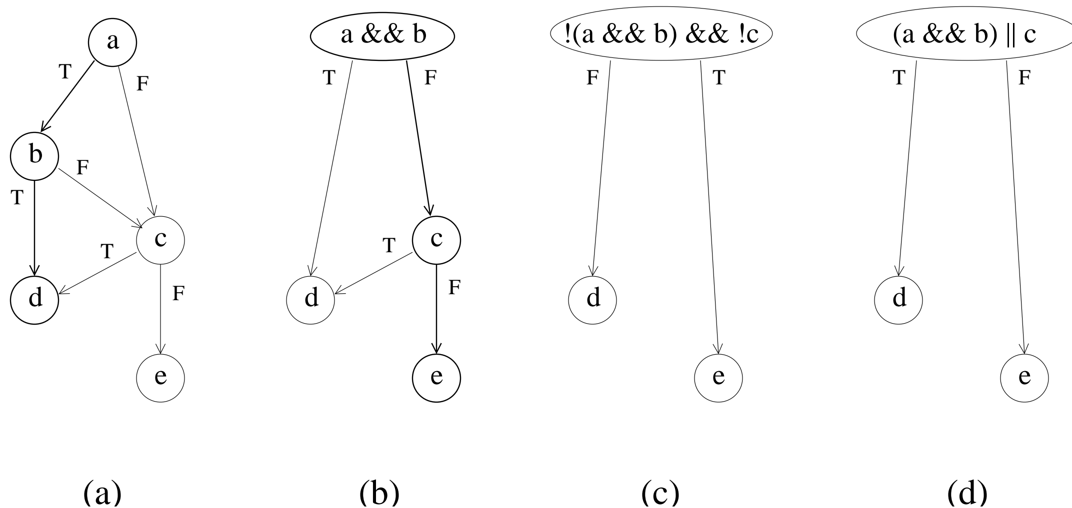


Figure 5.7: Reduction and simplification of a compound `if`.

In figure 5.7(a) we see how to reduce the subgraph of consisting of nodes `(a,b)` to a single node and produce the second graph, figure 5.7(b). The intuition is that to perform the reduction there must be a subgraph containing only two conditional statements which has only two destinations coming out of it. One destination (C) must be a successor of *only* both nodes in the subgraph, while the other destination (D) must be reachable by passing through *both* of the nodes in the subgraph. Destination D is then treated as the target of a compound conditional joined by the "and" (`&&`) operator and the reduction is performed, and C as the failure case. Finally, there must be only one entry point to the subgraph, and that entry point must be able to reach the rest of the subgraph.

Sometimes the logic of the control flow statements does not agree with creating this conjunction. The second control flow graph in figure 5.7, (b), shows just such a

case, where (`(a && b) && c`) is an obviously incorrect interpretation of the control flow logic. The goal we want is to have each edge on the target path to have the value `true` and the on the failure path `false`. If the current subgraph does not express this, we negate the conditional control flow statement's expression and exchange the truth values of its successor edges. The reduction is then performed, as shown in figure 5.7(c).

We use algorithm 17 to pattern match from statement s in a control flow graph to see if it is the root of a compound condition. If we get a return value of `true` then we can apply the compounding reduction to the subgraph rooted at s. To perform all possible compounding reductions we repeatedly iterate over the control flow graph, checking if reductions are possible, until no reductions are performed in an iteration.

Algorithm 17 isCompoundCondition(Statement s)

---

1. If s is not an `IfStmt`
   Return **false**
2. $succ_0 \leftarrow s.trueSuccessor$
   $succ_{fail} \leftarrow s.falseSuccessor$
3. If $succ_0$ is a successor of $succ_{fail}$
   swap($succ_0$, $succ_{fail}$)
4. If ($succ_{fail}$ is not a successor of $succ_0$) or
   ($succ_{fail}$ has more than 2 predecessors)
   Return **false**
5. If ($succ_0$ is not an `IfStmt`) or
   ($succ_0$ has more than 1 predecessor)
   Return **false**
6. Return **true**

Once we have finished reducing the subgraphs to compound conditional flow statements, we examine the resultant conditional expressions and minimize the number of negations in them. The intuition is that negation is generally hard to understand, and we want to present the conditional expression in the simplest form possible. Note that a boolean proposition in this expression may be a method call and therefore have program side effects. Because of this, we are restricted in the types of minimizing

transforms that can be performed on the expression tree.

The primary restriction is that we must preserve the original shape of the expression tree. Expression evaluation can be seen as a postfix operation on a depth first traversal of the expression tree. When we perform this traversal, then, we can produce a string recording the operations of the expression in the order that they were performed. Because the operands in the expression tree may be method calls, this string also uniquely identifies the order of potential side effects of the operands. For our program to retain its original semantics this ordering of side effects must be preserved through any transform that we may make to the expression tree. Since this order is unique to the shape of the tree, the transforms must not change the expression tree's shape.

Figure 5.8 shows a simple expression with method operands, the corresponding expression tree, and the ordering of potential side-effects based on the traversal of that expression tree.

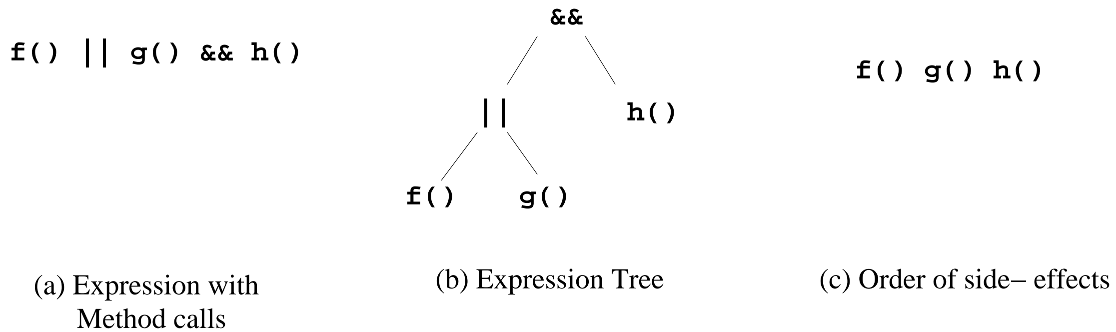


Figure 5.8: An Expression with potential side-effect and their ordering.

With this restriction in mind, Dava's minimization algorithm is made up of 2 phases and employs only the following safe transforms.

1. $(A) \equiv !(!A)$. On both sides of the equivalence, regardless of whether A is true or false, A will be evaluated. Introducing or removing double negation does not interfere with the order of the possible side effects of A.

2. The deMorgan's law which states that $(A||B) \equiv !(!A$ `&&` $!B)$. Consider if A is true, then A will be evaluated and B will not. This is obvious on the left, and we can see on the right that the `&&` will short circuit B from being evaluated.

If, however, A is false, first A is evaluated, then B is evaluated. Again this is obvious on the left, and also describes the behavior of the right.

3. The deMorgan's law which states that $(A \texttt{ \&\& } B) \equiv !(!A||!B)$. Consider if A is false, then A will be evaluated and B will not. This is obvious on the left, and we can see on the right that the `||` will short circuit B from being evaluated. If, however, A is true, first A is evaluated, then B is evaluated. Again this is obvious on the left, and also describes the behavior of the right.

After the description of the 2 minimization phases, we prove that this algorithm achieves the desired minimization of negations.

We begin with some arbitrary logical expression tree made up of "`and`" operators, constants and variables. A "`not`" flag may be attached to any sub-expression to negate it. The first phase is to push all the `not` flags in the expression tree down to the leaves. This is performed by algorithm 18 **pushNegationDown**.

Algorithm 18 pushNegationDown(Expression e)

1. If e is a Logical Constant **or** Variable
   Return
2. If $e.notFlag$ is $true$
   deMorgan(e)
3. **pushNegationDown**($e.leftOperand$)
   pushNegationDown($e.rightOperand$)

The second phase is to recursively traverse the resulting expression tree in a bottom-up manner, and for each sub-expression e obtain a "truth score" t_e.

$$t_e = \begin{cases} -1 & : \quad e \text{ is negated} \\ +1 & : \quad e \text{ is a positive leaf} \\ t_{e.rightOperand} + t_{e.leftOperand} & : \quad t_{e.rightOperand} + t_{e.leftOperand} < 1 \\ +1 & : \quad \text{otherwise} \end{cases}$$

This score will tell us when to apply deMorgan's law to the expression tree so we can minimize the overall number of negations in the expression. We claim that for any node e, if deMorgan's law is applied to e, then t_e is the maximal number of

Algorithm 19 deMorgan(Expression e)

1. If e is a Logical Constant **or** Variable
   Return

2. If $e.exprType$ is an `and` Expression
   $e.exprType \leftarrow$ `or` Expression
   else
   $e.exprType \leftarrow$ `and` Expression

3. If $e.notFlag$ is $true$ `// flip boolean value of notFlag`
   $e.notFlag \leftarrow false$
   else
   $e.notFlag \leftarrow true$

4. If $e.leftOperand.notFlag$ is $true$ `// flip boolean value`
   $e.leftOperand.notFlag \leftarrow false$
   else
   $e.leftOperand.notFlag \leftarrow true$

5. If $e.rightOperand.notFlag$ is $true$ `// flip boolean value`
   $e.rightOperand.notFlag \leftarrow false$
   else
   $e.rightOperand.notFlag \leftarrow true$

---

negations that can be removed from the *sub-expressions*. Figure 5.9 below illustrates this. The tree on the left shows the root with a score of -2, and the given sequence of trees shows that 2 negations, in fact, can be removed from the sub-trees by applying deMorgan's law to the root. The application also adds one new negation at the root of e, so the total change in the number of negations for the whole expression tree is $t_e + 1$. If the expression tree is rooted with a negation, the root negation can be removed by interchanging the `if` and `else` clauses of the conditional control flow statement.

Now, the minimization algorithm is done in the *same* pass as the calculation of the truth scores. If we find that $t_e < 0$ we apply deMorgan's law to e and reset t_e to

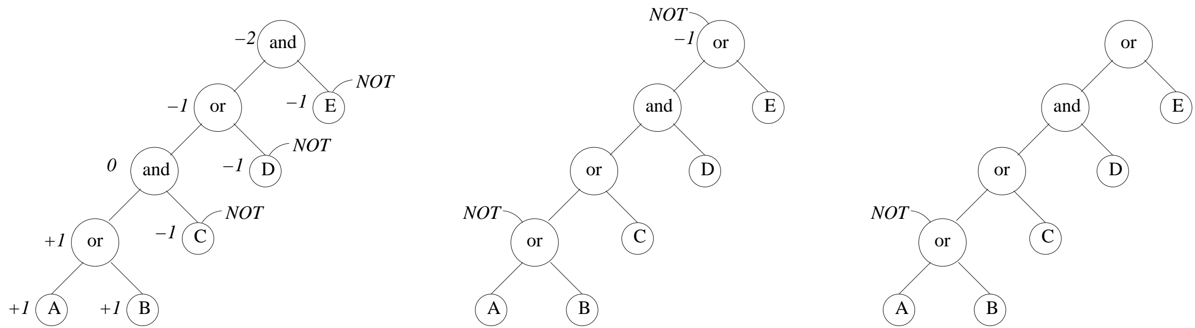


Figure 5.9: Removal of 2 negations from an expression tree.

-1. Then, if the truth score for some of the children of e is less than 1, we recursively apply deMorgan's law to those children. Algorithm 20 **minimizeNegations** page (99) shows how this is done.

Algorithm 20 minimizeNegations(ConditionalStmt s)

1. **pushNegationDown**(*s.conditionalExpr*)
2. **trickleNegationUp**(*s.conditionalExpr*)
3. If *s.conditionalExpr.notFlag* is *true*
   swap(*s.trueConsequent*, *s.falseConsequent*)
   $s.conditionalExpr.notFlag \leftarrow false$

We now prove by weak induction that this does minimize negation. The base case is simple propositions: regardless of whether a proposition is negated or not, there is no way to represent that proposition with fewer negations. The induction step concerns two minimally negated trees being joined by a non-negated root. The goal is to show that with a simple manipulation, the newly joined tree will have minimal negation. First consider that in each sub-tree there may be several points at which repeated application of deMorgan's law *may* produce a new sub-tree with the same number of negations, but we know by assumption, never fewer. Now, if a sub-tree indicates that it can do this at its root, then we know that we can effectively hoist one of the negations up out of the sub-tree and to its root. If both sub-trees, then, can be rooted by negations, whether or not by hoisting, we know that we can decrement

Algorithm 21 trickleNegationUp(Expression e)

1. If e is Negated
   $t_e \leftarrow -1$
   Return

2. If e is a Variable or Constant
   $t_e \leftarrow 1$
   Return

3. **trickleNegationUp**($e.rightOperand$) // Process right sub-expression
   trickleNegationUp($e.leftOperand$) // Process left sub-expression

4. $t_e \leftarrow t_{e.rightOperand} + t_{e.leftOperand}$
   If $t_e > 1$
   $t_e \leftarrow 1$

5. If ($t_e < 0$)
   applyDeMorganToSubTree(e)
   $t_e \leftarrow -1$

Algorithm 22 applyDeMorganToSubTree(Expression e)

1. If ($t_{e.rightOperand} < 1$) and ($e.rightOperand.notFlag$ is *false*)
   applyDeMorganToSubTree($e.rightOperand$)

2. If ($t_{e.leftOperand} < 1$) and ($e.leftOperand.notFlag$ is *false*)
   applyDeMorganToSubTree($e.leftOperand$)

3. **deMorgan**(e)

the total number of negations by one in the new tree, by applying deMorgan's law to the joining root.

Next, we see that there is no further way to decrement this total number of negations in the joined tree. Consider if there were, this would imply that we perform some application of deMorgan's law either at the root or in at least one of the sub-trees. It cannot be at the root since this would simply get us back to our initial state, where by assumption we could not improve either of the sub-trees. It cannot be in a sub-tree. To see this, we partition each sub-tree into two sections: that which

was undisturbed by our previous hoisting operation and that which was changed. By assumption, no application of deMorgan's law in the section that was undisturbed can reduce the number of negations. If we look at the changed section, we now see that each changed section contains a new minimal number of negations (the previous minimum minus one since by construction only one hoist was possible) and any application of deMorgan's law could only increase the number of negations. We conclude that there is no further way to decrement the number of negations.

Let's re-examine the implemented algorithm to make sure that it conforms to our proof. The first phase of the algorithm (**pushNegationDown**) sets up the condition that the join is never negative, and established the proof's base case. Then the second phase (**trickleNegationUp**) implements the induction. A truth score of less than 0 indicates that a sub-tree can be reformed, via a hoist, to be rooted at a negation and this is always performed. Finally, as we have seen in the description of the hoisting algorithm above (**applyDeMorganToSubTree**), deMorgan's law is applied to each sub-tree in such a way that it does not increase the number of negations. Our algorithm does indeed conform to the proof.

In summary, we reduce certain control flow graphs to conditional control flow statements with aggregated expressions, and simplify these expressions to contain a minimal number of negations.

5.2.2 Class names

There are two potential issues with class names. First, there may be a name clash between a class and a package. Although there is no ambiguity at the bytecode level, such name clashes will produce compile time errors with `javac`. This is presumably to encourage clarity in Java source code. Second, we want to add `import` statements to a method. In Grimp all class references are made with their full package prefixes. Although explicit, if we use the full name in our decompiled code it will be unnecessarily verbose. There is one phase per each of the problems. The first phase deals with package/class name clashes, and the second with the addition of `import` statements.

Without whole program knowledge, changing the names of classes or packages is not a safe procedure. We could change a name and accidentally break some outside reference. However, since several non-Java sourced members of our test suite require such changes to be recompilable we are compelled to put this feature in.

The first phase begins by trying to enforce the Java package/class capitalization convention; the first letter of a package name should be lower cased, while the first letter of a class should be capitalized. In enforcing this convention we may cause fresh name clashes. If this happens, we abandon the convention and generate a new random suffix for the package name such that the new package name will be both unique and not clash with any classes. The choice to change the package and not the class name was made because it creates less visual impact. When we change a package name we must change only the `import` headers in classes external to the current package, while a change in class name will be displayed every time that class is referenced.

In the second phase the methods of a class are scanned twice for references to other classes. Since every class reference will contain a package name prefix, the first scan just establishes a table of all the package names and creates a list of `import` statements. These `import` statements are then used to augment `javac`'s `CLASSPATH` environment variable. The second scan then checks to see if the package name prefix can be dropped from the class reference. Most of the time we will be able to; the only occasion when the prefix will *not* be dropped is when there are two different classes in the augmented `CLASSPATH` with the same base class name. Although `javac` is unambiguous about what will happen and will always choose the first class to found in the `CLASSPATH`, the situation is not visually clear. For this reason, all clashing class names are allowed to keep their full package prefixes, while all others are shortened to just their class names.

5.3 Readability Transforms performed on the SET

We now perform several transforms on the SET to improve the appearance and recompilablity of the output. To do this we search the SET in a bottom-up manner for structure idioms, and when found, perform the appropriate transforms to the SET. The SET traversed once for each idiom to avoid complication since some idioms may interfere with each other.

5.3.1 `synchronized()` blocks

Java is a multi-threaded programming language, providing inter-thread critical sections with `synchronized()` blocks. At the virtual machine level, Hoare's monitor

mechanism is implemented with `entermonitor` and `exitmonitor` instructions. These atomic instructions signal the virtual machine to obtain or release a lock on an object thus beginning or ending a critical section.

The JVMspec [13] requires that the use of these monitor instructions are 1) balanced: that during the invocation of a method, a thread should not try to release a lock on an object if it doesn't already own the lock, and 2) complete: that any lock obtained during the invocation should be released by the time the method exits.

The requirements only talk about *runtime* and not static constraints, so monitor instructions can in fact be placed in the bytecode with a low degree of structure. For example, the following would only suggest that the loop must be executed twice.

```
monitorenter o;
monitorenter o;
while (x) {
    monitorexit o;
    x = foo(a);
}
```

Figure 5.10: Unstructured use of monitor instructions.

If a thread exits a method by having an exception thrown, it must still release any object locks it holds. This means that every critical section should be covered by an area protection. If any exception is thrown, then the first handler for the area of protection will release the lock and rethrow the exception. Figure 5.11 (page 103) is a simple Java sourced code fragment that uses a `synchronized()` block.

```
System.out.println("start");
synchronized(o) {
    System.out.println("in synchronized block");
}
System.out.println("finish");
```

Figure 5.11: A simple `synchronized()` block in Java.

The unstructured Grimp output of this code is shown in figure 5.12 below.

The result of our restructuring algorithm so far will produce the output in figure 5.13 (page 104).

```
   java.lang.System.out.println("start");
   r3 = r1;
   entermonitor r3;

label0:
   java.lang.System.out.println("in synchronized block");
   exitmonitor r3;
   goto label2;

label1:
   $r6 := @caughtexception;
   r4 = $r6;
   exitmonitor r3;
   throw r4;

label2:
   java.lang.System.out.println("finish");
   ------------------------------------------------------------
   catch java.lang.Throwable from label0 to label1 with label1;
```

Figure 5.12: The Grimp equivalent of figure 5.11.

```
System.out.println("start");
entermonitor r3;
try {
    System.out.println("in synchronized block");
    exitmonitor r3;
}
catch (Throwable r6) {
    r4 = r6;
    exitmonitor r3;
    throw r4;
}
System.out.println("finish");
```

Figure 5.13: The structured code output before `synchronized()` blocks are found.

We now reduce the SET representation of figure 5.13 to a synchronized block. To do this we search the SET for the following pattern. Please note that the pattern should be read while referencing figure 5.13.

- There exists a statement sequence S which contains the sub-sequence: `entermonitor`

A followed by `try` statement B.

- A obtains a lock on some object C.
- The `try` statement B contains a statement sequence D whose last member is an `exitmonitor` E.
- There must be no abrupt exit from D.
- E releases a lock on C.
- There is only one `catch` clause F for the `try` block and furthermore this catches an exception of type `java.lang.Throwable` and assigns it to reference G
- F contains a statement sequence of the following:
  1. G *may* be assigned to a local H. This statement is optional.
  2. There is an `exitmonitor` which releases a lock on C.
  3. There is a `throw` instruction with reference to either G or H.

Once this pattern has been found, we remove the `entermonitor` A and the `try` B from the statement sequence S and replace them with a new `synchronized()` block T. We then remove `exitmonitor` E from statement sequence D and place D in T.

Resolving unstructured uses of monitor instructions

As shown in figure 5.10 above (page 103), monitor instructions do not have to be placed in the bytecode in a structured manner. However, because every critical section that can be converted to a `synchronized()` block is covered by an area of protection, the exception preprocessing and `try` nesting phases of our algorithm *ensure* that there are no nesting or structural problems between any potential `synchronized()` block and any other structured construct. Figure 5.14 below illustrates a contrived example where a `synchronized()` block with two entry points is "fixed" by exception preprocessing.

The consequence of this is that all attempted reductions to `synchronized()` blocks are guaranteed to be semantically correct and will fit within the already structured SET. However, we may not always find the appropriate reduction pattern with the occurrence of monitor instructions. In these cases we are stuck with statically

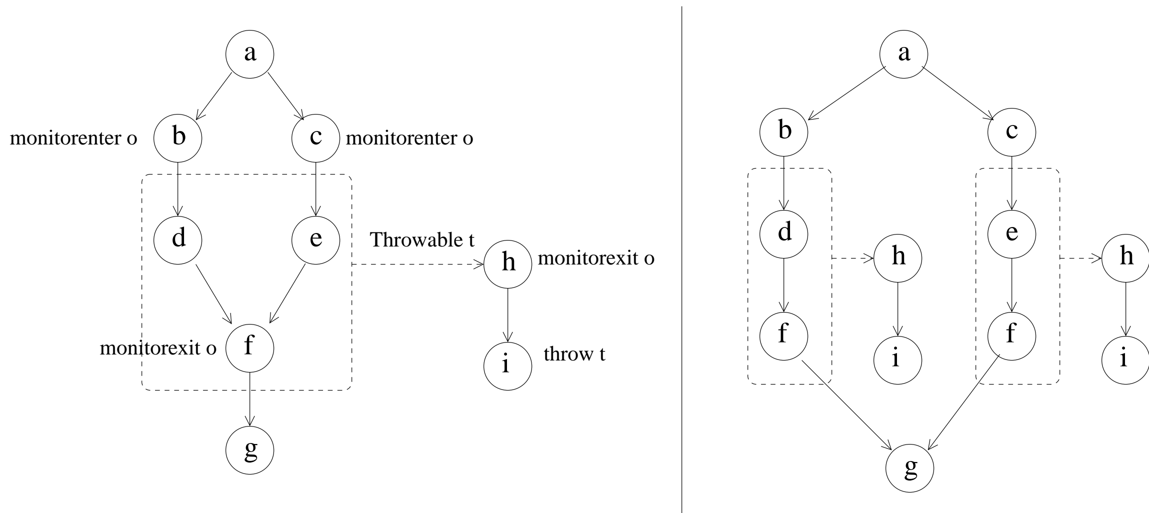


Figure 5.14: Exception preprocessing can resolve structuring problems involved with creating `synchronized()` blocks.

unstructured object locking. Since there are no monitor instructions in pure Java, we built a trivial library that implements object locking. The library provides a single class called `DavaMonitor` with static methods called `monitorenter` and `monitorexit`.

Once we have finished performing the reductions for `synchronized()` blocks, we iterate through the SET and replace all remaining instances of monitor instructions with static method calls to the `DavaMonitor` class.

Miscellanea

Given that we have an effective fallback mechanism, it might seem that we could have skipped `synchronized()` block reduction altogether. In example 5.13 we could have simply converted the monitor instructions to static method call to the monitor library.

Unfortunately, the Java language specification requires that we declare, in the method's `throws` clause, all exceptions that are 1) not derived from `java.lang.RuntimeException` and 2) are not caught in the current method. This means that to be able to recompile the method, we must declare that the method throws a `Throwable` exception because of the `throw` statement in the `catch` clause.

The better solution is to try to remove the `throw` statement by performing the reduction to a `synchronized()` block. Then we analyze the contents of the `synchronized()` block to find which exceptions are possibly thrown and declare them appropriately. The advantages are that we minimize changes to methods due to the type widening of declared thrown exceptions and retain a greater accuracy in the information presented in the final output.

5.3.2 `finally` blocks

Since many operations may need to be done on an area of protection before it can be represented in a structured manner, `finally` clauses, which are exceptions with extra requirements, are only found after all other structuring of exceptions have been completed.

In bytecode `finally` clauses employ a `jsr` instruction, which most decompilers probably use as detection fingerprint. In Grimp the body of code reached by `jsr` instruction is directly inlined for flow sensitive analyses, and so the `jsr` fingerprint is removed.

Dava uses the following pattern to find `finally` blocks:

To perform the above reduction we scan the SET for the following pattern.

1. There exists a `try` statement A that has a `catch` clause B that catches the `Throwable` exception type.
2. B contains a statement sequence C that terminates with a `throw` statement D that rethrows the caught exception.
3. There must be no abrupt exit from C.
4. We produce a new statement sequence E by copying C up to D.
5. There exists a statement sequence F following `try` statement A that is homomorphic to E.
6. For every other `catch` clause G, G must contain only one `try` statement H.
7. There must be no abrupt exit from H that does not immediately go to F.

```
try {
    ...
}
catch (Exception e) {
    try {
        a;
    }
    catch (Throwable t) {
        b;
        throw t;
    }
}
catch (Throwable t) {
    b;
    throw t;
}
b;
```

```
try {
    ...
catch (Exception e) {
    a;
}
finally {
    b;
}
```

Figure 5.15: Pattern that is searched for to build `finally` blocks.

8. H must have only one `catch` clause J which catches the `Throwable` exception type.

9. The statement sequence K inside J must be homomorphic to D.

If this pattern is found we do the following: 1) We remove `try` statement H from `catch` G and replace it with statement sequence from inside H, 2) we remove `catch` B, 3) we remove statement sequence F, and "finally" 4) create a new `finally` clause for A that contains the statement sequence E.

5.3.3 `for` loops

Dava treats `for` loops as a syntactic sugaring of `while` loops. Because `for` is such a commonly used construct, however, it is worthwhile to recover `for` loops for readability.

The pattern for this reduction is much simpler than the previous two.

```
i=0;                                for (i=0; i<10; i++) {
while (i<10) {                          System.out.println(i);
    System.out.println(i);          }
    i++;
}
```

Figure 5.16: `for` loop pattern.

1. In statement sequence S there exists a sub-sequence of an assignment statement A followed by a `while` statement B.
2. B contains a statement sequence C that ends in an assignment statement D.
3. C must not contain any `continue` statements that target B.

The assignment statement A and the `while` statement B are removed from S and replaced with a new `for` statement E. E is given the assignment from A as initialization, the conditional expression from B as the stopping condition, and the assignment statement D as increment.

Note that had there been any `continue` statements in the original Java source code, after this reduction they would now look like the left side of figure 5.17.

```
for (i=0; i<10; i++) {              for (i=0; i<10; i++) {
   Label_0: {                           ...
       ...                              continue;
       break Label_0;                   ...
       ...                          }
   }
}
```

Figure 5.17: Pattern that "fixes" `breaks` in `for` loops.

We remedy this by a second pass over the SET, performing another reduction. In this case the pattern we are looking for is a `for` loop A which contains only a single labeled block B. When found, we simply remove B from A and replace it with B's contents. We then scan B former body for `break` statements that targeted B and replace them with `continue` statements that target A.

5.3.4 `if-else` chains

Sometimes nested `if-else` statements can be converted into `if-else` chains. Figure 5.18 (page 110) illustrates the two transforms that are performed to do the conversion. On the left is the original restructuring, the middle shows the result of the first transform and the final output is on the right.

```
if (x) {
    if (y) {
        A;
    }
    else {
        if (z) {
            B;
        }
        else {
            C;
        }
    }
}
else {
    D;
}
```

```
if (!x) {
    D;
}
else {
    if (y) {
        A;
    }
    else {
        if (z) {
            B;
        }
        else {
            C;
        }
    }
}
```

```
if (!x) {
    D;
}
else if (y) {
    A;
}
else if (z) {
    B;
}
else {
    C;
}
```

Figure 5.18: Conversion of nested `if-else` to `if-else` chain.

The first transform makes sure that nested `if`s or `if-else`s are held only in `else` clauses. The reason for this is to allow for the nested `if` to be joined on to the current `else`. The pattern searched for here is an `if-else` A where the `if` clause of A contains a single `if` or `if-else` statement, and the `else` clause of A does not. When found, the conditional expression for A is negated and the `if` and `else` clauses of A trade roles.

The second transform is, in fact, only alternative way of printing, with the caveat is that there can be no labeled `break` in any of the `if-else` bodies that will be rendered meaningless by the new representation. The pattern here is an `if-else` A where the `else` clause of A contains a single `if` or `if-else`. If the pattern is matched, A is marked for the alternative printing.

5.3.5 Conditional assignments

Conditional assignments are found from a very strict pattern shown in figure 5.19 (page 111): there is an `if-else` A in which both the `if` and `else` clauses contains just a single assignment to the *same* local. If this pattern is found, A is replaced with a conditional assignment using the conditional expression from A and the right hand sides of the `if` and `else` clauses' assignments.

```
if (x) {
    a = b;
}
else {
    a = c;
}
```

```
a = (x) ? b : c;
```

Figure 5.19: A conditional assignment.

5.3.6 `if-else` / `continue` substitutions

The last idiomatic transform we present is a matter of personal taste more than the others. It is included because it reduces the nesting level for part of the code in a loop.

```
while (x) {
    A;
    if (y) {
        B;
    }
    else {
        C;
    }
}
```

```
while (x) {
    A;
    if (y) {
        B;
        continue;
    }
    C;
}
```

Figure 5.20: Using a `continue` to reduce level of nesting.

The pattern here is a loop A whose body is a statement sequence B that ends in an `if-else` C. As a first step the `if` and `else` clauses are checked to see how many

lines of code each will produce. If the `if` clause produces more lines than the `else` clause, the conditional expression of C is negated and the `if` and `else` clauses reverse their roles. Basically, we want to ensure that the `if` is smaller than the `else`. The second step is scan the bodies of both the `if` and `else` clauses and to convert any `break` statements that target C into `continue`s that target A. The final step, then, is to remove the `else` clause from A, to append the `else`'s body to the statement sequence B, and to append a `continue` to the body of the `if` clause.

Chapter 6

Testing and Results

6.1 Introduction

There are three ways we test and obtain results. First, we test each component of the decompiler, second, we test the decompiler on a benchmark suite, and third, we compare the output of Dava to a number of other leading Java decompilers.

Measuring how well Dava works is hard because there is very little published material on what the "standard" problems of decompiling Java should be. Although we compare our output with that of other Java decompilers, for the comparison to be meaningful, we first have to define what the key problems are. Dava's development life cycle, shown in figure 6.1 (page 114), relies heavily on testing to reveal not only programming bugs, but conceptual errors and previously unknown types of problems. For this reason testing, implementation and results are highly related.

To make sure that our results are not overly skewed by this strong relation, we perform two types of tests, component testing and benchmark testing. These verify that our solutions are correct and that new problems have simply not been overlooked.

Component testing is performed to exercise some feature of the decompiler, for example the reconstruction of `synchronized()` blocks. The goal is to not only to make sure that the component performs its function properly, but that it does not break any other part of Dava. This is done on a version of Dava called *dava-current*. Once we have finished testing and fixing dava-current and feel that it is strong enough for public use, we move its new components to *dava-stable*.

The stable version is then tested on a number of *benchmark* programs. These

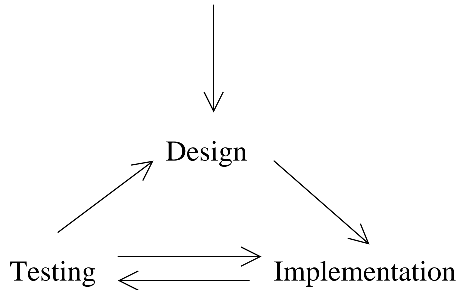


Figure 6.1: Development life cycle for Dava.

primarily come from the Ashes Hard Suite [22], and have been used to reveal new problems to be solved in dava-current.

As of the writing of this thesis, dava-stable performs at a comparable or better level than most other Java decompilers on both the component and benchmark tests. However, dava-stable does not yet incorporate all the features found in dava-current. Table 6.2 (page 115) gives an overview of what features have been implemented in what version, and the results of our testing. In this table, "Passed" means that the component is able to successfully decompile all tests it was given, while "Failed" means that for some test an incorrect output was given. There is no such thing as a partial pass.

6.1.1 Measures

There are two types of measures that we use, subjective and objective. An example of a subjective measure is how readable the decompiled output looks; poor, average, or good. An example of an objective measure is whether or not the output is recompilable. From a results point of view, we are interested in *what* the output of the decompiler looks like, and we *not* interested in *how* it was gotten. That is, knowing the hardware platform, Java virtual machine, operating system, or implementation of the Java API that are used, does not affect our evaluation of Dava.

Although the restructuring parts of the algorithm were designed with the intention of yielding easy to read and "natural", human-like output, these are subjective measures. In the following sections we give many examples of the output and make comments about what we feel are the strengths and weaknesses, but not make any

Description	Implemented in dava-current	Implemented in dava-stable	Component Tests	Benchmark Tests
SET construction	Yes	Yes	Passed	Passed
Basic loops	Yes	Yes	Passed	Passed
Nested loops	Yes	Yes	Passed	Passed
Multi-entry point loops	Yes	Yes	Passed	Passed
`if` and `if-else` blocks	Yes	Yes	Passed	Passed
`switch` blocks	Yes	Yes	Passed	Passed
Labeled blocks	Yes	Yes	Passed	Passed
Labeled `break`	Yes	Yes	Passed	Passed
Labeled `continue`	Yes	Yes	Passed	Passed
Class/package name fixing	Yes	Yes	Passed	Passed
Simple statement fixing	Yes	Yes	Passed	Passed
Constructor call fixes	Yes	Yes	Passed	Passed
`static` blocks	Yes	Yes	Passed	Passed
`throws` declarations	Yes	Yes	Passed	Passed
Throw `null` fix	Yes	Yes	Passed	Passed
Class literal fix	Yes	Yes	Passed	Passed
Basic exception handling	Yes	Yes	Passed	Passed
`synchronized` blocks	Yes	Yes	Passed	Passed
Useless `try` removal	Yes	Yes	Passed	Passed
Exception preprocessing	Yes	No	Passed	-
Exception handler removal	Yes	No	Failed	-
Advanced exception handling	Yes	No	Failed	-
Structural sugaring*	No	No	-	-

*Includes aggregating `if` statements, `for` loops, `if-else` chains, conditional assignments and `if-else` / `continue` substitutions.

Figure 6.2: Overview of Implementation and Testing.

strong claims on their subjective qualities.

There are 3 basic objective measures.

1. Completion. Did decompilation complete? If decompiling an application rather than a single class, were all the necessary classes decompiled? This measure is more useful for benchmark tests and less so for component testing.

2. Correctness. For example, is the restructuring correct? Is the output obviously or obviously not Java? Are the types of variables and constants correct?

3. Recompilability. Can the decompiled output be recompiled by `javac` without error? Does the compiled output run? Does it appear to run correctly? If the recompiled output is decompiled, does it converge to the original decompiled version?

These three objective measures, together with a subjective commentary, are applied to each component and benchmark test. Following this we compare the output of Dava with the output of several other Java decompilers. Briefly, section 6.2 covers component testing with subsections for basic loops, multi-entry point loops, `if`, `if-else`, and `switch` statements, labeled blocks, `break` and `continue` statements, basic exception handling, advanced exception handling, `synchronized` statements, class/package name clash resolution, and `throw` declarations. Section 6.3 gives a description of our benchmark programs and their test results, and section 6.4 compares our performance to several other leading Java decompilers.

6.2 Component Testing

Each of the tests here exercises a component of the decompiler singly and in combination with the other parts of the decompiler. Since Dava was built one component at a time, the first components have very few combination tests, while later components have many. For the sake of brevity, the issues from chart 6.2 have been grouped together in the following subsections.

6.2.1 Basic Loops

The first issue developed and tested were simple single entry point loops. The test cases were first written in Java, compiled with `javac` and then disassembled with `javap -c` to make sure that they exercised the correct problems.

There are three types of loops, `while`, `do-while` and `while(true)`, and three series of tests, one for each type of loop. Each series uses the chosen type of loop and tests various configurations by adding statements to either the body of the loop, or

```
while (x<10) {
}
```

```
System.out.println();
while (x<10) {
}
```

```
while (x<10) {
    System.out.println();
}
```

```
while (x<10) {
}
System.out.println();
```

Figure 6.3: A basic set of tests for simple loops.

prefixing or suffixing the loop. For example, the first set of tests in the `while` loop series deals with empty statement sequences as shown in figure 6.3 (page 117).

The `while` series then continues by placing the various loops inside a `while` loop. Figure 6.4 (page 117) shows a `while` in `while` nesting.

```
while (x<10) {
    while (y<10) {
    }
}
```

```
while (x<10) {
    System.out.println();
    while (y<10) {
    }
}
```

```
while (x<10) {
    while (y<10) {
        System.out.println();
    }
}
```

```
while (x<10) {
    while (y<10) {
    }
    System.out.println();
}
```

Figure 6.4: A basic set of `while` in `while` tests.

All these code snippets will produce differing control flow graphs. The control flow graphs for the examples in figure 6.4 are given in figure 6.5 (page 118). This illustrates that even though the original code can look simple and seem to change little, there can be profound differences in the control flow graph that the restructurer has to work on. For this reason, we felt it would be important to try to be as thorough as possible in testing simple combinations of loops.

These tests were then extended to exercise many of the variants of different types of loops nesting each other. Unfortunately, the total number of tests needed to rigorously explore the number of possible loop combinations increases exponentially

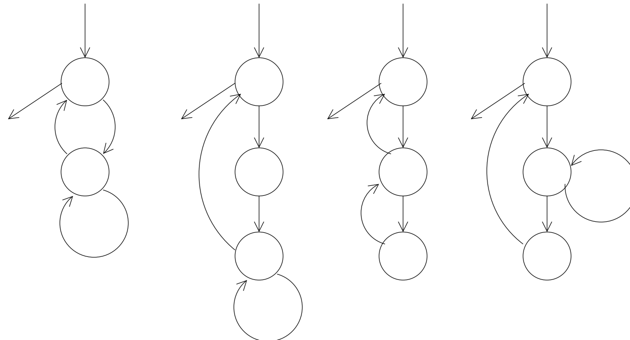


Figure 6.5: Control flow graphs for figure 6.4.

with the loop nesting level. We therefore decided that a hand coded set of tests consisting a complete set of simple tests and a random sample of complex cases would be sufficient.

Altogether, twenty two tests were performed on various configurations of simple loops. At this time simple loops have yielded perfect completion, correctness, and recompilability results. Although it is possible to contrive examples that will perform otherwise, on the tests we ran, Dava has always matched the original code in its selection of loop types.

6.2.2 Multi-entry point Loops

Multi-entry point loops are not directly restructured. Instead, Dava converts every multi-entry point loop into a single entry point version and passes the task on to the single entry point restructurer. The simplest multi-entry point loop is shown in figure 6.6.

The issue, then, is the correct insertion of new statements and redirection of control flow. Special attention has to be given to make sure that the new statements are put in the appropriate areas of protection. Since we are *not* concerned with how the resultant loops are restructured, but only with how the transform from multi-entry point to single entry point is made, we made up only four test cases. The test cases here were hand coded in Jimple and fed into Soot to be assembled into class files. These classes exercised four different combinations of exception handling in the entry points and all exhibited perfect completion, correctness, and recompilability. A

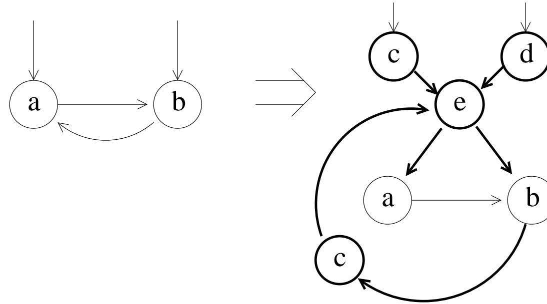


Figure 6.6: Control flow graph of a simple multi-entry point loop.

fifth "fun" example was tested as shown in figure 6.7 and was correctly decompiled.

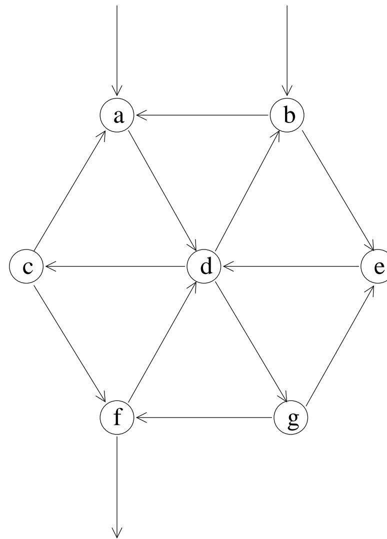


Figure 6.7: A "fun" multi-entry point loop.

While the output from this "fun" example is correct, it was still difficult to understand. This is reasonable given the highly complex structure of this example. In contrast, the four simple tests, are quite understandable. My feeling on the overall performance of the multi-entry point loop strategy in Dava is that it is excellent for the difficulty of the problem.

It should also be noted that Jimple was an ideal language to build these test cases

in, as it provides simple statements without the complexity of stack-based code and unstructured control flow.

6.2.3 `if`, `if-else`, and `switch` Statements

The `if`, `if-else`, and `switch` testing was performed in a similar manner to basic loops. The test cases were written in Java, compiled with `javac`, and disassembled to verify that they expressed the correct problems. As with basic loops, we created an extensive set of statement sequence and nesting test cases, and a random sample of more complex test cases. About twenty test cases were created altogether, and have all passed completion, correctness, and recompilability.

A set of tests were then built to verify the interaction between loops and `if` and `switch` statements. Again because the huge number of possible combinations precludes exhaustive testing, we used a random sample of eight tests to ensure that loops and `if` and `switch` statements work together properly. Each test had a minimal nesting of three control flow statements, with a variety empty and non-empty bodies. These tests have also all passed completion, correctness, and recompilability.

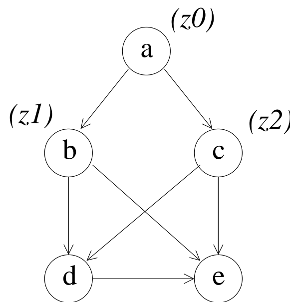
As with loops, all tests yielded decompiled source code that is equivalent to the tests' original source.

6.2.4 Labeled Blocks, `breaks` and `continue` Statements

Labeled Blocks, `break` and `continue` statements really start to show off the power of Dava. The suite of tests for this section is made up of two parts, a set of eight new test that exercise a random sample of simple basic labeled blocks and `break` and `continue` statements, and the modification of a large number of test programs from the previous sections.

With the inclusion of labeled blocks, `breaks` and `continue` statements we *should* be able to decompile any Java method that does not contain exceptional control flow. After building the eight simple tests, we disassembled and randomly modified the control flow targets from about twelve of the more complex tests from the previous test suites. These twelve were then reassembled and run to check that they were still verifiable Java bytecode.

Figure 6.8 (page 121) shows the control flow graph for a simple test and the resulting output code from Dava. All eight simple and all twelve advanced tests have



(a) Control Flow Graph

```
public void bar(boolean z0, boolean z1, boolean z2)
{
    label_0:
    {
        if (z0 == false)
        {
            if (z2 != false)
            {
                break label_0;
            }
        }
        else
        {
            if (z1 != false)
            {
                break label_0;
            }
        }

        System.out.println("d");
    }

    System.out.println("e");
    return;
}
```

(b) Dava Output

Figure 6.8: Example program control flow graph and Dava output.

yielded perfect completion, correctness, and recompilability results. The subjective results are excellent. When the simple tests were constructed, they were built with the intention of adding a controlled number of labeled blocks. In this case, Dava produces the exact output that was hoped for.

The advanced tests, generally produced very complex output with a high level of nested labeled blocks. Although these restructurings were hard to understand, they were always correct, complete and recompilable. Again, the complexity is to be expected because Dava was structuring tests with intentionally “broken” control flow structures. After studying the output, it is not obvious how a more advanced approach could improve on Dava’s results.

6.2.5 Basic Exception Handling

Basic exception handling refers exceptions that would be generated from a structured language. This rules out most of the bizarre cases that were shown in chapter 4. The test suite here consisted of seven simple and five complex tests and was built in Java and compiled with `javac`. As with other similar test suites, these tests were also disassembled to verify that they exhibited the correct problems.

```java
public int foo1( int i, int j)
{
    while (true)
    {   try
        {   while (i < j)
                i = j++/i;
        }
        catch (RuntimeException re)
        {   i = 10;
            continue;
        }
        break;
    }
    return j;
}
```

(a) Original Java method

```java
public int foo1(int i0, int i1)
{
    int $i2;

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

(b) Dava output

Figure 6.9: Decompiled code for method `foo1()`

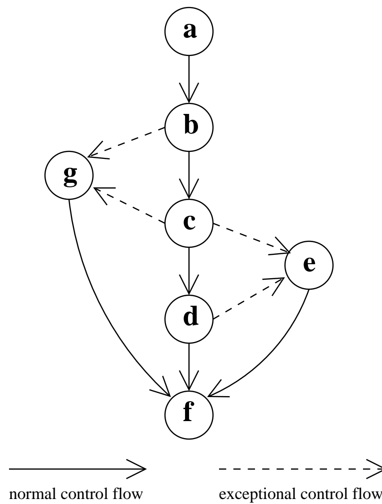
The simple tests covered the usual variety of empty and non-empty bodies. Note that to compile an empty `try` block, one must catch only `RuntimeException` and its sub-classes. The complex tests then tried to exercise exception dependent loops, the creation of labeled blocks and various nesting problems. An interesting example is shown in figure 6.9 (page 122) where Dava preserves the exception dependent loop, but also makes this loop do double duty for an originally nested inner loop. The new structure is equivalent to the original and it is only a matter of taste as to which expresses the desired control flow effect better.

All twelve basic exception handling test passed completion, correctness, and recompilability. In general the output was quite readable, although in the advanced tests, the decompiled versions frequently deviated from the original source code.

6.2.6 Advanced Exception Handling

Dava-stable is already able to handle many advanced exception problems, which we would like to highlight at this time. For example, figure 6.10 (a) (page 123) illustrates the control flow graph for a method that we hand coded in Jimple and assembled with Soot. This method had two areas of protection, one that covers

statements (b,c) using g as the exception handler, and one that covers statements (c,d) using e as the handler. These areas of protection intersect but do not nest and so cannot be directly translated into try blocks.




```
public void foo2()
{
    System.out.println("a");
    label_0:
    {   try
        {   System.out.println("b");
        }
        catch (RuntimeException $r9)
        {   System.out.println("g");
            break label_0;
        }
        try
        {   System.out.println("c");
        }
        catch (RuntimeException $r9)
        {   System.out.println("g");
            break label_0;
        }
        catch (Exception $r5)
        {   System.out.println("e");
            break label_0;
        }
        try
        {   System.out.println("d");
        }
        catch (Exception $r5)
        {   System.out.println("e");
            break label_0;
        }
    }
    System.out.println("f");
    return;
}
```

(a) Original control flow graph (b) Dava output

Figure 6.10: Control flow graph and decompiled code for method foo2()

In structuring this example, Dava successfully broke the areas of protection into three non-intersecting areas of protection and duplicated the appropriate handler statements. This example passes correctness and recompilability.

Dava-stable has also passed tests demonstrating that it is able to remove catch clauses for which it is statically provable that the caught exception type will never be thrown. These tests were written in Jimple and assembled with Soot and followed two patterns in their construction. First, we inserted spurious areas of protection in the method's exception table, and second we extended already existing areas of protection. The intuition with extended areas of protection is that will likely cause nesting problems and have to be split into two or more parts. Each part, then,

may or may not contain the potentially thrown exception site and `catch` blocks are removed accordingly. Finally, in all tests which produced `try` blocks without any `catch` clauses, Dava successfully removed the useless `try` blocks.

In future work, dava-current will be able to properly handle all tests involving self-targeting area of protection. Several tests are currently failing on this issue due to bugs in implementation.

6.2.7 `synchronized` Statements

Both successful `synchronized` statement restructuring and the fall-back mechanism were tested. Four tests for successful `synchronized` restructuring were written in Java, again employing various combinations of empty and non-empty statement bodies. As usual, these were compiled with `javac` and disassembled to make sure they presented the correct patterns.

Six tests that employed the use of monitor instructions were then written in Jimple and assembled into class files with Soot. Two of these made unstructured use of the instructions such that they would create unverifiable bytecode, two make use of the instructions in a structured way that should *not* result in the creation of `synchronized` blocks but still be verifiable bytecode, and two were structured such that they should be rebuilt into `synchronized` blocks.

In all ten cases Dava created correct, recompilable source files. In the case of the two unverifiable methods, the conversion of the monitor instructions to calls into Dava's monitor library rendered the classes to be verifiable.

6.2.8 Class/Package Names Clashes

This problem originally showed up with an application compiled to Java bytecode by the SmallEiffel compiler. Here the problem was that the application's name was used both for the name of a bootstrapping class and package directory in the same location. Dava's fix ensures that name clashes don't happen in general.

To test this problem we used the original Eiffel application, and a small application of my own in which we tried to maximize the possible number of name clashes over a set of four packages. Currently, the class/package name capitalization convention is followed unless that causes new name clashes. The fallback is to append a tail to the

package. For example, package `spread_illness` is already lower cased. Had there been both classes named `spread_illness` and `Spread_illness`, then the new name for the package would be `spread_illness_p0`. Name clash resolution has passed completion, correctness, and recompilability.

6.2.9 `throws` Declarations

This problem originally was found in an application compiled from the ML language. A large number of exception types were generated by the ML compiler, used in many places but never declared in the methods' `throws` attributes. The solution here was rebuild the `throws` information from scratch. Subsequently, we found other benchmark programs that also contained insufficient `throws` method attributes.

Given that the ML compiler was generating at least twelve exception types per application, and that our benchmark suite also contained random declaration errors, we felt that relying on set of benchmark test programs was sufficient. The `throws` declarations tests from the benchmark programs have all passed completion, correctness, and recompilability.

6.3 Benchmarks Description and Testing

The core benchmark suite for Dava is the Ashes [22] "hard" test suite, which is a series small and mid-sized applications from various source languages. These are useful because they free of restrictions on reverse engineering and present a variety of problems from Java sourced code and from compilers other than `javac`. Figure 6.11 (page 126) gives a quick overview of the contents of the benchmark suite. The following paragraphs go into detail about the results of each member of the suite.

boyer: This a toy theorem prover written in ML and compiled with an ML to bytecode compiler. Because the source is a functional language the division of code of between classes is very uneven. The driver class `Main` is 254 lines which are mostly complex chains of method calls, the main application class `G` is 783 lines and contains most of the application logic, while the remainder of the classes usually have only about 30 to 40 lines each. Both regular and exceptional control flow are generally simple, but `throws` declarations are non-existent. A visual scan of the decompiled code reveals that except for the method call chains in the driver, it is easy to read and understand.

Name	Description	Source	Classes	Lines	Result
boyer	Theorem Prover	ML	34	1764	Passed
lexgen	Generates lexer for SML	ML	67	6857	Passed
illness	Simulation of Illness Spread	Eiffel	11	1797	Passed
lu	Matrix Factorization	Java	1	229	Passed
fft	Fast Fourier Transform	Java	1	274	Passed
matrix	Matrix inverter	Java	2	353	Passed
puzzle	Image Recognition	Java	3	1397	Passed
decode	Decryption	Java	5	732	Passed
machineSim	Micro-architecture Simulator	Java	12	2313	Passed
javazoom	Mp3 to WAV Converter	Java	37	12534	Passed

Figure 6.11: Description of Core Suite.

lexgen: Lexgen is a full strength lexer generator, also written in ML and compiled with an ML to bytecode compiler. As with boyer, the code division among classes is uneven. In this case the driver class `Main` has only 160 lines, while the main application class `G` has 5576 lines. The characteristics of the decompiled code are much like boyer and are usually easy to read.

illness: This is a small simulation of the spread of a virus through an 18 by 18 grid-based environment. It is written in Eiffel, and was compiled with an Eiffel to bytecode compiler. It is made up of a driver class, a main application class and 9 supporting utility classes. This is one of the few benchmarks that needs to use packages. Several interesting characteristics of the illness application are that there are package/class name clashes, class/keyword name clashes, and several instances of problems with `throws` declarations. As well, the decompiled version of illness reveals some inefficiencies in the SmallEiffel compiler's generated code, specifically that unnecessary boolean evaluations are sometimes inserted.

lu: LU does simple matrix factorization and is written in Java and compiled with `javac`. This is an interesting example to decompile because the matrix operations contain many nested loops. While the original was only 162 lines and the decompiled version is 229 lines, the decompiled version is still quite easy to understand.

fft: FFT is an implementation in Java of a Fast Fourier Transform of some complex double precision data. This application contains many nested loops, and also presents special problems with the representation of the double precision data.

matrix: This application is a simple matrix inverter written in Java. Its most interesting feature is that it contains a very high number of nested `if` statements. While Dava does not yet aggregate `if` statements, matrix would be an excellent test for this when it is added.

puzzle: This is an image recognition program written in Java, which attempts to fit four dimensional puzzle pieces together. It has very deep control flow structure nesting. While the original source code has nestings up to six levels deep, the decompiled output does go up to seven because of the lack of `if` aggregation. In general, though, the decompiled output closely mimics the original source code.

decode: This is an implementation in Java of an algorithm for decoding encrypted messages using Shamir's Secret Sharing scheme. This application contains complex use of double precision data and nested loops. Decompiled code readability is excellent.

machineSim: This is a machine simulator program. Implemented in Java, it simulates a micro-architecture executing an instruction stream. This application is interesting because it is the first example of a Java sourced program makes use of both exception handling and explicit throwing. As well, there is quite a bit of complex regular control flow in the original source. Decompiled output was both correct and reasonably easy to read.

javazoom: Javazoom is a Java sourced mp3 to WAV audio file format converter. At over 12,000 lines this is the largest application in the core suite. It exercises many features including packages, explicitly thrown and caught exceptions and `throws` declarations. Output is generally easy to read.

The entire core set of benchmarks now passes completion, correctness, and recompilablity on dava-stable.

6.4 Comparison to other Decompilers

We will now look at the selected output of four other Java decompilers. The comparisons are meant to show where Dava is better and worse than other decompilers. The four decompilers we look at are Jasmine version 1.10 [17, 24], Jad version 1.5.8 [10], Wingdis version 2.16 [30], and the SourceAgain version 1.1 online demo [23]. `Jasmine` (also known as the SourceTec Java Decompiler) is an improved version of `Mocha`, probably the first publicly available decompiler. `Jad` is a decompiler

that is free for non-commercial use whose decompilation module has been integrated into several graphical user interfaces including `FrontEnd Plus`, `Decafe Pro`, `DJ Java Decompiler` and `Cavaj`. `Wingdis` is a commercial product sold by WingSoft. Finally, `SourceAgain` is a commercial Java decompiler that has a web-based demo version.

The first test we will look at is a simple pair of nested loops with a compound conditional on the inner loop. This test was written in Java and compiled with `javac`. The original code and decompiled output from Dava plus the four other decompiler are shown in figure 6.12 (page 129).

All the decompilers give correct output. Jasmine (b) correctly finds the two `for` loops and pulls the declaration of one of the variable declarations into the appropriate loop. Apart from neglecting to put a double slash (`//`) in front of the comment, and not pulling the second variable declaration into the inner loop, Jasmine's output is perfect. Jad's output (c) is perfect, except for the insertion of a spurious pair of brace brackets ({}). SourceAgain's output (d) is excellent, although it inserts extra initializations at each variable declaration. Although correct, Dava's output (e) is not as appealing. Three problems immediately appear: (1) Dava does not yet build control flow statements with aggregated expressions and has to resort to using a `break` statement (2) `for` statements are not yet built, and (3) all variable declarations are done only at the top of the method. Wingdis (f) does a little better and finds the `for` statements, but seems to suffer from the same sort of deficiencies.

It is obvious that decompiler writers have put much effort into finding and rebuilding idiomatic segments in the bytecode. Even though Dava is behind in this area, chapter 5 details many algorithms that should bring Dava up to par on this type of example. We now turn our attention to more difficult tests.

The first example is the exception dependent loop that was originally shown in figure 6.9 (page 122). As we saw at the time, Dava did not mimic the original code, but did give a correct, readable decompilation. Figure 6.13 (page 130) shows the output of the other four decompilers.

Here, the key issues are that we have an unconditional loop that only iterates upon the throwing of an exception. Unfortunately, none of the others correctly decompiled this. Jasmine tried to create a complex `for` loop, but failed to find the loop body. It also issued dead code and a meaningless "pop" statement. SourceAgain dropped outer loop and the exception entirely. Jad did the best of the lot, only missing a target for its `break` statement, and still producing an uncompilable comment. Lastly, Wingdis seems to have given up on the control flow structure entirely and has emitted

```
public int foo3( int x) {
    for (int i=0; i<10; i++)
        for (int j=1; (j<i) &&
                      (j%3 != 0); j+=i)
            x += goo( x);

    return x;
}
```

(a) Original Java

```
public int foo3(int i)
{
Declarators could not be incorporated in
for statement
    int k;
    for (int j = 0; j < 10; j++)
        for (k = 1; k < j &&
                    k % 3 != 0; k += j)
            i += goo(i);
    return i;
}
```

(b) Jasmine

```
public int foo3(int i)
{
    for(int j = 0; j < 10; j++)
    {
        for(int k = 1; k < j &&
                      k % 3 != 0; k += j)
            i += goo(i);

    }

    return i;
}
```

(c) Jad

```
public int foo3(int i)
{
    int j = 0;

    for( j = 0; j < 10; ++j )
    {
        int k = 0;

        for( k = 1; k < j &&
                    k % 3 != 0; k += j )
            i += goo( i );
    }
    return i;
}
```

(d) SourceAgain

```
public int foo3(int i0)
{   int i1, i2;

    i1 = 0;

    while (i1 < 10)
    {   i2 = 1;

        while (i2 < i1)
        {   if (i2 % 3 != 0)
            {   i0 = i0 + this.goo(i0);
                i2 = i2 + i1;
            }
            else
            {   break;
            }
        }

        i1 = i1 + 1;
    }

    return i0;
}
```

(e) Dava

```
public int foo3(int int1)
{
    int int3;
    int int2;
    for (int2= 0; (((byte)int2) < 10); int2++)
    {
        for (int3= 1; (int3 < int2);
                              int3= int3 + int2)
        {
            if ((int3 % 3) == 0)
            {
                break;
            }
            int1= int1 + goo(int1);
        }

    }

    return int1;
}
```

(f) Wingdis

Figure 6.12: Original and decompiled code for method `foo3()`

```
public int foo1(int i, int j)
{
    RuntimeException e;
    for (i = j++ / i; i < j; i = j++ / i)
        /* null body */ ;
    return j;
    pop e
    i = 10;
}
```

(a) Jasmine

```
public int foo1(int i, int j)
{
    while( i < j )
        i = j++ / i;
    return j;
}
```

(b) SourceAgain

```
public int foo1(int i, int j)
{
    while(true)
        try
        {   while(i < j)
                i = j++ / i;
            break MISSING_BLOCK_LABEL_25;
        }
        catch(RuntimeException runtimeexception)
        {   i = 10;
        }
    return j;
}
```

(c) Jad

```
    public int foo1(int int1, int int2)
    {

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

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

(d) Wingdis

Figure 6.13: Decompiled code for method `foo1()`

meaningless `goto` statements instead.

It should be noted that this example was written in Java, compiled with `javac` and no modifications were made to the bytecode before decompilation. If these decompilers are meant to target only `javac` sourced code, they still have a ways to go in their abilities to handle structured control flow. We now see how they perform on class files that were not generated by `javac`.

The next example is the advanced exception test that was shown in figure 6.10 (page 123). Again, we saw that even in its incomplete state, Dava is able to correctly

restructure and decompile this example. Figure 6.14 (page 131) shows how the other decompilers fared on this test.

```
public void foo2()
{
    System.out.println("a");
    System.out.println("b");
    try
    {   System.out.println("c");
        System.out.println("d");
    }
    // Misplaced declaration of an exception variable
    catch(D this)
    {   System.out.println("e");
    }
    System.out.println("g");
    return;
    this;
    System.out.println("f");
    return;
}
```

(a) Jad

```
public void foo2()
{
    System.out.println("a");
    System.out.println("b");
    System.out.println("c");
    System.out.println("d");
    pop this
    System.out.println("e");
    System.out.println("f");
    return;
    pop this
    System.out.println("g");
}
```

(b) Jasmine

```
public void foo2()
{
    System.out.println("a");
    try
    {   System.out.println("b");
        try
        {   System.out.println("c");
            System.out.println("d");
        }
        catch (Exception e0)
        {   System.out.println("e");
        }
    }
    catch (RuntimeException e0)
    {   System.out.println("g");
    }
}
```

(c) Wingdis

```
public void foo2()
{   System.out.println( "a" );
label_9:
    {   try
        {   System.out.println( "b" );
            try
            {   System.out.println( "c" );
                break label_9;
            }
            catch( Exception exception1 )
            {   System.out.println( "e" );
            }
        }
        catch( RuntimeException runtimeexception1 )
        {   System.out.println( "g" );
        }
        System.out.println( "f" );
        return;
    }
    System.out.println( "d" );
}
```

(d) SourceAgain

Figure 6.14: Decompiled code for method `foo2()`

Jad produces dead code, a meaningless `this` statement, and only one `try` block, even though there are two entries in the method's exception table. Jasmine, produces dead code, a `pop this` statement, and no areas of protection. Wingdis produces a recompilable source, but does not capture the semantics of the exception handling. For example statement `e` is under protection, such that if a `RuntimeException` is

thrown, control will be transfered to statement `g`, which is clearly not specified in the method. Likewise SourceAgain produces a recompilable source file, but also fails to capture the semantics of the original method.

From these comparisons, we can conclude that although other decompilers may be more advanced in their abilities to handle common and simple control flow in a method, they are not able to handle more advanced control flow, or potentially the output from optimizers and compilers other than `javac`.

6.5 Conclusions

In this chapter, we have seen three ways of evaluating Dava; component testing, benchmark testing, and comparison to other Java decompilers. In component testing we established that the implemented feature really solved the problems they were designed to work on, in benchmark testing we looked at the decompilation of a number of applications "in the wild" to gauge if our decompiling feature set in Dava is reasonable, and in comparison, we see what limits the competition has accepted in trying to decompile difficult examples.

Dava was split into two versions for testing, dava-current and dava-stable. Dava-current was used for testing new components. Once we were satisfied that the new component was functioning properly and did solve the problem it was designed for, the component was moved to dava-stable. Dava-stable was repeatedly tested on a core set of benchmark programs. If a benchmark failed in completion, correctness, or recompilability, a new component to address the appropriate problem was designed for dava-current. This was repeated until the core benchmark set passed completely. Other benchmark applications were also used with an emphasis on bytecode from sources other than `javac`.

Dava currently out-performs most other leading Java decompilers in its ability to handle the widest range of verifiable bytecode. Dava is currently weak in its presentation of idioms, but this is an implementation issue rather than an inherent disability. With further implementation, Dava should be able to overcome these problems and produce the highest quality output possible.

Chapter 7

Conclusions

This thesis has presented new algorithms for decompiling Java bytecode into Java source code. The approach can be tuned for various goals. If the goal is the direct representation of bytecode in source, certain advanced transforms can be turned off, at the expense of potentially not being able to decompile as wide a variety of bytecode as would be otherwise possible. If, however, an all out attempt at reverse engineering is the goal, the full set transforms can be turned on to represent a truly wide range of bytecode configurations in pure Java source.

Key to Dava's approach is a new data structure called the Structure Encapsulation Tree. This data structure allowed us to put together the control flow structures in any order. We therefore were able to build the basic control flow structures in an order according to the difficulties involved in recognizing the types of structures. The result is a robust suite of algorithms that find a full structuring for any control flow graph. We then looked at each of the structure finding sub-algorithms individually and proved why they work.

Once we have finished basic structuring, we looked at more advanced issues, including syntactic sugaring, and issues for creating recompilable Java applications.

Finally, we tested the decompiler. There were three types of tests, component tests, benchmark tests, and comparison to other decompilers. Dava has passed the high majority of its component and benchmark tests and has demonstrated with these that it is stable and capable of handling a wide range of input classes. Specifically, we tested applications sourced from Java, Ada, Haskell, Eiffel, and ML which ranged from a few hundred lines of decompiled code to several thousand. Lastly, we compared Dava's output to that of four other current Java decompilers and found that although

Dava lags temporarily in its abilities to recognize certain simple programmer idioms, it excels in its ability to correctly decompile a wider range of verifiable bytecode.

The approach presented in this thesis seems to be more robust than any other decompiler available today. With continued implementation of the idiom handling sub-algorithms and debugging of the advanced exception handling, Dava could become the premiere Java decompiler available.

Bibliography

[1] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–250, March 1992.

[2] Brenda S. Baker. An algorithm for structuring flowgraphs. *Journal of the Association for Computing Machinery*, pages 98–120, January 1977.

[3] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, July 1994.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill and MIT Press, 1990.

[5] Ana M. Erosa. A goto-elimination method and its implementation for the McCAT C compiler. Master's thesis, School of Computer Science, McGill University, May 1995.

[6] 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.

[7] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. Microsoft technical report, Microsoft Research, October 1998.

[8] Etienne M. Gagnon, Laurie J. Hendren, and Guillaume 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.

[9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1997.

[10] Jad - the fast Java Decompiler
. URL: http://www.geocities.com/SiliconValley/Bridge/8617/jad.html.

[11] Sun Microsystems Java Homepage. URL: http://java.sun.com.

[12] Patrick Lam. Of Graphs and Coffi Grounds: Decompiling Java. Technical report, School of Computer Science, McGill University, September 1998.

[13] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[14] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[15] Jerome Miecznikowski and Laurie Hendren. Decompiling Java using Staged Encapsulation. In *The Working Conference on Reverse Engineering*, pages 368–374, October 2001.

[16] Jerome Miecznikowski and Laurie Hendren. Decompiling Java Bytecodes: Problems, Traps and Pitfalls. In *CC 2002 - International Conference on Compiler Construction*, April 2002.

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

[18] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[19] Todd A. Proebsting and Scott 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.

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

[21] The Sable Research Group. URL: http://www.sable.mcgill.ca.

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

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

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

[25] Sun Microsystems. URL: http://www.sun.com.

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

[27] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In David A. Watt, editor, *Compiler Construction, 9th International Conference*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, Berlin, Germany, March 2000. Springer.

[28] Raja Vallée-Rai and Laurie J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Sable Technical Report 1998-4, Sable Research Group, McGill University, July 1998.

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

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