PROGRAMMER-FRIENDLY DECOMPILED JAVA

by

Nomair A. Naeem

School of Computer Science
McGill University, Montréal

August 2006

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

Abstract

Java decompilers convert Java class files to Java source. Common Java decompilers are *javac-specific* decompilers since they target bytecode produced from a particular `javac` compiler. We present work carried out on Dava, a *tool-independent* decompiler that decompiles bytecode produced from any compiler. A known deficiency of tool-independent decompilers is the generation of complicated decompiled Java source which does not resemble the original source as closely as output produced by javac-specific decompilers. This thesis tackles this short-coming, for Dava, by introducing a new back-end consisting of simplifying transformations.

The work presented can be broken into three major categories: transformations using tree traversals and pattern matching to simplify the control flow, the creation of a flow analysis framework for an Abstract Syntax Tree (AST) representation of Java source code and the implementation of flow analyses with their use in complicated transformations.

The pattern matching transformations rewrite the ASTs to semantically-equivalent ASTs that correspond to code that is easier for programmers to understand. The targeted Java constructs include `If` and `If-Else` aggregation, for-loop creation and the removal of abrupt control flow. Pattern matching using tree traversals has its limitations. Thus, we introduce a new structure-based data flow analysis framework that can be used to gather information required by more complex transformations. Popular compiler analyses *e.g.*, reaching definitions, constant propagation *etc.* were implemented using the framework. Information from these analyses is then leveraged to perform more advanced AST transformations.

We performed experiments comparing different decompiler outputs for different sources of bytecode. The results from these experiments indicate that the new Dava back-end considerably improves code comprehensibility and readability.

Résumé

Les dcompilateurs Java convertissent le code binaire compil Java en code source Java. Les dcompilateurs Java les plus communs sont *spcifiques* au compilateur javac parce qu'ils ciblent le code binaire produit par un compilateur `javac` particulier. Nous prsentons notre travail sur Dava, un dcompilateur *indpendant* qui dcompile du code binaire Java compil partir de n'importe quelle source. Une faille connue des dcompilateurs indpendants est la gnration de code source Java complexe qui ne ressemble pas autant au code source original que celui produit par les dcompilateurs spcifiques javac. Cette thse s'attaque cette faille, pour Dava, en introduisant un nouveau systme de transformations de simplification.

Le travail prsent peut tre divis en trois catgories majeures : les transformations utilisant la traverse d'arbres et la reconnaissance de squences pour la simplification du flot de contrle, la cration d'un systme d'analyse du flot de contrle pour une reprsentation en tant qu'Arbre de Syntaxe Abstrait (AST) du code source Java et l'implmentation d'analyses du flot pour usage dans les transformations complexes.

Les transformations utilisant la reconnaissance de squences rcrivent les AST pour produire de nouveaux AST smantique quivalente, correspondant du code qui sera plus facile comprendre pour les programmeurs. Les constructions Java cibles incluent les aggrgations `If` et `If-Else`, la crations de boucles for et l'limination de flot de contrle abrupte. La reconnaissance de squences utilisant la traverse d'arbres a ses limitations. Nous avons donc dcid d'introduire un nouveau systme d'analyse du flot de donnes bas sur la structure qui peut tre utilis pour obtenir de l'information requise par des transformations plus complexes. Des analyses de compilateurs communes (par example : l'obtention de dfinitions, la propagation des constantes, etc.) ont t implmentes en utilisant notre systme. L'information produite par ses analyses est utilise pour produire des transformations plus avances.

Des expriences qui comparent la sortie produite par diffrents compilateurs reprsentant plusieurs sources de code binaire ont ts ralises, dmontrant que le nouveau systme d’analyse et de transformations de Dava amliore considrablement la clart et la lisibilit du code source produit.

Acknowledgements

First and foremost I would like to thank my supervisor Professor Laurie Hendren for introducing me to the wonderfully exciting field of programming languages and compilers, for her guidance in my research work and for her high expectations from her students. Her cheerful nature and her humor always kept me going in those dark hours and her quick insight and knowledge made my stay at the Sable Research Group a true learning experience.

A special thanks to Professor Clark Verbrugge for taking the time out to teach me ”faux-621”, for spending countless hours discussing potential research topics and for being a mentor in Laurie’s absence. I would also like to thank the Professors from the School of Computer Science for the wonderful courses taught by them that kept me here for six years. Thanks also to the admin and system staff for their help on countless occasions.

Additional thanks to my friends and members of the Sable Group – in no particular order – Grzegorz Prokopski, Dayong Gu, Chris Goard, Chris Pickett, Sokham Pheng, Ondřej and Jennifer Lhoták, Jerome Miecznikowski and Navindra Umanee. A special thanks to Maxime Chevalier-Boisvert for helping me translate my abstract into French. Mike Batchhelder’s work on Java obfuscation and his repeated “successful” attempts to crash Dava were a true inspiration for numerous transformations and bug fixes which became part of this thesis.

Thank you to Ahmer Ahmedani for being my buddy at McGill, for our discussions on religion and world affairs and our coffee breaks. Also my pool partners Waqqas, Farhan, Moiz and Moeed for the much needed time-outs. Last, but not least, I thank my parents, sisters and my wife for their love, devotion and support.

Dedicated to

My Parents,
Dr. Pervaiz Naeem Tariq and Dr. Shahida Naeem

and

My Wife,
Mariam Rasool

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction and Motivation

Since its creation, the Java [GJS97] programming language has become increasingly popular. The highly object-oriented design, exception handling, runtime checking and garbage collection are some of the features making Java an attractive language for developers. The biggest reason for Java's popularity, however, is the portability of the binaries for Java. Java compilers, such as the standard `javac` compiler created by Sun Microsystems [Sun, Jav], produce Java class files and these are the binary form of the program which can be distributed or made available via the Internet for execution by Java Virtual Machines (JVMs) [LY99]. Although the `javac` compiler is the most usual way of producing class files, there are an increasing number of other tools that also produce Java class files. Figure 1.1 shows some other sources of bytecode. There exist compilers for other languages including AspectJ[KHH+01, asp03, ACH+05, abc], SML and C[AMP] that can produce class files. Also, bytecode produced by compilers can be processed by bytecode optimizers which produce faster and/or smaller class files, instrumentors and obfuscators which seek to produce class files that are hard to decompile and understand.

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

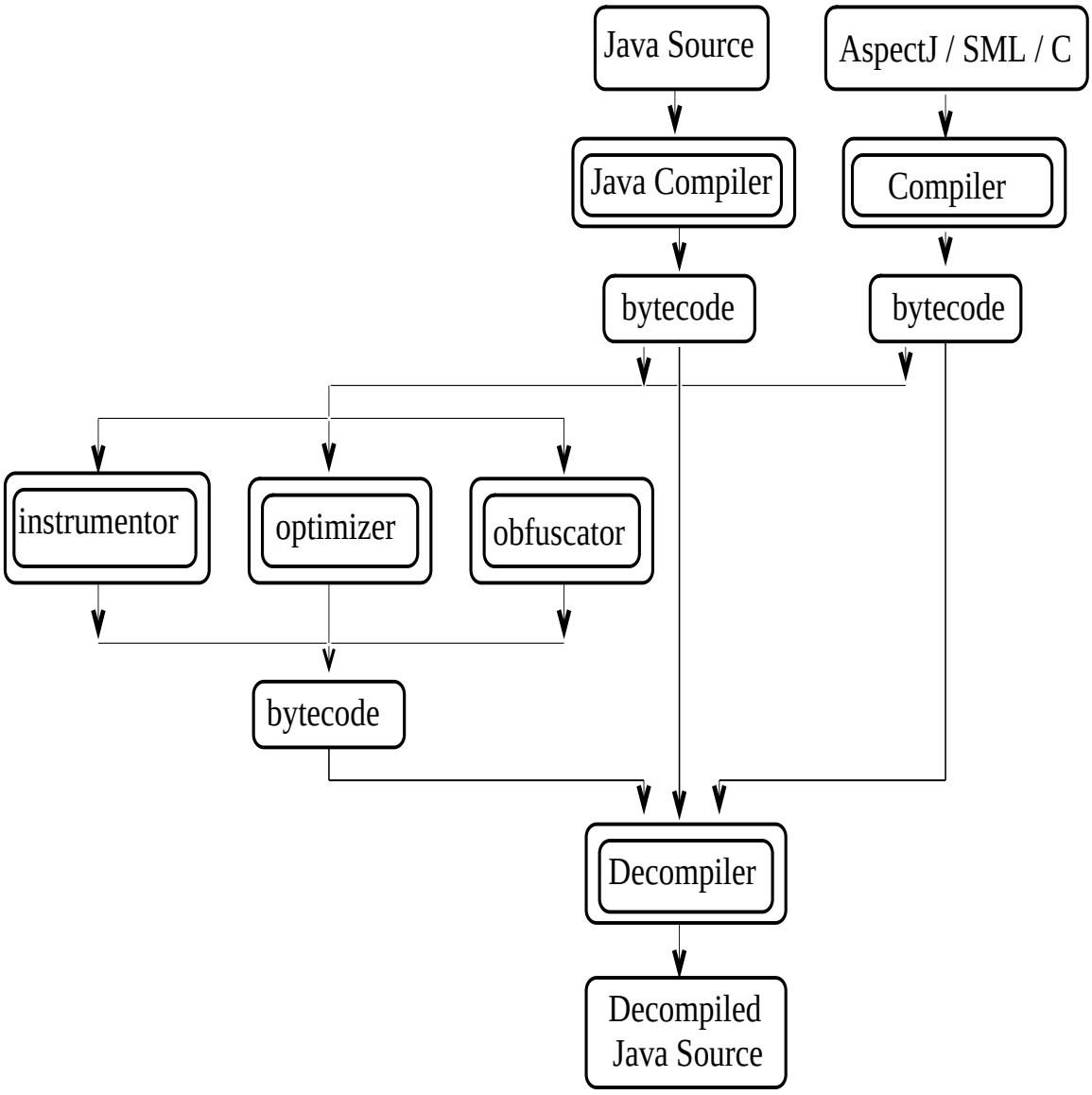


Figure 1.1: *Sources of Java bytecode*

1.1 Javac-specific Decompilers

The original decompilers, such as Mocha[Moc], Jad[Jad], Jasmin[Jas], Wingdis[Win] and SourceAgain[Sou], are *javac-specific decompilers* in that they work by reversing the specific compilation patterns used by the standard `javac` compiler. When given class files produced by a `javac` compiler, they can produce very readable source files that correspond closely to the original program. For example, consider the original Java program in Figure 1.2(a). When this program is compiled using `javac` from jdk1.4 to produce a class file and then decompiled with SourceAgain and Jad, one gets the very respectable results in Figure 1.2 (b) and (c).

By assuming that the bytecode to be decompiled was produced with a specific Java compiler, javac-specific decompilers are able to simplify the decompilation task by reversing the code generation strategy employed by the targeted compiler. By applying pattern matching, inferred from the known code generation patterns of the compiler, the task of creating a javac-specific decompiler becomes relatively easy and fast. Sometimes the patterns applied to get the decompiler output are very specific. For example, compare the results for Jad between the case when the original program was compiled with jdk1.4 (Figure 1.2(c)) and with jdk1.3 (Figure 1.2(d)). Clearly the Jad decompiler was implemented to understand the code generation patterns from `javac` from jdk1.3 and it does not produce as nice an output when used on class files produced using `javac` from jdk1.4. Hence as the code generation strategy of the targeted compiler changes there is a need to update the decompilation patterns in javac-specific decompilers to maintain their performance.

Although javac-specific decompilers perform well for specific compiler-generated code, they are not able to decompile any arbitrary bytecode. This stems from the fact that often the bytecode does not follow the same patterns implemented in the decompiler. This is even more true for bytecode passed through optimizers and obfuscators. In this situation javac-specific decompilers are often not able to produce valid Java code.

(a) Original Code

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

(b) SourceAgain (jdk1.4)

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

(c) Jad (jdk1.4)

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

(d) Jad (jdk1.3)

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

(e) Dava (jdk1.4)

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

Figure 1.2: *Comparing decompiler outputs*

1.2 Tool-independent Decompilers

Dava [MH01, MH02] is a *tool-independent decompiler* built using the Soot [Soo, VRGH+00] Java optimizing framework. Dava makes no assumptions regarding the source of the Java bytecode and is therefore able to decompile arbitrary verifiable bytecode. However, this generality comes with a price. Since the Dava decompiler relies on complex analyses to find control-flow structure in arbitrary bytecode, the decompiled code is often not programmer-friendly. For example, in Figure 1.2(e), the output from Dava is correct, but not very intuitive for a programmer. The goal of this research has been to provide tools that can convert the correct, but unintuitive, output of Dava to a more programmer-friendly output.

1.3 Java Obfuscators

Java obfuscators aim to prevent code comprehension by mostly changing the names of identifiers in the Java bytecode. The first-generation obfuscators replace class, field, method and local variable names with confusing and often misleading names. This kind of obfuscation does not restrict reverse engineering attempts through decompilers.

A new class of Java obfuscators has also emerged that perform control flow obfuscations. These second-generation obfuscators introduce complex, yet verifiable, bytecode which causes most decompilers to fail. Since Dava is a *tool-independent* decompiler and since obfuscated bytecode is verifiable bytecode, Dava is usually able to produce valid Java source for obfuscated code.

The challenge of providing programmer-friendly output for obfuscated bytecode is complex. For example, consider the example in Figure 1.3. In this example we compiled the Java program given in Figure 1.3(a) with `javac` and then applied the Zelix KlassMaster obfuscator[Klaa] to the generated class file. Figures 1.3(b) and (c) show the results of decompiling the obfuscated class file with Jad and SourceAgain (only key snippets of the code are shown). In both cases the decompilers failed to produce valid Java code. However, as shown in Figure 1.3(d), Dava does create a valid Java program, which exposes the extra code introduced by the obfuscator. Even though correct, clearly this code is not very programmer-friendly. This thesis lays down the foundations to address the big challenge of

(a) Original Code

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

(b) Jad

```
     if(flag)/* Loop isn't completed */
        continue;
     s1.equals(s);
     if(flag) goto _L4; else goto _L3
_L3: JVM INSTR ifeq 59;
     goto _L5 _L6
_L5: break MISSING_BLOCK_LABEL_48;
_L6: break MISSING_BLOCK_LABEL_59;
```

(c) SourceAgain

```
  do{ String str = null;
       if( i >= a.size() ){
       //goto couldn't be resolved
         goto 81 }
  }while( !bool );
```

(d) Dava

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


Figure 1.3: *Decompiling Obfuscated Code*

how we can convert the obfuscated code into something that is more readable.

1.4 Thesis Contributions and Organization

Dava's initial implementation focused on correct detection of Java constructs and did not address the complexity of the output. To be useful as a program understanding tool it is important that Dava competes with other decompilers not only in the range of applicability, but also the quality of output. By relying solely on the structure of the flow of control Dava is able to produce Java source code which is semantically equivalent to the original source code for most verifiable bytecode. However, as mentioned earlier (Figures 1.2 and 1.3), the output does not resemble the original source as closely as one would like.

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

- Since the new back-end for Dava works by rewriting the AST we developed a visitor-based AST traversal framework, as outlined in Chapter 3.
- The visitor-based framework can be employed to do simple transformations to conform the output to generally accepted programming idioms as demonstrated in Chapter 4.
- Using the traversal mechanism we developed a large number of simple structural patterns that could be used to perform structural rewrites of the AST. These transformations mainly target the control flow of the decompiled output. Details of these transformations can be found in Chapter 5.
- Simple structural patterns can be used for many basic tasks, but in order to do many more complicated rewrites we needed to have data flow information. Thus, we have developed a structural data flow analysis framework, as outlined in Chapter 6.

- Given the flow analysis information computed using the framework we have developed several more advanced patterns. In Chapter 7 we discuss our advanced patterns for improving the code quality including the use of reaching definitions, reaching copies, constant propagation *etc.* information in transformations.

Chapter 8 discusses new heuristic-based identifier renaming algorithms introduced in Dava to help program comprehension. In Chapter 9 we discuss some metrics to measure the effect of the transformations on the complexity of decompiled output. Empirical results, using the metrics established, are also discussed. Chapter 10 discusses some related work. In Chapter 11 we mention some future work planned for Dava and our conclusions.

Chapter 2
Background: Dava Architecture

Dava is built using the Soot Java bytecode transformation and annotation framework. Soot provides three internal representations (`baf`, `jimple` and `grimp`) to develop and test new compiler optimizations. Java bytecode is first converted to `baf` which is a stack-based representation of disassembled Java class files. Figure 2.1(a) shows a small Java method. In Figure 2.1(b) we show the `baf` representation of this method. As can be seen from the figure the `baf` representation resembles closely to the Java bytecode produced by the compiler. Control flows through the code using labels and `goto` statements and a stack is used to perform operations on data.

`Baf` is then converted to `jimple` which is a 3-address representation of Java bytecode. The most important difference between `baf` and `jimple` is the absence of the Java stack in `jimple`. `Jimple` also uses a static type inference engine to infer primitive and reference types from the Java bytecode [GHM00]. Figure 2.1(c) shows the `jimple` representation of the code in Figure 2.1(a). This representation is the most powerful intermediate representation for performing compiler optimizations like copy propagation and array bounds checks.

The third intermediate representation in soot is `grimp` which stands for aggregated `jimple`. This is the highest level intermediate representation in Soot and is therefore used as input to Dava. Figure 2.2 shows the `grimp` representation of the code in Figure 2.1(a). Control flow in `grimp` is still implemented using explicit labels and gotos. Java's try-catch blocks are represented as areas of protection in the form of exception handlers within the

(a) Original Code

```
1 public int foo(int a,int b){
2   try{ a= a*4+b; }
3   catch(RuntimeException re){}
4   return a;
5 }
```

(b) Baf

```
1 public int foo(int, int) {
2   word r0, i0, i1;
3   r0 := @this: ir;
4   i0 := @parameter0: int;
5   i1 := @parameter1: int;
6   label0:
7     load.i i0;
8     push 4;
9     mul.i;
10    load.i i1;
11    add.i;
12    store.i i0;
13  label1:
14    goto label3;
15  label2:
16    store.r i1;
17  label3:
18    load.i i0;
19    return.i;
20  catch java.lang.RuntimeException
21    from label0 to label1 with label2;
22 }
```

(c) Jimple

```
1  public int foo(int, int){
2    ir r0;
3    int i0, i1, $i2;
4    java.lang.RuntimeException r1, $r2;
5
6    r0 := @this: ir;
7    i0 := @parameter0: int;
8    i1 := @parameter1: int;
9
10   label0:
11     $i2 = i0 * 4;
12     i0 = $i2 + i1;
13
14   label1:
15     goto label3;
16
17   label2:
18     $r2 := @caughtexception;
19     r1 = $r2;
20
21   label3:
22     return i0;
23
24   catch java.lang.RuntimeException
25     from label0 to label1 with label2;
26 }
```

Figure 2.1: *Baf and Jimple representations*

code. The code itself is represented using a reduced set of statements, as compared to Java, which contains aggregated expressions. The reason why `grimp` is chosen as the starting point of the decompilation process is that certain decompilation issues have been already dealt with in the creation of this intermediate representation. As already mentioned, `grimp` is stack-less so the Java expression stack has been eliminated. Also from the type inference engine appropriate types have been applied to all variable declarations.

```
1  public int foo(int, int){
2    ir r0;
3    int i0, i1;
4    java.lang.RuntimeException r1, $r2;
5
6    r0 := @this;
7    i0 := @parameter0;
8    i1 := @parameter1;
9
10   label0:
11     i0 = i0 * 4 + i1;
12   label1:
13     goto label3;
14   label2:
15     $r2 := @caughtexception;
16     r1 = $r2;
17   label3:
18     return i0;
19
20   catch java.lang.RuntimeException from label0 to label1 with label2;
21 }
```

Figure 2.2: *Grimp representation*

In Section 2.1 we discuss the old Dava decompiler to which we have added a new back-end. The front-end takes the `grimp` representation of the Java bytecode as input and

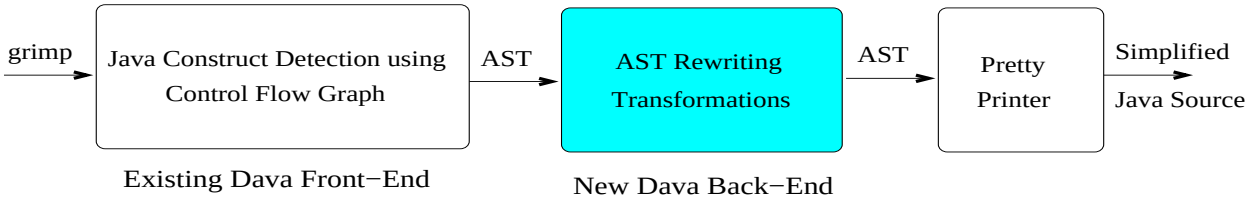


Figure 2.3: *Dava Architecture*

produces an Abstract Syntax Tree representation of the decompiled Java source. Previously this AST used to be pretty printed as the decompiler output. However, this thesis introduces a new back-end to Dava which takes the complicated, through semantically correct AST, and transforms it via AST rewriting to a simplified AST. This modified AST is then pretty printed to produce more programmer-friendly Java source.

2.1 Existing Front-End

The internal workings of the Dava front-end are shown in Figure 2.4. The `grimp` representation of the bytecode is used to create a control flow graph (CFG). Each control flow graph node contains a `grimp` statement with predecessor, successor, dominator and reachability information. The control flow graph is also augmented with exception handling information retrieved from the *traps* information in the Java bytecode.

The next step is the detection of different Java constructs using the CFG as input. It is not feasible to use a reduction-based approach to construct detection because of the large set of isomorphic transformations possible for different Java constructs. Instead Dava employs a unique approach, called staged encapsulation, to retrieve the Java constructs out of the CFG. The strategy involves a series of complicated structuring algorithms which find Java control flow statements based on their semantics rather than their locations relative to other control flow statements. Since these analyses are general and do not resort to pattern matching and/or simulating control flow using state machines, Dava is able to handle highly unstructured `grimp`. This property proves to be crucial during decompiling convoluted code *e.g.*, obfuscated bytecode (Section 7.3.7).

As shown in Figure 2.4, the Structure Encapsulation Tree creation phase can be broken

into three categories:

- Regular Control flow. This include analyses for the detection of `While` and `Do-While` loops and `If` and `If-Else` conditional statements. This is followed by analyses to determine `Switch` constructs and `Labeled-Block` accompanied by the identification of `break` and `continue` statements.

- Exceptional Control Flow. This involves the detection of the `Try-Catch` blocks. As mentioned earlier the CFG has already been augmented with exception handling information available through *traps* in the Java bytecode. Since Java bytecode does not restrict overlapping exception handlers, ensuring that the `Try-Catch` blocks nest properly within each other is a non-trivial task and requires several analyses.

- Idiomatic Control Flow. `Synchronized` blocks are detected in this stage. Although Java bytecode is a high level representation yet there is still a large gap between the bytecode and the Java source that it represents. The `Synchronized` detection attests to this fact. In Java, synchronized blocks are an easy way of providing mutual exclusion. Because of the syntax of the synchronized construct, proper nesting of synchronized blocks is always guaranteed. No such guarantees exist at the bytecode level. Also, since the bytecode represents synchronization using the *entermonitor* and *exitmonitor* bytecodes it has to go through great lengths to ensure that a monitor lock acquired is always released *e.g.*, when an exception is thrown while holding a monitor lock. In short, the bytecode representation of the Java `Synchronized` construct is complicated and a sophisticated graph analysis is required to be able to retrieved the `Synchronized` blocks from the CFG.

As each construct is detected a data structure called the Structured Encapsulation Tree (SET) is constructed. The last stage of the front-end is the creation of the Abstract Syntax Tree. Previously it was this AST which used to be emitted to a file to produce the decompiled Java source. Now the AST is fed into the newly created Dava back-end.

The AST exposes a different form of the constructed Java and allows for further analyses. Since most of the analyses presented in this thesis work on this AST it is useful to familiarize oneself with the constructs making up this tree. The type hierarchy of nodes

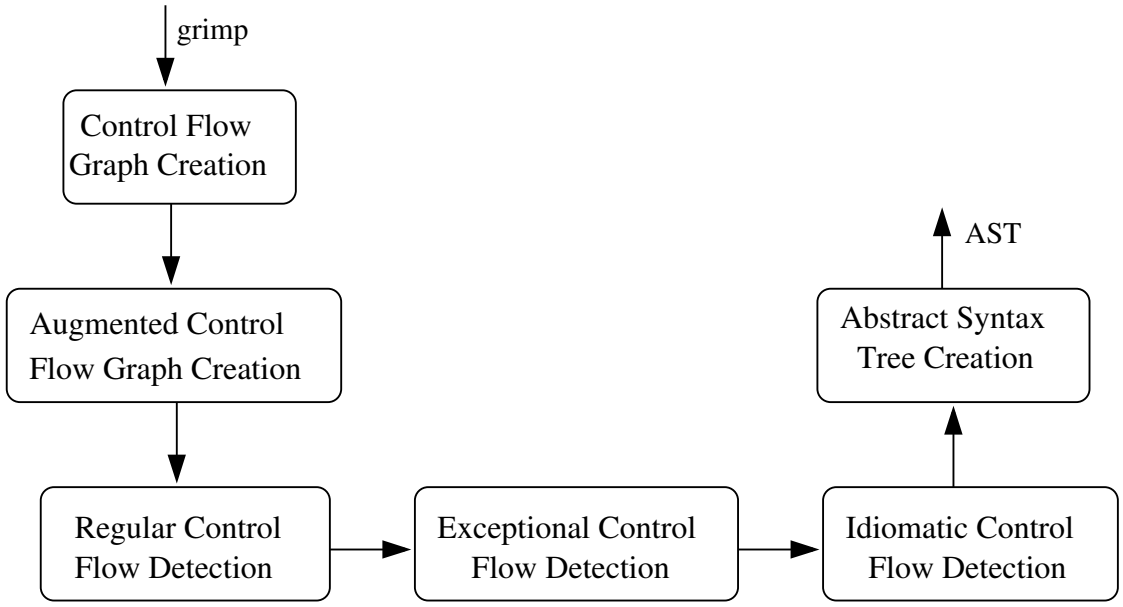


Figure 2.4: *The Dava Front-End*

which can occur inside a AST is shown in Figure 2.5. There is a node for each Java construct. There is also one special node called the `StatementSequence` node which contains the statements present in a particular Java construct. These statements are `grimp` statements which are printed out as Java statements. These include statements like assignment, `breaks` or `continues` etc. The reason for keeping such a structure for the AST nodes is that the nodes are more for the convenience of manipulating different Java constructs and less for carrying actual code.

2.2 New Back-End

As mentioned before, the purpose of this research was to simplify the output produced by Dava. We found that the AST representation of the Java bytecode is the ideal data structure to perform these transformations. Figure 2.6 shows the architecture of the back-end created. The first step is to perform basic transformations on the AST to make it conform more closely to programming idioms. Then simple pattern-based structuring transformations are

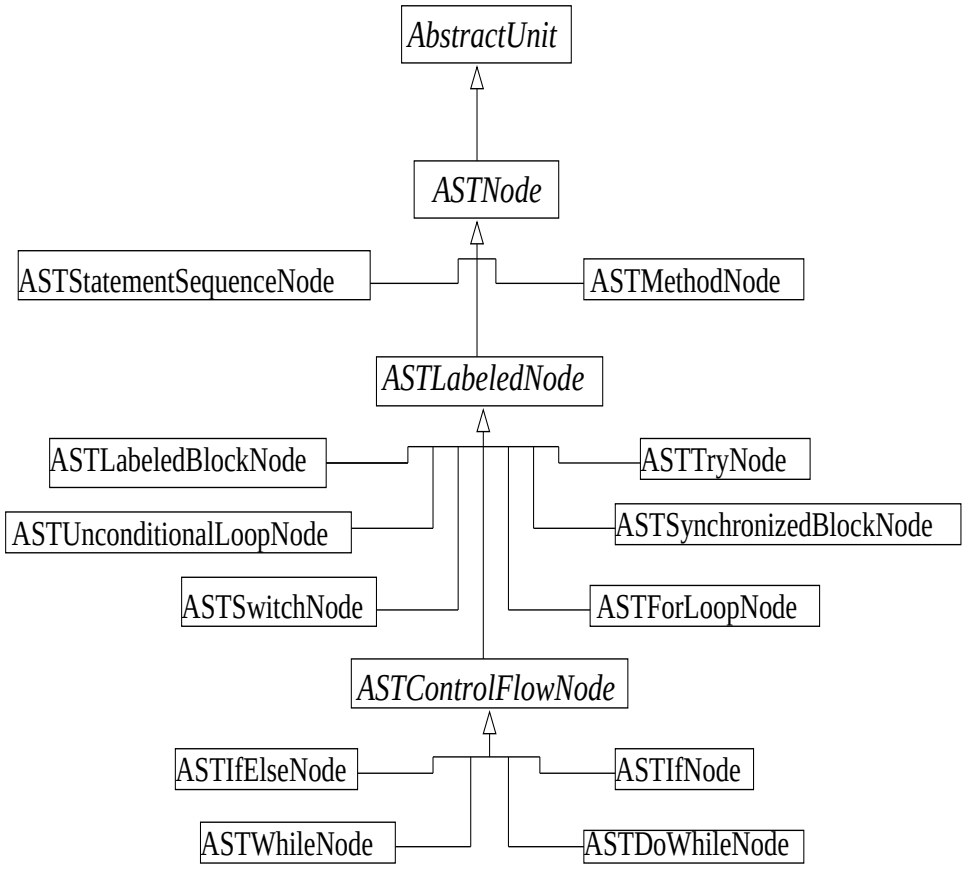


Figure 2.5: *Abstract Syntax Tree Class Hierarchy*

applied. The transformations detect the occurrence of certain sequences of AST nodes and replace them with modified nodes representing simplified Java constructs and/or control flow. However, it was noted that simple pattern-based transformations are not powerful enough in many instances. The third stage in the back-end employs a series of transformations enabled using flow-analysis information.

The application of patterns in the second or third stage of the restructuring can enable new transformations. The simple pattern-based structuring along with the flow-analyses-based transformations are therefore applied iteratively until no pattern matches. By carefully ordering the transformations and ensuring that transformations always move towards a fixed point we are guaranteed that the iterative application of transformation will terminate.

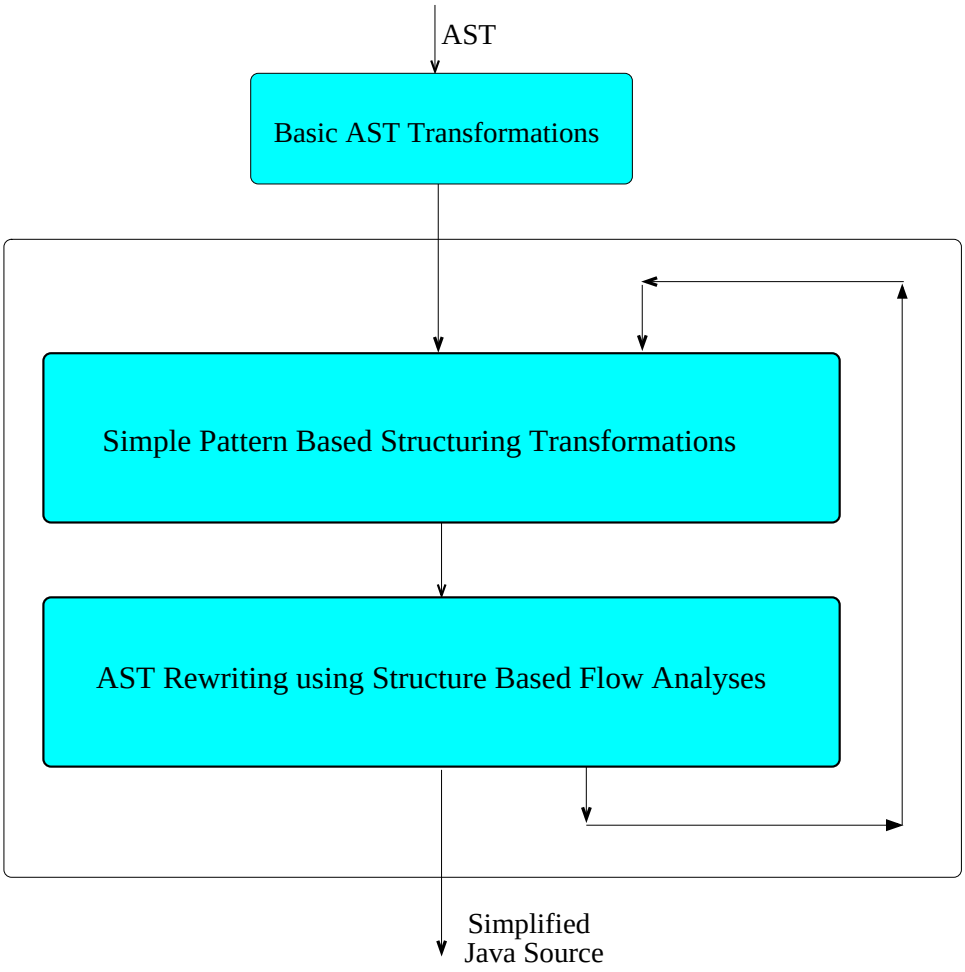


Figure 2.6: *The Dava Back-End*

Chapter 3
A Tree Traversal Algorithm

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

Inspired by the traversal mechanism provided by SableCC[GH98], tree walker classes were created using an extended version of the Visitor design pattern. The Visitor-based traversal allows for the implementation of actions at any node of the AST, separately from AST creation. This allows for modular implementation of distinct concerns and a mechanism which is easily adaptable to the needs of different analyses.

The traversal mechanism also provides `IN` and `OUT` methods which are invoked by the Visitor design pattern when entering and exiting each subtree node, respectively. Using these methods makes the task of subtree rewriting, needed all the time for transformations, a simple matter of overriding the appropriate method. Usually the transformations use the `IN` methods to gather information regarding the node being traversed. Future transformation decisions might use the information stored at this point. If a decision to modify the AST is made then often the `OUT` method is used to perform the transformation.

An example of the usefulness of the extended Visitor design pattern is the detection, and subsequent removal, of spurious `Labeled-Blockss`. A `Labeled-Block` is spurious if it encapsulates code that never targets the `Labeled-Block`. The Visitor design pattern provides an elegant way of implementing this transformation. Very briefly, such a transfor-

mation can be implemented as follows.

The `IN` method for entering a `Labeled-Block` is overridden and the label is stored in a data structure used to store all "active" labels. The traversal then continues with visiting the children of the `Labeled-Block`. The `IN` method of `break` statements is overridden (Note: only `break` statements can target a `Labeled-Block`). If the `break` statement explicitly targets a label then that label, from the list of active labels, is marked as `needed`. The `OUT` method of a `Labeled-Block` block is also overridden. This method checks whether it's label has been marked as `needed`. If unmarked, this indicates that there was no `break` statement targeting the `Labeled-Block` and hence the block is spurious and can be removed.

```
List activeLabels = new ArrayList();
List neededLabels = new ArrayList();

public void inASTLabeledBlockNode(ASTLabeledBlockNode node){
   activeLabels.add(node.getLabel());
}

public void inBreakStatement(BreakStatement stmt){
    NodeLabel label = stmt.getLabel();
    if(activeLabels.contains(label){
          neededLabels.add(label);
    }
}

public void outASTLabeledBlockNode(ASTLabeledBlockNode node){
    if(neededLabels.contains(node.getLabel)== false){
        //spurious labeled block detected
        //use AST rewriting to remove the labeled block
     }
}
```

Figure 3.1: *Pseudo-code for sample tree-traversal*

Apart from allowing transformations on the AST, the Visitor mechanism can also be used to gather information for other transformations/analyses to use. In the remaining sections of this chapter we discuss some of the tree traversals that have been implemented which play a supporting role for other transformations.

3.1 Finding AST Parent Nodes

The Parent-Node Finder traversal is responsible for gathering information regarding the different constructs in the AST. The class produces a HashMap, keyed by a node in the AST and the parent of this construct as the value. In terms of this traversal a construct is either a Java construct *e.g.*, `If`, `Do-While` *etc.* or any `grimp` statement present within the `Statement-Sequence` node of the AST.

This analysis is required since transformations often traverse the AST and, at some stage during the traversal, decide that a particular node has to be moved/replaced. Since such a modification requires ancestor information it might have been a good idea to store a parent pointer within each of the AST constructs. As the original implementors of Dava had not intended to perform AST analyses this information is currently not present in the AST class definitions. One option would have been to go through the code that creates and manipulates AST nodes and add parent information. Instead we chose to write this helper analysis which can be used to get appropriate parent information whenever needed.

The traversal algorithm works as a wrapper around the AST. It can be queried at any time during a transformation to provide ancestor information. An example of the use of this helper traversal is in the case of copy elimination (Section 7.2.1) where to remove a particular copy statement the `Statement-Sequence` node containing this statement has to be found.

3.2 Finding the Closest Abrupt Target

Java programs contain two types of abrupt control flow statements: `continue` and `break`. The `continue` statement is used to terminate the current iteration of the closest loop. On encountering a `continue` statement the program execution continues with the re-evaluation

of the condition of the loop. For the case of `For` loop the update statements are executed before the evaluation of the condition.

The `break` statement can be used to terminate the execution of not only the closest loop but also the execution of the closest `Switch` statement. In each case the program execution continues from just after the end of the statement broken.

The semantics discussed above are for *Implicit* `break` and `continue` statements. Java also has *Explicit* `break` and `continue` statements. These are statements of the form: `break labelN;` and explicitly target a labeled construct within the code. With explicit `breaks` the program execution breaks the labeled construct explicitly stated in the statement. Explicit `breaks` are more powerful in the sense that they can be used to break from any Java construct which has a label. In our implementation this would mean all ASTNodes inheriting from the ASTLabeledNode (Figure 2.5). Explicit `continues` on the other hand do not introduce new statements that can be targeted by `continues`. The advantage of explicit `continues` is that these can be used to break out of an outer loop from within an inner nested loop.

Finding the targets of `explicit` abrupt statements is easy since the label targeted is explicitly mentioned in the abrupt statement. However, in the case of an *Implicit* `break` or `continue` statement the construct targeted has to be tracked by moving up the AST. A traversal was implemented which keeps track of the current construct that might be targeted by an *Implicit* abrupt statement (a stack where targetable nodes are pushed when entering the node and popped when exiting them). A mapping is created where the key is the abrupt statement and the value the current targetable construct (top of the stack). This information can be used by other analyses and is also used internally within the structure based flow analysis framework (Chapter 6).

3.3 Finding all variable Uses

A depth first traversal of the tree is utilized to find all the uses of a local variable within a method. Similarly, all the uses of a field within a particular method can also be found. The results of the traversal can then be queried. Given a local or field as the key, the results provide a list of all places where this variable might be used. A number of transformations

e.g., ensuring that final fields get defined on all paths and only once (Section 7.4.1), use these results.

3.4 Finding all Definitions

Another trivial analysis, this gathers a list of all definitions (assignments to locals or fields) within a method. This information is used by a number of analyses including the `newInitialFlow` implementation of the reaching defs flow analysis (Section 7.1). The following tree traversal analysis is another analysis which uses the definitions found by this analysis to gather further information.

3.5 Constant Primitive Field Value Finder

This analysis finds all primitive fields that have a constant value throughout the execution of a program. This information helps to give the extra information needed for more accurate constant propagation as discussed in Section 7.3.2.

The algorithm is a two-step process. In the first step all definitions for all fields with primitive type in the application are collected. The all definitions finder analysis, discussed in the previous subsection, is used to return a list of all definitions in each method. Definitions to non-primitive fields are removed. At the end of this stage a list is created containing all places in the code where the field might be assigned.

The second step processes each field one at a time. Algorithm 1 shows this stage. As mentioned earlier, the analysis only tracks values of fields with primitive types. Java compilers store constant values for static final fields inside the constant pool. The SOOT framework converts these constant values to tags to which Dava has access. Hence the first step for a primitive type field (as shown in Algorithm 1) is to look up whether there is a constant value tag for this field. If one is found, the constant value tag provides the value for this field. If not, then the list of definitions found in stage one of the analysis is checked. If there is no definition for this field that means the field is never assigned a value. We can therefore assume that the field gets the default value for this primitive type field *i.e.*,

booleans get `false` and others get zero. We can hence return the default constant value for this field.

If there were some assignments to this field then the algorithm checks that all the assignments are default value assignments. This check must be made because a context-insensitive inter-procedural analysis does not keep track of the order of execution of statements. Hence a claim for the value of a field, after the execution of an unordered set of assignments to the field, can only be made if all assignments assign the same value to the field. Further, since a field might not be initialized, at declaration time, in which case it is assigned the default value, a claim can in fact only be made if all the assignments to a particular field are default values.

The end result of this analysis is a list of fields which always have the constant values. This can include fields which are final and hence are by definition constant or fields which are either never assigned or are always assigned the default value.

Algorithm 1: Finding constant valued fields

Input: SootField *field*, List *defList*

Output: Constant value if found else null

```
//Only deal with primitive fields
if !(field.getType() instanceof PrimType ) then
    return null
//static final fields have constant value tags
if hasConstantValueTag(field) then
    return getConstantValueTag(field)
//if field is never assigned
if deflist.size() == 0 then
    return createDefaultValue(field.getType());
else
    //field is assigned some value within the code
    forall definitions d, in defList do
        //Assignment should only be default assignment
        if !d.isDefaultAssignment() then
            return null;
    end
    //All assignments were default
    return createDefaultValue(field.getType());
```

Chapter 4
Basic AST Transformations

The ability to traverse the AST, using a Visitor-based design pattern, allows for modular implementation of transformations. New traversals of the AST checking for simple patterns can be implemented and plugged into the Dava back-end by inserting a call to the new transformation in the already executing list of transformations. Given the traversal mechanism, at a bare minimum, the mechanism can be used to transform Dava's output to produce code conforming more closely to programming idioms.

Programming idioms are common programming practices among the programmer community. These are highly subjective since they deal with a programmer's personal preference and style of coding. Nevertheless, in this section, we discuss some programming idioms which, in our view, make program comprehension easier.

4.1 Condition Simplification

Expressions evaluating to boolean types are often used as unary conditions. An artifact of the restrictive condition grammar in Dava (Figure 5.2) resulted in representation of such boolean expressions as binary operations, comparing the expressions to the boolean constants `false` or `true`.

Figure 4.1 shows the different conversions that can be carried out. Since most programmers are used to reading boolean expressions in the form of unary conditions the effect of these transformation is that code becomes less verbose and easier to read.

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

Figure 4.1: *Converting Binary Conditions to Unary Conditions*

Applying this pattern on our working example of Figure 1.2(e) results in the simplification of the two boolean conditions in Statement 3 and 4.

4.2 Shortcut increments and decrements

Another simple transformation for ease of reading code is the use of shortcut increment and decrement statements. It is common practice to represent the increment statement i = i + 1 using the increment operator ++ and using a similar decrement operator for the i = i - 1 statement. This transformation replaces occurrences of i = i + 1 with i++ and i=i-1 with i- -. A more general case for this is when a variable is updated using the previous value of the variable along with a constant. For example, the expression x = x + 2 is converted to x += 2.

4.3 De-Inlining Static Final Fields

Standard Java compilers inline the use of static final fields. The reasoning is that since the field is final the value is not going to change and hence the constant value can be used in the bytecode instead of having to look up the value from a class attribute. The decompiled output therefore contains the constant values wherever there was a static final field in the original code. We think it is a good idea to recover the use of the field that was used in the original code since the name of the field might be able to deliver some contextual information to the programmer. A transformation was written which keeps a pool of all

static final fields and their corresponding values found in a particular class. A depth first traversal is then carried out that checks for the occurrence of constant values in the code. When a constant value is encountered it is checked with the list of known values for the different static final fields. If there is a match then the use of the constant value is replaced by the use of the static final field. For example in Figure 4.2(a) the createMinArray method returns a new array with size 5. However, a static final MINSIZE is also declared with the value 5. The De-Inlining transformation will detect this occurrence and generate code as shown in Figure 4.2(b). This kind of transformation allows for more use of identifiers in the code and the contextual information provides the programmer insight into the code.

(a) Inlined field

```
1 static final int MINSIZE = 5;
2
3 public int[] createMinArray(){
4    return new int[5];
5 }
```

(b) De-Inlining

```
1 static final int MINSIZE = 5;
2
3 public int[] createMinArray(){
4    return new int[MINSIZE];
5 }
```

Figure 4.2: *DeInlining Static Final Variables*

4.4 Variable Declarations and Initialization

Dava was previously unable to convert multiple variable declarations into a single declaration. Also, previously a declaration and the subsequent initialization of the variable was always broken into two consecutive statements (Figure 4.3(a)). A transformation now aggregates variables with the same type into one declaration. Also a variable which is initialized as soon as it is declared can now be initialized as part of the declaration (Figure 4.3(b)). This is a common programming idiom and makes the code more natural.

(a) Unreduced

```
1 int a;
2 int b;
3 b=3;
4 int c;
```

(b) Reduced

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

Figure 4.3: *Variable Declarations and Initialization*

4.5 String concatenation

String concatenation in Java can be carried out using the overloaded + operator. The semantics of the operation allows for the addition of a String to a primitive type or any object (whose `toString` method is automatically invoked to get its String representation). For instance the argument "hello" + 5 represents the concatenation of the `String` "hello" with the `String` representation of the integer 5. In bytecode this conversion is achieved by using the `StringBuffer` class. A new `StringBuffer` is created whenever `String` coercion is required and the operands to the *addition* operator are `appended` to the `StringBuffer`. The final output is the `toString` of the `StringBuffer`. For instance the argument "hello" + 5 would be represented as
`((new StringBuffer()).append(‘‘Hello’’).append(5).toString()).`
We have implemented a transformation that looks for this pattern and retrieves the arguments to the chained `append` methods. From there the argument is reconstructed using the + operator.

A common occurrence of this is the `System.out.println` method invocation, used to output information. Programmers normally pass, as argument to this method, the expression which might contain implicit String coercion using the overloaded + operator. With this transformation we are able to retrieve the original expression written by the programmer. Figure 4.4 shows such an example where the verbose code previously generated by the decompiler has now been simplified using the + operator. In our view this makes the

code much easier to read and adhere more closely to general programming practices.

(a) Unreduced

```
1 System.out.println(
2    (new StringBuffer()).append(‘‘Hello’’).append(5).toString())
```

(b) Reduced

```
1 System.out.println(‘‘hello’’+5)
```

Figure 4.4: *String Concatenation*

4.6 Shortcut Array Declarations

Arrays can be initialized using the shortcut declaration and initialization statement. For example an array of the first five primes can be declared using: int[ ] primes = {1,2,3,5,7}; When compiled the Java bytecode represents this as the initialization of an array of size 5 followed by the assignment of each of the five elements of the array. The decompiled output for the `primes` array, as represented in the bytecode, is shown in Figure 4.5(a).

A pattern has been devised which converts the verbose array initialization code of Figure 4.5(a) to the shortcut array declaration shown in Figure 4.5(b). Algorithm 2 shows the transformation which looks for this pattern. The algorithm is self-explanatory. Briefly, we start by looking for a statement which creates a new array. If one is found then we find whether the length of the array is a known constant. This is important since we can only use the shortcut array initialization statement if all elements of the array are being initialized.

If the size of the array is known then we check the subsequent statements. If all of them initialize the *appropriate* element location *i.e.*, the elements are initialized in order, the

(a) Unreduced

```
int[ ] primes = new int[5];
primes[0] = 1;
primes[1] = 2;
primes[2] = 3;
primes[3] = 5;
primes[4] = 7;
```

(b) Reduced

```
int[ ] primes = {1,2,3,5,7};
```

Figure 4.5: *Verbose declaration of the* `primes` *array*

pattern is matched. The verbose array creation and initialization statements are removed and replaced with the shortcut declaration and initialization statement.

4.7 Removing default constructors

A Java class does not need to have a declared constructor if certain conditions exist. These are: the presence of only one constructor and the constructor being the default constructor *i.e.*, the constructor takes no arguments and executes no code except for the invocation of the default `super` constructor. When a class containing no constructor is compiled, Java compilers produce the default constructor as the <`init`> method which is then invoked in the bytecode whenever an object of this class is created.

When decompiling a class with a default constructor the reverse approach can be taken. If the bytecode contains only the default constructor then this constructor can be removed. Algorithm 3 shows in pseudo-code the process of checking whether a constructor can be removed from the class definition.

The algorithm starts off by finding all constructors defined by the class. If there is more than one constructor the algorithm quits since in the presence of an overloaded constructor along with the default constructor we cannot predict that all objects will invoke the overloaded constructor. If there is only one constructor then it is checked whether this is the

Algorithm 2: Shortcut Array declaration and initialization

```
Input: ASTStatementSequenceNode node
List stmts = node.getStatements()
Iterator it = stmts.iterator()
while it.hasNext() do
    Stmt s = it.Next()
    if ! (s. containsNewArrayExpr()) then
        //First stmt of pattern should contain a new array creation
        continue
    if ! (s. getArrayExpr().getSize() instanceof IntConstant ) then
        //Can only apply pattern for arrays declared with known size
        continue
    int length = s. getArrayExpr(). getSize()
    for int i=0;i<length;i++ do
        if ! (it.hasNext() ) then
            //Not all array elements initialized
            transform = false
            break
        Stmt temp = stmts.get(stmts.indexOf(s) + i)
        if stmt temp does not initialize index i of array then
            //Can't continue since we require inorder initialization of elements
            transform=false
            break
    end
    if transform then
        //Remove statement s and the following length number of stmts
        //Create the new shortcut declaration and initialization stmt
        //Add statement to position currentIndex in the statements list
end
```

Algorithm 3: Removing the Default Class Constructor

```
Input: SootClass sootClass

constructorList ← sootClass.RetrieveConstructors()
if constructorList.size() != 1 then
    //class contains more than one constructor
    return;
end
SootMethod constructor = constructorList.get0
if constructor.getParameterCount() != 0 then
    //constructor not the default constructor
    return;
end
Body methodBody = constructor.getActiveBody()
if ! methodBody.isEmpty() then
    //constructor doesnt have an empty body
    return;
end
InvokeExpr superInvocation = methodBody.getConstructorExpr()
if superInvocation.getArgsCount() != 0 then
    //super invocation not the default invocation
    return;
end
//all conditions fullfilled. Remove the constructor
sootClass.removeMethod(constructor)
```

default constructor *i.e.*, it has no arguments in its method signature. If we do find that the only constructor has no arguments in its signature then the method body's contents are checked. If there is no code, except for the default `super` invocation, we can continue with the removal algorithm.

A related improvement in the output produced is the removal of default `super` invocations from a constructor's body. Whenever a Java constructor is invoked, if a `super` call is not explicitly present as the first statement in the method body, the default `super` con-

structor is automatically invoked. Hence, if a constructor has an explicit `super` call to the default parent constructor then this statement is redundant. Such an invocation is therefore removed from the constructor body. Obviously this only works when the explicit `super` invocation is the default invocation *i.e.*, a super invocation without any arguments.

4.8 The super invocation

The Java specification requires any call to a constructor (`super()` or `this()`) to be the first statement in a constructor's body. Since such a restriction does not exist at the bytecode level, the bytecode representation of a constructor can have code preceding the invocation of the `<init>` method.

Even though one cannot write statements before the invocation of `this()` or `super()` in Java, the compilation of a method might result in bytecode being placed before the invocation of another constructor from within the constructor's body. For instance if the code in Figure 4.6(a) is compiled, the produced bytecode (Figure 4.6(b)) has the invocation of the `iterator` method before the call to `<init>` (Statements 5 to 10 in Figure 4.6(b)). Sections 4.8.1 discusses this in more detail and Section 4.8.2 discusses similar issues for bytecode produced by an AspectJ compiler.

Naively decompiling such code would result in uncompilable code unless the statements added before the invocation of the parent constructor are handled appropriately. Section 4.8.3 discusses the solution implemented in Dava.

4.8.1 Invalid code using complicated expressions

Figure 4.6(a) shows two classes A and B where B extends A. The constructor of B invokes the parent constructor using the Java `super` statement. However, within the arguments of the `super` call the `Iterator` "it" is being assigned the same value as the argument being sent to the parent constructor. This is valid Java code since the super class `A` expects an `Iterator` as the argument to its constructor. Also since the call to `super` is the first statement in the constructor of `B`, the code will compile since all Java requirements are satisfied.

(a) Original Code

```
1 class A{
2   public A(Iterator it){
3   }
4 }
5 class B extends A{
6   public B(List list,
7            Iterator it){
8     super(it=list.iterator());
9   }
10 }
```

(b) Jasmin Code

```
1 .method public <init>
2  (Ljava/util/List;Ljava/util/Iterator;)V
3  .limit stack 3
4  .limit locals 3
5  aload_0
6  aload_1
7  invokeinterface java/util/List/iterator()
8                    Ljava/util/Iterator; 1
9  dup
10 astore_2
11 invokespecial
12          A/<init> (Ljava/util/Iterator;)V
13 return
14 .end method
```

Figure 4.6: *Complex Expressions*

At the bytecode level, the call to `super` is converted to a series of bytecodes which first evaluate the argument of the call to `super` and then invoke the `super` method. The evaluation of the argument results in the invocation of the `iterator` method of the `List` class and the assignment of the result to the constructor parameter “it”. This evaluation is shown in Figure 4.6(b) by statements 5 to 10. Statement 11 is the invocation of the constructor of the parent class.

As expected given the bytecode in Figure 4.6(b) Dava produces the output shown in Figure 4.7. Statements 2 and 3 are the evaluation of the argument and statement 4 is the invocation of the parent constructor using the evaluated argument. The decompiled output is correct decompilation of the bytecode but is incorrect Java code since the call to the parent constructor (Statement 3) is not the first statement of the method. The decompiled code obviously does not recompile.

It can be argued that since the original code was able to represent the evaluation of the

```
1 public B(List r1, Iterator r2){
2   Iterator r3;
3   r3 = r1.iterator();
4   super(r3);
5 }
```

Figure 4.7: *Uncompilable code due to incorrect placement of* `super`

argument within the invocation of the parent constructor, the decompiler should be able to reconstruct the expression as an argument to the parent constructor. This is indeed correct. However, there can be other occurrences where code might get added before the invocation of the parent constructor which we discuss in the next section.

4.8.2 Invalid code using `Preinitialization` in AspectJ

AspectJ[KHH+01, asp03, ACH+05, abc] is an aspect-oriented extension to Java. It enables clean, modular implementation of cross-cutting concerns such as logging and error handling. The AspectJ language introduces a set of constructs, called pointcuts, which can be used to pinpoint locations in the execution of code where the behavior of the program can be altered if the need be. One such construct is the `preinitialization` construct. Using this construct the programmer is able to target the point just before the execution of the super() call within the execution of a constructor. The programmer can weave `advice` at this point which is executed whenever this execution point is reached. One possible kind of advice is `before` advice which lets the programmer add code to run before the matched point in the program. The result of weaving `before` advice on a pointcut using the `preinitialization` construct maps to adding code at the start of the constructor. If however, in the original constructor the first statement was an invocation of another constructor (parent or own class) the advice is added before this invocation. This is exactly what Java disallows. An example is shown in Figure 4.8(a). There are two classes A and B where B extends A. B's constructor invokes `super` and then executes a print statement. An

aspect is then introduced which prints out the string “before preinit” before the invocation of the parent constructor. Using an AspectJ compiler such as `abc` [abc] the two classes and the aspect are compiled.

(a) Original Code

```
1 class A {
2   public A(){
3   }
4 }
5 class B extends A {
6   public B(){
7      super();
8      S.O.println("hello");
9   }
10  public static void main (
11                 String[] args){
12     B b = new B();
13  }
14 }
15 aspect Aspect {
16    before() :
17       preinitialization(B.new()) {
18      S.O.println("before preinit");
19    }
20 }
```

(b) Runtime Behaviour

```
1 before preinit
2 hello
```

(c) Dava’s Output

```
1 public class B extends A{
2   public B(){
3     Aspect.aspectOf();
4     S.O.println("before preinit");
5     super();
6     S.O.println("hello");
7   }
8   public static void main
9                  (String[] r0){
10    B r1;
11    r1 = new B();
12  }
13 }
```

Figure 4.8: *Effect of a preinitialization pointcut targeting a constructor with before advice*

The execution of class B’s main method results in the creation of an object of type B. The output from this is shown in Figure 4.8(b). Notice the string “before preinit” gets printed before the string “hello”. The reason being that the advice is executed before the call to the parent constructor. The decompiler output produced for the constructor of class B is shown in Figure 4.8(c). Statements 1 and 2 are the `before` advice, followed by invocation

of the parent constructor in statement 3 and then statement 4 is B's constructor's remaining body. Clearly this decompiled code produces the same output as in Figure 4.8(b), however, it is not compilable code. The reason being that the invocation of the parent constructor is not the first statement in B's constructor.

Before talking about correcting Dava's output it is worth mentioning that both Jad and SourceAgain also fail to produce correct code. Although Jad fails to decompile AspectJ produced Java bytecode most of the time, it is able to decompile the bytecode produced by the simple example in Figure 4.8(a). The output in this case is exactly that of Dava's (Figure 4.8(c)). SourceAgain does produce compilable Java code but with one major flaw. Its output contains only Statements 1, 2 and 4 of Figure 4.8(c). So in this case although the output produced by SourceAgain is compilable it is semantically not equivalent to the bytecode being decompiled. In our view this output is even more incorrect than the uncompilable code produced by Dava and Jad.

4.8.3 Transforming invalid code using indirection

To avoid compilation errors produced by Java compilers the `super()` or `this()` invocation needs to always be the first statement in a constructor's body The most elegant solution for this is to execute the extra code as an argument to the constructor invocation. This is illustrated in Figure 4.9 which shows a class A and its constructor. The code in (a) shows invalid pseudo-code because of the presence of the offending statement chunk marked X before the invocation of `super`. This error has been fixed in (b) by moving the offending code, X, as an argument to `super`. In the remaining section we deal with the algorithm implemented in Dava dealing with moving the chunk X as an argument to `super`.

Code X in Figure 4.9(a), which we want to execute as an argument to `super`, can be any arbitrary code. It could be the complex evaluation of an argument (as discussed in Section 4.8.1) or it could be some code added by the application of some advice (Section 4.8.2). Hence, it is not possible to handle all cases as an evaluation of an expression in an argument to `super`. Instead a method invocation, executing X as an argument to `super`, is required. Let's name this method `PreInit`. Also, if we want this method to be part of the same class which contains the constructor with the compiler error (class A in Figure 4.9(a))

(a) Invalid pseudo code

```
class A{
  public A(<args1>){
      ---------
   X --------- //code causing
      --------- //compilation error
      super(<args2>)
      ---------
   Y ---------
      ---------
  }
}
```

(b) Valid pseudo code

```
public A(<args1>){
    super(<args2>, X)
    ---------
 Y ---------
    ---------
}
```

Figure 4.9: *Avoiding compilation errors due to* `super` *invocation*

this method needs to be static. This is so because a non-static method of a class cannot be invoked until the constructor of the class has finished executing. Figure 4.10(a) shows the creation and invocation of the `PreInit` method. Code X is executed as the body of method PreInit.

However, it might not be possible to introduce a new constructor in the parent class. An example of this is when the parent class is a library class which one does not have access to or one does not want to change. This issue is handled by introducing a new constructor in A which takes one extra argument (marked `SOMETYPE` in Figure 4.10(b)). The old constructor invokes this new constructor with the `PreInit` method as the last argument. The `PreInit` method executes code X and returns `SOMETYPE`. The remaining code (`super` and code Y) from the old constructor is moved into the newly created constructor.

Another issue is that copying code X into the newly created method may result in undefined variables. The code in X could be using any of the arguments of the constructor (args1 in Figure 4.10(a)). This is handled by passing args1 *i.e.*, all arguments to the old constructor as arguments to the newly created `PreInit` method.

(a) Using a static method to execute code X

```
class A {
  public A(<args1>){
      super(<args2>,A.PreInit())
      ---------
   Y ---------
      ---------
  }
  private static void PreInit(){
      ---------
   X ---------
      ---------
  }
}
```

(b) Creating a new constructor

```
public A(<args1>){
   this(<args1>,A.preInit(<args1>));
}
public A(<args1>,<SOMETYPE>){
    super(<args2>);
    ---------
 Y ---------
    ---------
}
private static SOMETYPE
               PreInit(<args1>){
    ---------
 X ---------
    ---------
   return SOMETYPE;
}
```

Figure 4.10: *Introducing the private static PreInit Method*

Now let us discuss what the return type (`SOMETYPE` in Figure 4.10(b)) should be. One thing to note is that it is quite possible that code X, which is executed before the invocation of `super`, could define some variables that are part of the arguments to `super` *i.e.*, part of args2. Hence `SOMETYPE` needs to be a data structure which can be used to return all possible arguments in args2.

Also since args2 can be zero or more arguments we want `SOMETYPE` to be a data structure which can return a list of arguments. Different arguments of args2 could then be retrieved from within this data structure using a `get` method. However, there is also another consideration: the newly constructed constructor, the one which has parameters args1 and `SOMETYPE` needs to be unique. It is therefore not possible to use any existing Java library `collection` class as `SOMETYPE` since then we stand a chance of creating a constructor with

a signature which might already exist. For example, if the original args1 had an integer type as the only argument and we chose an ArrayList as `SOMETYPE` then the new constructor would have two arguments, an integer followed by an ArrayList. There is a possibility, however remote, that such a constructor already exists in the class.

In order to avoid such an occurrence we decided to create a new data structure with a unique type for the application. The data structure is a wrapper class for the Java Vector class. This new class, named `DavaSuperHandler`, allows the method `PreInit` to store all the args2 and return them as an argument to the new constructor. We are also guaranteed that the signature of the new constructor will not match any existing constructor as we just created `DavaSuperHandler` which is the last argument of the constructor.

Figure 4.11 shows the `PreInit` method with `DavaSuperHandler` as its return type. Also the new constructor has `DavaSuperHandler` as its last argument. In `PreInit` the method stores args2 into `handler` before returning this object. This is possible since all of these arguments are either any of args1 or any variable declared or defined in the code X. The new constructor retrieves these arguments using the `get` method defined in the `DavaSuperHandler` class.

With these changes to the code the old constructor now executes X followed by a call to super and then body Y. To see this lets follow the chain of events. When the old constructor is invoked this results in the invocation of the new constructor. However before the new constructor is executed all arguments to the constructor have to be evaluated. We have added to the arguments our own `PreInit` method which causes code X to get executed. Once this code is executed all values of args2 are packaged in a `DavaSuperHandler` object and made available to the new constructor. The new constructor then invokes `super`. In its arguments it retrieves the args2 stored within the `DavaSuperHandler` object returned by `PreInit`. Once `super` has executed, then code Y is executed. This chain of execution satisfies the language rules since the first statement in the old and the new constructor are always either an invocation of another constructor of the same class or an invocation of the parent constructor. We also make sure that the new constructor's signature does not conflict with any existing constructor and also that the `PreInit` method is uniquely defined in the class.

```
1  public A(<args1>){
2     this(<args1>,A.preInit(<args1>));
3  }
4  public A(<args1>,DavaSuperHandler handler){
5     super(handler.get(0), .... handler.get(n));
6     /*
7      * where n are the number of arguments in
8      * the super invocation
9      */
10     ---------
11  Y ---------
12     ---------
13 }
14 private static DavaSuperHandler PreInit(<args1>){
15     ---------
16  X ---------
17     ---------
18    DavaSuperHandler handler = new DavaSuperHandler();
19    //code to store args2 into handler comes here
20    return handler;
21 }
```

Figure 4.11: *Storing and Retrieving args2*

Chapter 5
Simple Pattern Based Structuring

Most of the transformations implemented to simplify the output are targeted towards control flow simplifications. These are handled as the second stage of Dava's back-end after the application of basic programming idioms to the AST. Figure 5.1 shows the sequence of application of the transformations in this stage. As seen from the figure, control flow simplifications can be broadly divided into three main categories: conditional aggregation, loop strengthening and handling abrupt control flow. The order of application of these transformations has been carefully chosen in order to maximize the number of patterns matched using the minimum number of traversals through the tree. However, even with this ordering, traversals sometimes have to be reapplied since the matching of a later pattern might enable a preceding transformation. In the next three sections we discuss, in detail, the patterns and related transformations.

5.1 Conditional Aggregation

The cryptic control flow in the decompiled output is complex largely due to the fact that Java bytecode only allows binary comparison operations for deciding control flow. However, this restriction does not exist in Java where boolean expressions can be aggregated using the && and || operators. Previously, Dava did not make use of this ability and hence converted each comparison operation into a separate conditional construct. This results in the creation of unnecessary Java constructs and their complicated nesting further increases

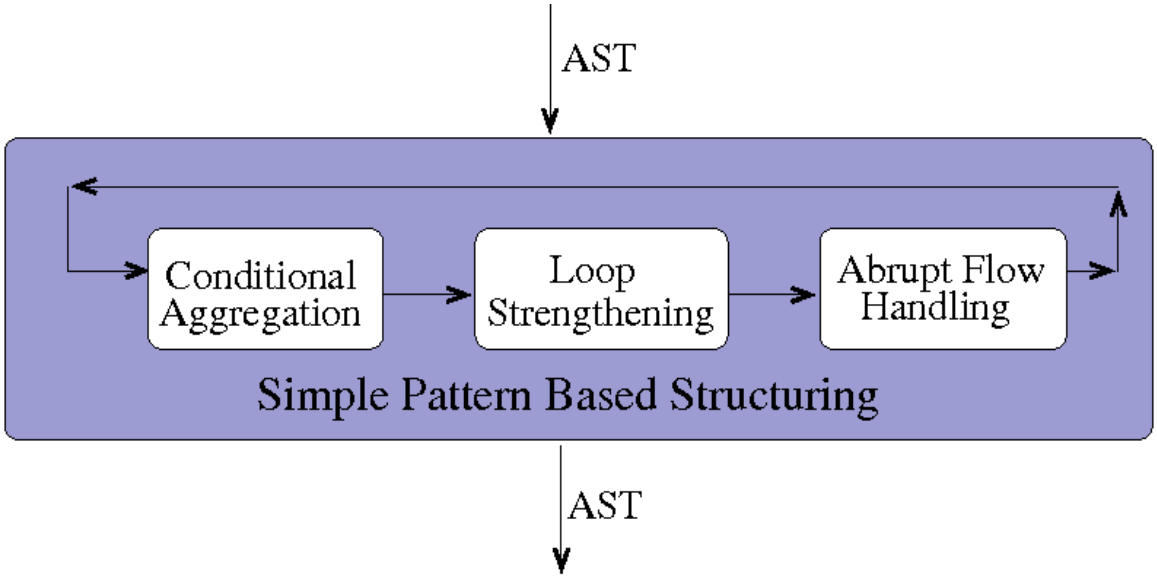


Figure 5.1: *Simple Pattern Based Structuring*

code complexity. For instance, an `If` statement evaluating two conditions using the && operator in the source code gets decompiled into two `If` statements, one completely nested within the other. Similarly, if a loop checks for multiple conditions in the source this gets transformed into a loop with one condition. The remaining conditions are checked within the loop body. By statically checking for such patterns, and merging the different conditions, the number of Java constructs can be reduced, thereby reducing the complexity of the output.

The reason that Dava, until now, did not use the ability to represent aggregated conditions in Java is that the `grimp` intermediate representation, which is the input to the decompiler, only contains binary comparison operators. The remaining parts of this section are divided as follows: we first enrich the `grimp` grammar by giving it the ability to represent both unary conditions and aggregated conditions along with the existing binary conditions (Section 5.1.1). Then in Section 5.1.2 we discuss the pattern which is used to aggregate two `If` statements by combining their conditions using &&. In Section 5.1.3 we discuss a number of patterns which combine `If` and `If-Else` statements using the || operator.

5.1.1 Grammar for aggregated boolean expressions

All types of AST nodes extending ASTControlFlowNode in Figure 2.5 contain a condition which decides the flow of control through the program. The old grammar (`grimp`) and the new enriched grammar, used by Dava, for the condition that can occur in the control flow nodes are presented in Figure 5.2. The old grammar is very restrictive in the sense that it only allows binary comparison operations. Unary conditions, that evaluate to `true` or `false` in Java, are not allowed by the grammar. To be able to represent such conditions using the old grammar a unary expression has to be compared with the BooleanConstants `true` or `false` (Section 4.1). This results in decompiled code that looks machine generated and is generally less readable.

Another issue with the old grammar was that expressions could not be aggregated with logical symbols && and ||. In the old grammar, an arbitrary boolean expression can be represented only by breaking the expression into multiple binary comparison control flow checks. This results in complicated control flow and causes the output of the decompiler to have many levels of nestings because of the use of many simple checks. To reduce the complexity of the control flow and at the same time improve the readability of the code, it is preferable to have relatively complicated checks, as is possible with the new grammar, but use only a few of them. Chapter 9 defines Conditional Complexity based on this enriched grammar. It is expected that as the conditional complexity increases, due to increased aggregation, the number of conditional constructs will decrease.

The important additions made to the grammar, as can be seen in the right side of Figure 5.2, are the addition of unary expressions (e.g. a boolean variable, a method returning a boolean etc), the introduction of && and || symbols and the composition of unary and binary conditions using these symbols. Note that the grammar presented in Figure 5.2 is an ambiguous grammar. The purpose of the grammar is to illustrate the different types of conditions that can occur, within the Dava AST. The `SootExpr` in the grammar is treated as a token in the grammar. Additionally, for the case of the `BoolSimpleExpr` all alternatives for this production are restricted to have boolean types.

The addition of the new grammar has been carried out in such a way that the previous analyses built in Dava still function as intended although without using the expressiveness

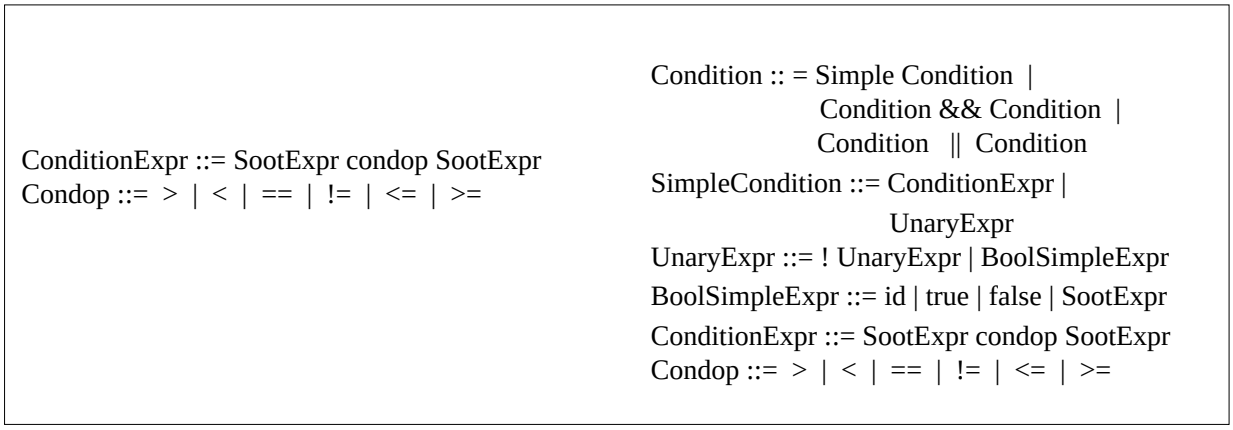


Figure 5.2: *Dava's AST Condition Grammar*

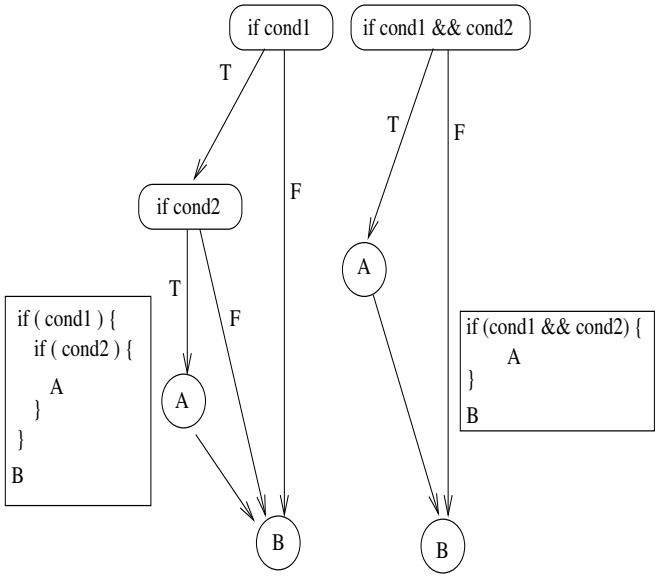
of the added grammar. New analyses introduced into Dava are implemented using the new grammar.

5.1.2 `And` Aggregation

`And` aggregation is used to aggregate two `If` statements into one using the && symbol. Figure 5.3(a) shows the control flow of two `If` conditions, one fully nested in the other. From the control flow graph it can be seen that A is executed only if both `cond1` and `cond2` evaluate to true. B is executed no matter what. In Figure 5.3(b) we see the reduced form of this graph where the two `If` statements have been merged into one by coalescing the conditions using the && operator. Statements 9 to 13 in Figure 1.2(e) match this pattern. The matched pattern and the transformed code are shown in Figure 5.4.

The pattern not only decreases the nesting level of constructs, by removing the inner nested `If` statement, but also shrinks the overall size of the code. By shrinking the size of the code using such an aggregation strategy the code becomes more readable and the control flow is easier to follow.

In order to apply this transformation it is important to ensure that the nested `If` statement should be the only construct within the parent `If` statement. More specifically, during a depth first traversal of the AST this pattern is matched if:



(a) Unreduced (b) Reduced

Figure 5.3: *Reducing using the* && *operator.*

(a) Original Code

```
1 if(i0 < 3){
2   if(i1 == 1){
3     break label_0;
4   }
5 }
```

(b) Transformed Code

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

Figure 5.4: *Application of* `And` *Aggregation*

- An `If` statement s_1 has a child s_2
- s_2 is an `If` statement
- s_2 is the only child of s_1

Algorithm 4 shows the algorithm used to detect the And Pattern and to transform the AST accordingly.

Algorithm 4: And Aggregation

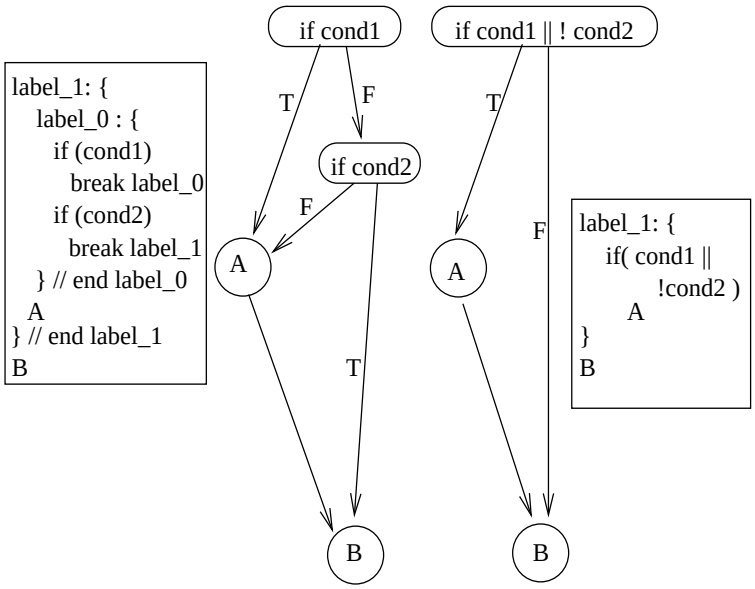
```
Input: ASTNode node
if node is an If Construct then
    B ← GetBody(node)
    if B has one ASTNode then
        v ← GetNode(B)
        if v is an If Construct then
            cond1 ← GetCondition(node)
            cond2 ← GetCondition(v)
            newCondition ← ASTAndCondition(cond1,cond2)
            newBody ← GetBody(v)
            newNode ← new ASTIfNode(newCondition,newBody)
            replace(node,newNode)
        end
    end
end
```

5.1.3 `Or` Aggregation

Figure 5.5 shows the control flow of the `Or` Operator. The unreduced version of the control flow shows that A is executed if `cond1` evaluates to true. If, however, the false branch is taken then `cond2` is evaluated and A is executed if this condition is false. B is executed no matter what. In short, A is executed if the first condition is true or the negated second condition is true, followed by the execution of B in all cases. This graph can therefore be

reduced to that in Figure 5.5(b) where the `If` statement aggregates the two conditions using the || operator.

One of the patterns to which the control flow graph in Figure 5.5(a) can map is shown in Figure 5.5. The pattern looks for a sequence of n `If` statements (n is 2 in Figure 5.5) with the first n-1 statements breaking to a particular label (`label_0` in Figure 5.5) and the nth statement targeting an outer label (`label_1` in Figure 5.5). During execution this results in the evaluation of a sequence of `If` conditions and as soon as any of the n-1 conditions evaluates to true or the nth condition evaluates to false a certain chunk of code (A in the figure) is targeted. If the program gets to the nth condition and this evaluates to true then in this case A is not executed. This code therefore corresponds to an `If` statement with A as its body and the condition the n-1 conditions and the negated nth condition combined using the || operator.



(a) Unreduced (b) Reduced

Figure 5.5: *Reducing using the || operator*

The decompiled code in Figure 1.2(e) (reproduced here in Figure 5.6(a) with the `And` aggregation applied) has one occurrence of this pattern. Statement 2 in Figure 5.6(a) is the

outer label and Statement 8 the inner one. There are two `If` statements in the sequence: statement 10 breaking the inner label and statement 13 targeting the outer one. The transformation removes the second `If` statement by moving its negated condition into the first statement. The new body of this statement consists of statement 16. The result of this transformation is shown in Figure 5.6(b).

(a) Original Code

```
1  label_2:{
2    label_1:
3    while(z0){
4      if (!z1){
5        break label_2;
6      }
7      else{
8        label_0:{
9          if(i0 < 3 && i1 == 1){
10           break label_0;
11         }
12         if(i1 + i0 >= 1){
13           continue label_1;
14         }
15       } //end label_0:
16       System.out.println(r1);
17     }
18   }
19 } //end label_2:
```

(b) Transformed Code

```
1  label_2:{
2    label_1:
3    while(z0){
4      if (!z1){
5         break label_2;
6      }
7      else{
8         if( (i0 < 3 && i1 == 1)
9                   || i1 + i0 < 1 ){
10           System.out.println(r1);
11        }
12     }
13   }
14 } //end label_2:
```

Figure 5.6: *Application of `Or` Aggregation*

This transformation can greatly reduce the size of the code and improve the readability as well. An interesting side-effect of the transformation is the removal of a `Labeled-Block` and `break` statements. The first n-1 statements all break `label_0` whereas the nth statement targets `label_1`. After the transformation all n-1 `break` statements have been removed

Algorithm 5: Or Aggregation

```
Input: ASTNode node
if node is a Labeled Block then
    foreach child nodeChild in node.GetBody() do
        if nodeChild is a Labeled Block then
            outerLabel ← GetLabel(node)
            innerLabel ← GetLabel(nodeChild)
            innerBody ← GetBody(nodeChild)
            if FindIfSequence(innerBody,outerLabel,innerLabel) then
                //Pattern Matched
                Create newCondition by aggregating the sequence of conditions
                using OR (last condition of the sequence is negated)
                foreach successor child sChild of node.GetBody() after nodeChild
                do
                    node.remove(sChild)
                    newBody.add(sChild)
                end
                newIfNode ← new ASTIfNode(newCondition,newBody)
                node.replace(nodeChild,newIfNode)
                break
            end
        end
    end
end
```

which also allows the removal of `label_0`. Also, although we cannot directly remove `label_1`, without checking that the `If` body does not target it, we have reduced the number of abrupt edges targeting it by one. In Section 5.3.3 we discuss an algorithm that checks for spurious labels and subsequently removes them.

The algorithm for the transformation is shown in Algorithm 5. If at any stage of the traversal of the tree we find a labeled block (`node` in Algorithm 5) then the body of this

block is searched for an inner labeled block. If one is found then the `FindIfSequence` function is invoked which checks that there is a sequence of `If` statements adhering to the pattern we are looking for. If the pattern is matched then first the `newCondition` is created. The body of the new `If` statement (`newBody` in Algorithm 5) is the sequence of all nodes within the outer labeled block which follow after the inner labeled block. Hence these nodes are removed from the outer block's body and used to create the body of the new `If` statement. Once done the new `If` statement replaces the inner labeled block.

Function: FindIfSequence

Input: List *body*, String *outerLabel*, String *innerLabel*
Output: boolean *FoundOrNot*
foreach *ASTNode node in body* **do**
    if *node is not an if construct* **then**
        return *false*
    ifBody ← `GetBody(node)`
    if *ifBody is not an abrupt statement* **then**
        return false;
    abruptStmt ← `GetStmt(`*ifBody*`)`
    if *node is the last node* && *abruptStmt targets outerLabel* **then**
        return true;
    else if *node is not the last node* && *abruptStmt targets innerLabel* **then**
        continue
    else
        return *false*
    end
end

Other `Or` Aggregation Patterns

We discuss some other patterns in this section which can map to an aggregation of conditions using the `Or` operator. In Figure 5.7, code A is executed if `cond1` evaluates to `true`.

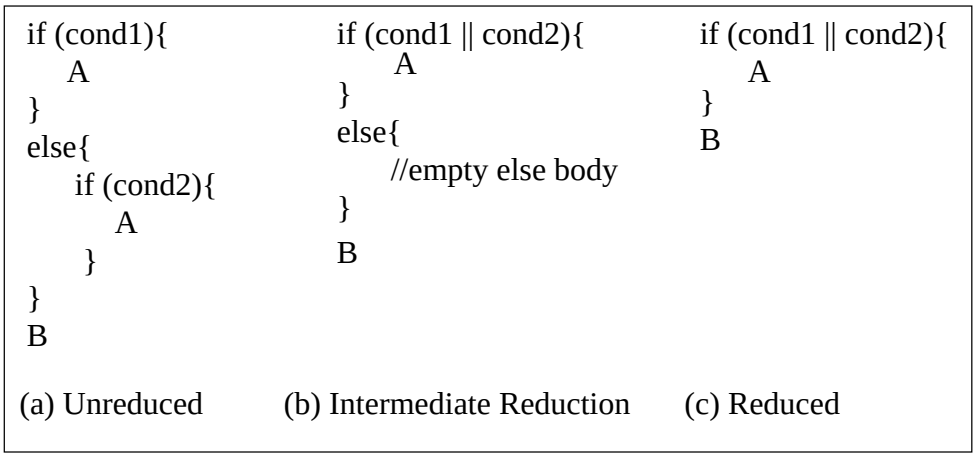


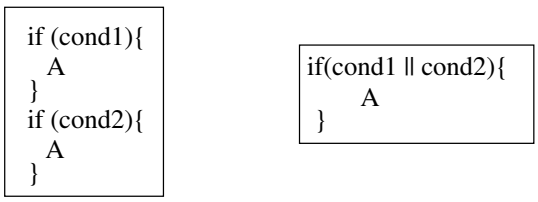
Figure 5.7: *Removing Nested `If` statements using the || operator*

If `cond1` is `false` then the second condition, `cond2` is evaluated with the `true` branch resulting in the execution of A. B is executed no matter what. The code therefore executes A if either `cond1` OR `cond2` evaluates to true. We can hence reduce the pattern by creating a new `If` statement which has the condition the result of aggregating `cond1` and `cond2` using ||. The transformation is implemented in two stages. The first stage involves removing the `If` statement in the `else` body of the `If-Else` construct and adding `cond2` into the condition of the `If-Else` statement. The removal of the `If` statement leaves the `else` body empty. The second stage of this transformation then takes the `If-Else` statement and converts it into an `If` statement.

Figure 5.8 shows another `Or` aggregation pattern. Figure 5.8(a) shows two `If` statements with the same body (in the general case the pattern works for a sequence of `If` statements with the same body). The pattern can be reduced to the one shown in Figure 5.8(b) where the two conditions of the `If` statements have been merged using ||. However, this transformation is only possible if the body common to the `If` statements (A in Figure 5.8) ends with an abrupt statement. The reason for this can be seen by inspecting the execution sequence of the code in Figure 5.8(a) in both cases, when the common body has an abrupt edge and when it does not.

- **BodyA has an abrupt edge**:
  Abrupt edges include breaks, continues and return statements. The code starts executing by checking `cond1`. If `cond1` evaluates to true then BodyA is executed. Since BodyA contains an abrupt edge the execution moves to another place in the code and the second `If` statement is not executed. If, however, `cond1` evaluates to false the second `If` statement is checked and BodyA is executed if cond2 evaluates to true. The important thing to note is that BodyA gets executed if `cond1` evaluates to true or if that doesn't then `cond2` does. Also because of the abrupt edges in BodyA, BodyA only gets executed once. In this case we can combine the `cond1` and `cond2` using the `Or` operator into one `If` statement with the body as BodyA.

- **BodyA has no abrupt edge**:
  In this case the code starts out by checking the condition of the first `If` statement. If this evaluates to true then Body A is executed. Since BodyA does `not` have an abrupt edge then the second `If` statement is executed. If this condition, `cond2` also evaluates to true BodyA is executed `again`. So in the case where BodyA does not have an abrupt edge, BodyA has a chance of running twice (in our example) and multiple times in the case of the more general pattern. Looking at this sequence of execution it should be clear that in this case one cannot aggregate the two `If` statements since that would change the semantics of the program.



(a) Unreduced (b) Reduced

The pattern is only applicable if Body Ais an abrupt edge (return/break/continue).

Figure 5.8: *Removing similar `If` statements using the || operator.*

Another very important thing to keep in mind is that the order of the conditions in

the aggregated Or Condition is important. The reason being that the evaluation of these conditions can have side effects. In the unreduced pattern, if `cond1` evaluates to true then the program will never evaluate `cond2`. Hence we need the same semantics for our reduced pattern. This is achieved by having `cond2` to the right of `cond1` in the aggregated condition. This ensusures that if `cond1` evaluates to true `cond2` will not be evaluated and we adhere to the semantics of the original program. The pattern 3 transformation is implemented using algorithm 6.

Algorithm 6: Or Aggregation for similar bodies

```
Input: ASTNode node
body ← GetBody(node)
Iterator it ← body.iterator()
while it.hasNext () do
    node1 ← it.Next()
    if ! it.hasNext () then
        return;
    node2 ← it.Next()
    if node1 and node2 are texttIf statements then
        body1 ← GetBody(node1)
        body2 ← GetBody(node2)
        if body1 and body2 are the same then
            if body1 has an abrupt Edge then
                leftCond ← GetCondition (node1)
                rightCond ← GetCondition (node2)
                newCondition ← ASTOrCondition (leftCond,rightCond)
                newIfNode ← ASTIfNode(body1,newCondition)
                body.remove(node1)
                body.replace(node2,newIfNode)
            end
        end
    end
end
```

5.2 Loop strengthening

Previously, in the case where loops have multiple conditions, Dava used one of these conditions as the loop condition and the remaining ones were added as `If` or `If-Else` statements inside the loop body. Hence, similar to `If` and `If-Else` statements, loops can now hold aggregated conditions to be evaluated before execution of the loop body. Therefore pattern matching can be used to strengthen the conditions within a loop. In the next two sections we discuss how `If` and `If-Else` statements nested within loops can be used to strengthen the conditions of loops and at the same time remove abrupt statements and shrink the code base.

5.2.1 Using a nested `If-Else` Statement to Strengthen Loop Nodes

The decompiler uses `If-Else` statements if the loop body is non-empty. The `If` body is the non empty body of the original loop and the `else` body contains abrupt control flow out of the loop. Two different types of patterns can arise as discussed below.

Figure 5.9(a) shows a while loop with an `If-Else` statement as its only child. Reasoning about the control flow shows that Body A is executed if both `cond1` and `cond2` evaluate to true. If either of the conditions are `false`, the loop exits. This fits in with the notion of a conditional loop with two conditions as seen in the reduced form of the code in Figure 5.9(b). Notice that the label on the `While` loop is still present in the reduced code. This is because there can be an abrupt edge in Body A targeting this label. After the reduction the algorithm in Section 5.3.3 is invoked to remove the label from the loop, if possible. Notice that if the bodies in the `If-Else` statement are reversed: the `If` branch contains the `break` out of the loop and the `else` branch contains a body similar to the BodyA mentioned above. In this case by adding the negated condition of the `If-Else` statement the same transformation can be applied.

Figure 5.10 shows a similar strengthening pattern for unconditional loops. The only difference is that in this case the `If-Else` statement is free to have any construct in both branches as long as one of the branches has an abrupt edge targeting the labeled loop. The reduction works by converting the `Unconditional-While` loop to a conditional loop with

(a) Unreduced conditional loops

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

(b) Reduced conditional loops

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

Figure 5.9: *Strengthening Loops*

Body A as the body of the loop. Body B is then moved outside the loop. The specialized pattern where Body B is empty makes this pattern the same as the pattern for `While` loops.

Looking at our working example (Figure 5.6(b)) where `And` and `Or` aggregation have already been applied, reproduced as Figure 5.11(a), we can see that statements 3 to 13 make a `While` loop which has one `If-Else` statement. Notice that in this case the `If-Else` statement is reversed: the `If` branch contains the break out of the loop and the `else` branch contains Body A (statements 8-10). In this case we can apply the `While` strengthening pattern by adding the negated condition of the `If-Else` statement into the `While` condition. The transformed code is shown in Figure 5.11(b).

5.2.2 Using a nested If Statement to Strengthen loop Nodes

Pattern matching on loops containing `If` statements results in loops with empty bodies with the work being done from within the conditions of the loop. Such kind of loops are often encountered in concurrent programs e.g. busy waiting.

The pattern shown in Figure 5.12 shows the transformation of a conditional while loop to a loop in which the strength of the loop condition has been increased by the addition of

(a) Unreduced unconditional loops

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

(b) Reduced unconditional loops

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

Figure 5.10: *Strengthening Unconditional Loops*

(a) Original Code

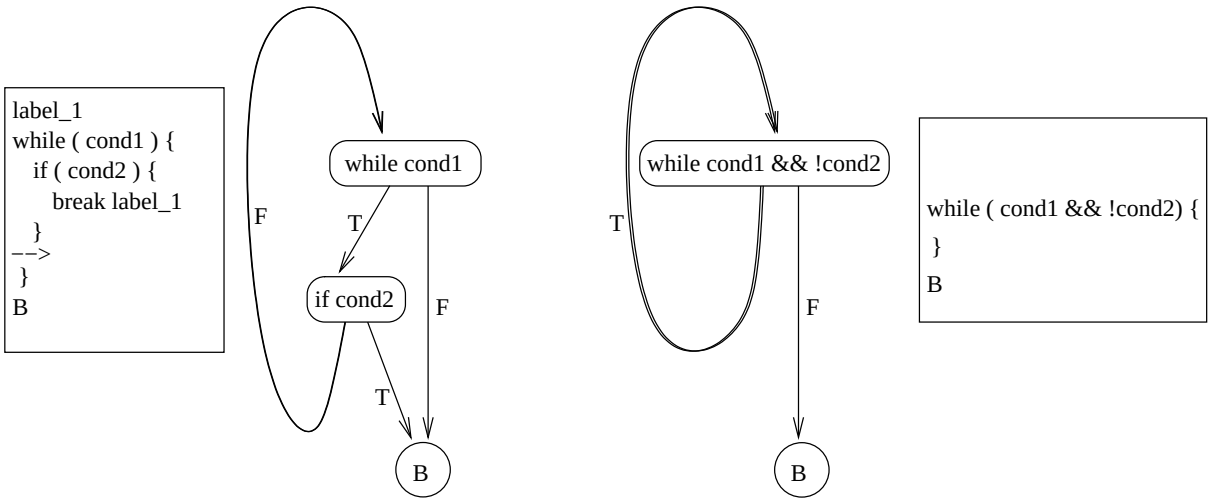
```
label_2:{
  label_1:
  while(z0){
    if (!z1){
       break label_2;
    }
    else{
       if( (i0 < 3 && i1 == 1)
                  || i1 + i0 < 1 ){
           System.out.println(r1);
       }
    }
  }
} //end label_2:
```

(b) Transformed Code

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

Figure 5.11: *Application of While Strengthening*

`cond2`. The reasoning for this is that the execution of the code stays within the while loop as long as `cond1` evaluates to true and `cond2` evaluates to false. If either `cond1` evaluates to false or `cond2` evaluates to true the while loop is broken and Body B is executed. Therefore the pattern in Figure 5.12(a) can be reduced to that in Figure 5.12(b). Note that the transformation is possible only if the while loop contains a single `If` statement in its body. Specifically the point marked with an arrow in Figure 5.12 should not have any AST Node. Algorithm 7 shows how the reduction can be implemented.



(a) Unreduced

(b) Reduced

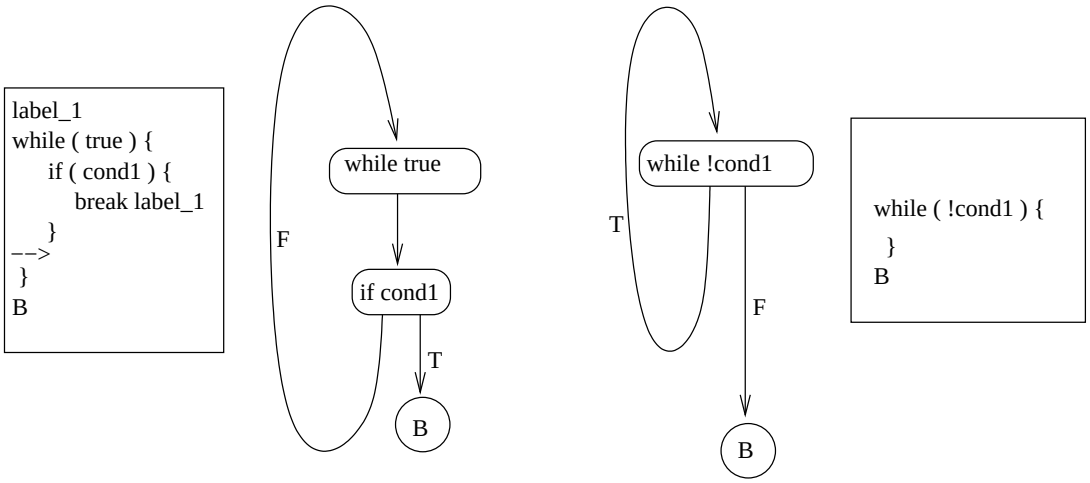
Figure 5.12: *Strengthening a While Loop Using an `If` statement*

Algorithm 7: Strengthening While Loops Using If statements

```
Input: ASTWhileNode node
label ← GetLabel (node)
body ← GetBody (node)
if the only child, onlyChild in body is an If statement then
    B ← GetBody(onlyChild)
    if B has one statement only then
        stmt ← GetStatement(B)
        if stmt is a break stmt then
            if label is the same as GetLabel(stmt) then
                cond1 ← GetCondition(node)
                cond2 ← GetCondition(onlyChild)
                cond2 ← FlipCondition(cond2)
                newCondition ← ASTAndCondition(cond1,cond2)
                newBody ← EmptyBody()
                newNode ← new ASTWhileNode(newCondition,newBody)
                replace(node,newNode)
            end
        end
    end
end
```

Figure 5.13 shows the counterpart of the previous pattern for unconditional loops. From Figure 5.13(a) it can be seen that the only way the loop terminates is if `cond1` evaluates to true. This can therefore be represented as a conditional loop with the negated `cond1` as the condition. Again it is important to notice that the transformation is possible only if the unconditional loop has the `If` statement as the only child. After the transformation the loop, which is now a conditional loop, will terminate only if the condition evaluates to false. Since the condition is the negated `cond1` the semantics of the code are maintained. The algorithm for this transformation is similar to Algorithm 7. The only differences being that the new while node contains the `If` statement's condition and that the new while node

replaces the old unconditional loop node.

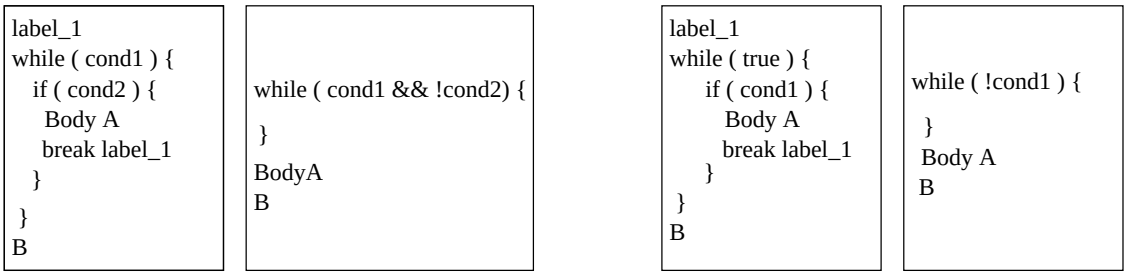


(a) Unreduced (b) Reduced

Figure 5.13: *Strengthening an Unconditional Loop Using an `If` statement*

The pattern above can be generalized to include the case when the `If` statement does not only contain the abrupt statement. The reason this restriction was imposed for conditional loops can be seen from Figure 5.14(a). The `If` statement contains a body (BodyA) followed by the `break` statement. If we were to apply the reduction we would get code shown on the right side of Figure 5.14(a). However, this code has different semantics from the original code. This can be seen by checking when BodyA gets executed. In the undreduced version BodyA gets executed only if `cond1` and `cond2` are both true. However in the reduced version BodyA can get executed if `cond1` is false.

In the case of unconditional loops it is noted that such a restriction is not needed. This can be seen in Figure 5.14(b). The reason for this being that no condition is checked in the unconditional loop and hence the control flow decision is made solely from within the loop body. As can be seen from the unreduced and reduced versions of this pattern BodyA gets executed only if `cond1` evaluates to true and at the same time results in the control exiting the loop.



(a) An Incorrect Transformation

(b) A Correct Transformation

Figure 5.14: *Strengthening an Unconditional Loop Using an `If` statement*

5.3 Handling Abrupt Control Flow

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

5.3.1 `If-Else` Splitting

The restructuring of the bytecode often results in the creation of `If-Else` statements where `If` statements would have sufficed, because of the `goto` statements linking the different chunks of bytecode together. An example of this is shown in Figure 5.15(a). The proposed transformation is shown in Figure 5.15(b). Notice that BodyB which was in the **else** branch of the `If-Else` statement has been removed out of the conditional statement. This is possible because of the abrupt edge at the end of the **then** branch of the `If-Else` statement. The abrupt statement indicates that control is going to flow to some other location of code. If we can confirm that the abrupt statement does not target a label on this `If-Else` statement

then we know that BodyB will not be executed even if it is outside the `If` statement. One additional requirement is that if the `If-Else` statement has a label on it then BodyB should never target this label since once removed from the **else** branch it is no longer under the scope of the label (which will now be on the `If` statement).

(a) Unreduced

```
if(cond1){
   BodyA;
   <abrupt edge>
}
else{
   BodyB;
}
```

(b) Reduced

```
if(cond1){
   BodyA
   <abrupt edge>
}
BodyB
```

Figure 5.15: *`If-Else` Splitting*

If this pattern does not get matched we also try the reverse of the pattern *i.e.*, where the **else** branch has a body followed by an abrupt statement and the **then** branch is some body which does not target any label on the `If-Else` statement. In this case, the new `If` statement contains the **else** branch as its body and the condition of the statement is the negated condition of the original `If-Else` statement. Figure 5.16 shows code from a real decompilation scenario where the reversed `If-Else` pattern gets matched. The `If-Else` statement in Figure 5.16(a) contains a `return` statement in the **else** branch. In Figure 5.16(b), the transformation is able to create a `If` statement with the abrupt edge as part of the body, by negating the original `If-Else` condition.

5.3.2 Useless `break` statement Remover

Another artifact of Java bytecode is the occurrence of unneeded `break` statements. Java constructs have predefined fall through semantics *i.e.*, after execution of a certain construct control moves to the next statement in the code. Using this knowledge it is sometimes

(a) Unreduced

```
if(i3 == 0)
{
   i0++;
}
else
{
   a.remove(i0);
   return;
}
```

(b) Reduced

```
if(i3 != 0){
   a.remove(i0);
   return;
}
i0++;
```

Figure 5.16: *`If-Else` Splitting*

possible to remove `break` statements which target the same code location that is the natural fall through of the labeled construct. Two examples of this are shown in Figure 5.17.

The algorithm works by looking for `break` statements in the code. Whenever a `break` statement is found, the transformation finds the target node of the `break` statement. Then each of the ancestors of the `break` statement up to the target node are analyzed. The `break` statement is unneeded if it is the last statement in its parent node, the parent node is the last node of its parent and so on until we reach the target node. For instance, in the left side of Figure 5.17 the `break` statement is unneeded since it is the last statement in the `If` statement which is itself the last node within the **then** branch of the `If-Else` branch. Hence the natural fall through, BodyD, is the same as that targeted by the `break` statement. The `break` statement can be safely removed. On the right side of Figure 5.17 again we see an unneeded break. The `break label8` statement targets BodyC which is the natural flow through after execution of BodyB. Hence this `break` statement can also be removed.

One important thing to remember is that `break` statements are also used to break out of a loop. Hence the transformation can only be applied if none of the ancestors of the `break` statement up to the targeted node is a loop construct.

If a `break` statement is found to be unneeded then an added advantage of this can be

```
label1:
if(cond1){
  BodyA
  if (cond2){
     BodyB
     break label1
  }
}
else{
  BodyC
}
BodyD
```

```
label8:
try {
   BodyA
}
catch (Exception e){
   BodyB
   break label8;
}
BodyC
```

Figure 5.17: *Removing useless* `break` *statements*

that the label might also become removable, as discussed in the next section.

5.3.3 Useless Label Remover

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

Labels can occur in Java code in two forms: as labels on Java constructs e.g. `While` loop or as labeled blocks. If a label is shown to be spurious, by showing that there is no abrupt edge targeting it, then in the case of a labeled construct the label is simply omitted. However, in the case of a labeled block, a transformation is required which removes the labeled block from the AST. Algorithm 8 shows how a spurious labeled block is removed by replacing it with its body in the parent node.

When applied to the code in Figure 5.11(b) `label_2` and `label_1` which were at statements 1 and 2 are both removed. Looking back at the original source code from which this decompiled output was generated (reproduced as Figure 5.18(a)) we see that, after

Algorithm 8: Removing Spurious Labeled Blocks

Input: ASTNode *node*

body ← `GetBody(`*node*`)`
Iterator *it* ← body.`iterator()`
while it.`hasNext()` **do**
  node1 ← it.`Next()`
  if *node1 is a Labeled Block Node* **then**
    if `IsUselessLabelBlock` *(node1)* **then**
      body1 ← `GetBody(`*node1*`)`
      Replace *node1* in *body* by *body1*
    end
  end
end

applying the AST rewriting, Dava's output, Figure 5.18(b), matches the original source code.

(a) Original Code

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

(b) Final Dava Output

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

Figure 5.18: *Comparing Dava output*

5.3.4 Reducing the scope of labeled blocks

While pattern matching labeled blocks to rewrite the AST, some pattern might not get matched because the labeled block contains too many children in its body. It is sometimes possible to reduce the scope of the labeled block. One such possibility can be seen in Figure 5.19(a). The unreduced code shows that `label_1`, which is a labeled block, consists of some code that does not use the label followed by code which targets this label (the `While` loop in Figure 5.19(a)). Since the initial code does not involve the use of `label_1` there is no reason why this code cannot occur outside the scope of the labeled block. As seen in the reduced form of the code (Figure 5.19(b)) the labeled block has been removed by placing the label directly on the `While` loop construct. Such a transformation is possible

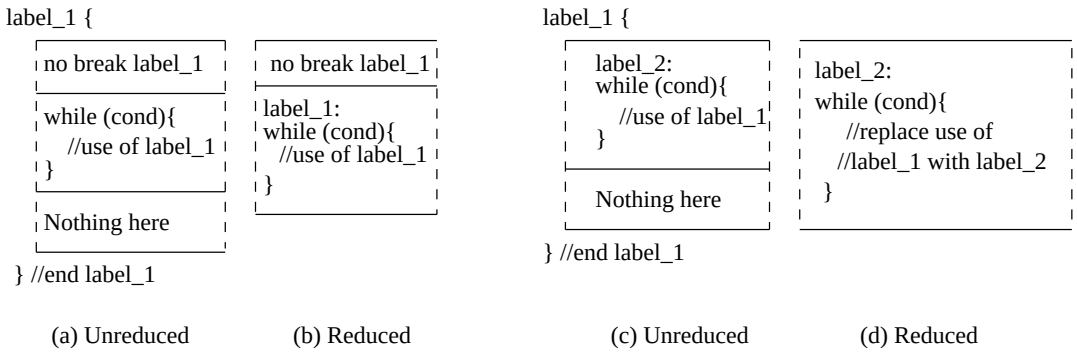


Figure 5.19: *Reducing the scope of Labeled Blocks*

if the following conditions hold:

- The construct that holds the abrupt statement targeting the labeled block should itself be able to hold a label. These include all the AST nodes derived from the AST Labeled Node from the type hierarchy in Figure 2.5.
- The construct that targets the label should be the last child of the labeled block node. The reason for this restriction is illustrated in Figure 5.20. In Figure 5.20(a) body A is a child of the labeled block occurring after the `While` loop which targets the labeled block. If, according to the transformation, we were to remove the labeled

block by placing the label onto the `While` loop (as shown in Figure 5.20(b)) then BodyA is no longer under the scope of the label. Hence the execution of the `break` statement breaks out of the loop but ends up executing BodyA which should not have been executed.

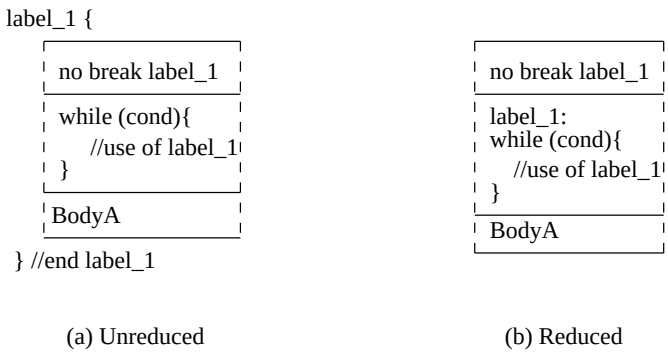


Figure 5.20: *Wrong Reduction of Scope*

- The construct that targets the labeled block should not already have a label on it. If such a situation arises, as shown in Figure 5.19(c), then the transformation can still be successful. However, in this case the construct's label is kept and any abrupt edge targeting the labeled block is made to target the label on the construct. In Figure 5.19(d) this means keeping the label `label2` on the `While` loop and removing the labeled block. Any abrupt statement targeting `label_1` is transformed to target `label2`. Obviously this is only possible if the labeled block had only one child otherwise the transformation changes the semantics for reasons similar to those discussed above.

The reasoning behind trying to reduce the scopes of labels is that if there are fewer children in a labeled block, then there are better chances that some other pattern will match. If no pattern matches, reducing the labeled block size still has the advantage of improving code complexity since the programmer now has to concentrate on a smaller chunk of code to understand the abrupt control flow targeting the labeled block.

Chapter 6
A Structure-Based Flow Analysis Framework

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

An example of this can be seen in Dava's output, Figure 1.3(d), for the obfuscated bytecode produced for the original Java source shown in Figure 1.3(a). Although semantically equivalent to the original code the output is hard to understand. However, since obfuscators have to ensure that their modifications do not change program semantics, a transformation of the output, making it similar to the original code, may be possible. This requires an answer to the questions: "What is the value of a particular variable at a program point?", "Is a particular piece of code ever executed?" and so on. To answer such questions one needs added information about the data and control flow which cannot be obtained from pattern matching and requires data flow analysis. We discuss more about decompiling obfuscated code in Section 7.3.7.

Although SOOT provides a flow analysis framework for each of the intermediate representations *i.e.*, `baf`, `jimple` and `grimp`, this support did not extend to the higher level intermediate representation of the decompiled code. Previously it was not possible to apply any flow analyses on Dava's AST. To perform more sophisticated transformations we implemented an analysis framework that can be used to implement static data flow analyses on Dava's AST. The analyses' results can then be leveraged to perform further transforma-

tion on the AST. The framework removes the burden of correctly traversing the AST from the analysis writer and allows him/her to concentrate on the analysis. With a framework in hand, the process of writing analyses for Dava has been streamlined making it easier for new developers to extend the system.

As the analyses for the decompiler are performed on the AST it is best to use a syntax-directed method of data flow analysis such as structural analysis[HDE+93, Sha80]. The advantage of using this technique is that it gives, for each type of high-level control-flow construct in the language, a set of formulas that perform data flow analysis. For instance it allows the analysis of a `While` loop by analyzing only its components: the conditional expression and the body. For this reason we find that structural flow analysis provides a more efficient and intuitive implementation of analysis on the tree representation than graph-based approaches. Apart from supporting ordinary compositional constructs such as conditionals and loops, the structural flow analysis also supports `break` and `continue` statements (Section 6.2).

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

The analysis begins by traversing the AST. As each Java construct is encountered a specialized method responsible for processing this construct is invoked. An `input` set containing information gathered so far is sent as an argument. Each construct is handled differently depending on the components it contains and its semantics. The processing of the construct might add, remove or modify elements of the `input` set. The result is returned in the form of an `output` set which then becomes the `input` set for the next construct.

This kind of structure-based flow analysis is not new. Similar work has been done by Emami et. al. [HDE+93, Ema93] for gathering alias and points-to-analysis information for the McCAT C compiler. Dava's flow analysis framework is an implementation of the same approach utilized in McCAT, but implemented for Java.

6.1 Merge Operations

An important construct in flow analyses is the merge operation. Merge defines the semantics of combining the information present in two `flow-sets`. Such a situation arises for instance when dealing with the `flow-sets` obtained by processing the `If` and `else` branch of an `If-Else` construct. Since the framework gathers sets of information the programmer has the choice of choosing between union and intersection as the merge operation. Customized merge operations might sometimes be needed for analyses. The framework allows the extension of the already implemented merge operations or the implementation of new merge operations. Section 7.3.1 shows such an extension of the intersection merge operation for the constant propagation analysis.

6.2 Dealing with Abrupt-Control Flow Constructs

In `grimp`, control flow is represented using explicit goto statements. The Structured Encapsulation Algorithms implemented in Dava are able to transform most of these goto statements, along with appropriate code bodies, into Java constructs like `If`, `While` *etc*. However, after all construct detection algorithms have been applied some `goto` statements might still be present in the AST. These remaining `gotos` are converted into `break` and `continue` statements and embedded into the AST.

We handle these statements as follows: whenever an abrupt control flow statement is encountered, the flow set containing information gathered by the analysis is stored. Processing then continues with a special `flow-set` named `BOTTOM` sent onwards indicating that this path is never realized (as the abrupt statement leads execution to some other area of the code). We use a hash table, keyed by labels, to store the `flow-sets` for unrealized paths. When a labeled construct is being processed all `break-sets`, or `continue-sets`, stored when encountering a `break`, or `continue`, targeting this label are retrieved. These are then merged with each other to get one out-set which is the conservative approximation summarizing the data flow sets from all abrupt statements targeting this particular construct. This `flow-set` is then merged with the `flow-set` obtained through analysis of the construct if no abrupt statement was encountered. The merging of the abrupt flow-sets is done

by the methods `handleBreak` and `handleContinue` for `break` and `continue` statements respectively.

In order to be complete in handling all abrupt statements one also needs to handle `return` and `throw` statements. The framework, on encountering one of these statements, outputs `BOTTOM`. Any other analysis-specific information to be gathered from the encountered abrupt statement can be obtained by over-riding appropriate methods provided by the framework.

6.3 Construct specific processing

Structure-based flow analysis derives its power from the fact that each high-level control-flow construct can be processed separately according to the semantics defined by the language. In this section, we discuss the handling of Java constructs present in the AST. Processing of each construct is presented with a control flow diagram showing the required semantics of the construct along with pseudo-code illustrating how the `flow-sets` are carried through the construct. Handling of `break` and `continue` statements is carried out as part of the processing and considerably complicates matters. The key to all these algorithms is the right order of merging the sets flowing through the constructs.

Java `Method` Node

A method construct is the simplest construct to deal with. The in-set is passed to the algorithm processing the body of the method. The output of processing this body becomes the out-set of the method construct (Figure 6.1(a)). This corresponds to the use of the flow-analysis framework for intra-procedural analyses. In the future if inter-procedural analyses are to be accommodated then the output set of processing the body would contain the output of regular execution of the method code merged with all possible exits of the method: `return` statements within the method's body and any `throw` statements that might escape the method.

Java `Labeled-Block` Nodes

Labeled blocks are often used in Java to separate different parts of an algorithm. Normal

```
process_Method(
        ASTMethodNode node,
        Object input){
  out1 = processBody(node,input)
  return out
}
```

(a) Java Methods

```
process_LabeledBlock(
       ASTLabeledBlockNode node,
       Object input){
  out1 = processBody(node,input)
  result = handleBreaks(out1,node)
  return result
}
```

(b) Java Labeled Blocks

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

(c) Java Statement Blocks

```
process_SynchBlock(
      ASTSynchronizedNode node,
      Object input){
  out1 = processSynchedLocal(
           local,input)
  out2 = processBody(node,out1)
  result = handleBreaks(out2,node)
  return result
}
```

(d) Java Synchronized Block

Figure 6.1: *Structural Flow-Analysis Algorithm for Simple Java Constructs*

code execution flows by entering the start of a labeled block and exiting at the end. However, `break` statements can be used to target the end of the labeled block from anywhere within the body of the block code. Taking that into account the processing of the labeled block is shown in Figure 6.1(b). If no `break` statement targets this block then the out-set of the block is the output of the processing the body of the block. However, to handle any break statements the output of normal execution of the block's code needs to be merged with all possible `flow-sets` stored when encountering a `break` statement targeting this

labeled block. This is done by statement 5 in Figure 6.1(b).

`Statement-Sequence` Construct

Figure 6.1(c) shows how the framework handles a sequence of statements. The processing method iterates through the statements in the sequence with the `output` set of one statement becoming the `input` of the next statement. The `output` set of the last statement is the `output` set of the sequence of statements.

One interesting thing to note is that it is while processing a `Statement-Sequence` that one may encounter abrupt statements. As mentioned in Section 6.2 when such an abrupt statement is encountered then the current flow-set is stored in the appropriate `breakList` or `continueList`. The out-set sent forward is `BOTTOM` indicating that this path is never taken. Hence the output set of a `Statement-Sequence` containing an abrupt statement is always `BOTTOM`.

`Synchronized` Construct

A synchronized block contains two components to be analyzed. First is the object on which the synchronization is carried out. The output of processing the synchronized object becomes the input of processing the synchronized body. Since synchronized blocks can have labels on them the final output is the result of merging the output of the synchronized body with any flow-sets stored in the `breakList`.

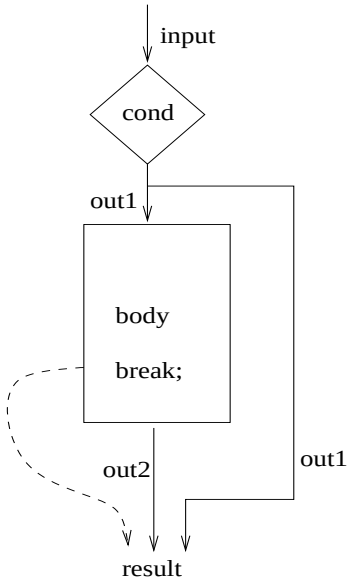
`If` Construct

Figure 6.2 shows the processing of `If` statements. Figure 6.2(b) shows possible control flow through an `If` statement. When an `If` statement is encountered the condition is evaluated. If the condition evaluates to `true` the `If` body is executed, otherwise control moves forward, skipping the `If` body. Keeping these semantics in mind the flow analysis processes an `If` statement (Figure 6.2(a)) by first processing the condition. This output (out1 in Figure 6.2(a)) becomes the input to process the `If` body. Since the `If` body might or might not be executed the output of the `If` statement is the merge of the out-set of just evaluating the condition (out1) with the out-set of processing the `If` body (out2). Once this merge is available any `break` sets that might have been targeting this `If` statement are

handled. That produces the final result of processing the `If` statement.

```
1 process_if(ASTIfNode node,Object input){
2   out1 = processCondition(condition,input)
3   out2 = processBody(node,out1)
4
5   //merge cond evaluating to false
6   out = merge(out1,out2)
7
8   result = handleBreaks(out,node)
9   return result
10 }
```

(a) Pseudo-code



(b) Graphical Representation

Figure 6.2: *The Structural Flow-Analysis Algorithm of If Construct.*

`If-Else` Construct

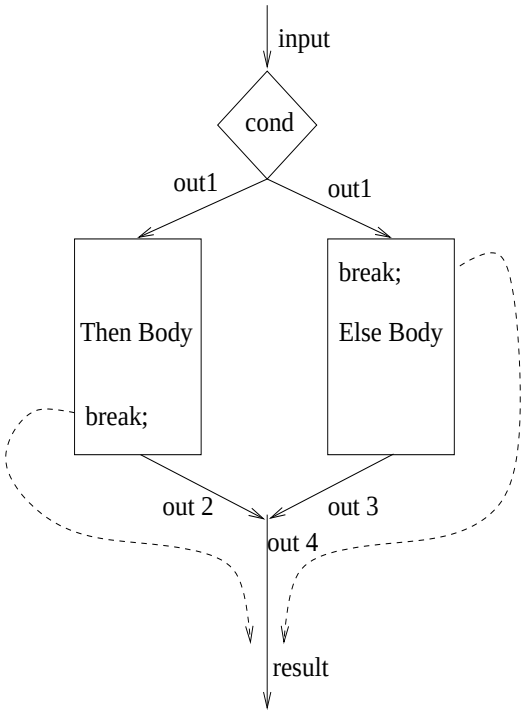
The semantics of an `If-Else` statement are almost the same as that of an `If` statement. Execution begins with the evaluation of the condition. If the condition evaluates to `true` then the `If` branch (also called the `then` branch) is taken. In case the condition evaluated to `false` then the `else` branch is taken. The processing of this construct begins with the processing of the condition. The out-set from the processed condition is cloned because depending on the evaluation of the condition the same flow-set will be carried into the `then` or `else` branch. The outputs of processing the two branch bodies (out2 and out3 in Figure 6.3(b) are then merged since statically we can not predict which branch is being taken. The only remaining thing to do is to handle any breaks that might have targeted the `If-Else` construct if it has a label on it. This is done by the `handlebreaks` method in Statement 11. The output of this becomes the result of processing the `If-Else` construct.

```
process_ifElse(ASTIfElseNode node,
                   Object input){
  out1 = processCondition(condition,input)
  clonedInput = clone(out1)
  out2 = processBody(thenBody,clonedInput)

  clonedInput = clone(out1)
  out3 = processBody(elseBody,clonedInput)

  out4 = merge(out2,out3)
  result = handleBreaks(out4,node)
  return result
}
```

(a) Pseudo-code



(b) Graphical Representation

Figure 6.3: *The Structural Flow-Analysis Algorithm of IfElse Construct.*

`While` Construct

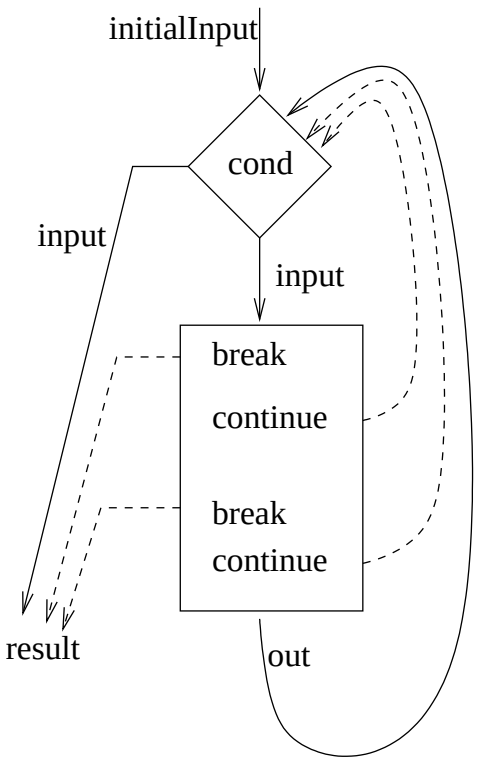
Processing loops complicates matters because of a fixed point iteration required to compute the out-set. Also with loops not only do we have to deal with `break` statements but also `continue` statements that could be targeting the loop. The semantics of the `While` loop dictate that processing starts with the evaluation of the condition. If the condition is `true` the body executes and then the condition is re-evaluated. Hence regular output *i.e.*, output without any `break` statements, from the `While` loop always ends with the evaluation of the condition. The `continue` statements stop the execution of the body at whatever place the `continue` statement is encountered and control goes back to the evaluation of the `While` condition.

Figure 6.4 shows the control flow and pseudo-code for handling a `While` loop. The solid back-edge indicates loop iteration and dotted lines indicate abrupt control flow. Firstly the

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

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

(a) Pseudo-code



(b) Graphical Representation

Figure 6.4: *The Structural Flow-Analysis Algorithm of While Construct.*

analysis processes the condition of the `While` construct. The `output` set of this becomes the `input` set for the fixed point computation. Within the fixed point computation the body of the `While` loop is processed followed by the generation of the `input` set for the next iteration.

The `input` set for the next iteration is generated by merging the `output` set of the current iteration with the `flow-sets` stored in the `continue` hash table, since `continue` statements could be targeting the loop.

Taking care of all possible entry points is essential for the correct working of the flow analysis. Since it is quite possible that the condition of the `While` loop evaluates to `false`

without any iteration of the loop it is important that the initialInput to the `While` loop be part of the input set to any re-evaluation of the condition. Hence the result of merging the output of any possible iterations (solid back edge labeled out in Figure 6.4(b)) with any `flow-sets` from the `continueList` (dotted back edges in Figure 6.4(b)) has to be further merged with the initialInput to the `While` loop. The result of this is the correct input to any further evaluations of the condition. Once the fixed point is achieved then any `flow-sets` stored in the `break` hash table are also merged using the `handleBreaks` method. The output of this method is the final output of processing the `While` construct.

`Do-While` Construct

The only difference between a `While` loop and a `Do-While` loop is that in a `Do-While` loop the loop body has to be executed at least once. The analysis starts off with first processing the body of the `Do-While` loop. Then any `flow-sets` stored in the `continueList` are merged to produce the in-set for the condition. Once the condition is processed the input set for further iterations is generated by merging the output of processing the condition with the initialInput to the `Do-While` loop. This takes care of whether this is the first execution of the body or an iteration. Once the fixed point has been achieved any `break` sets for this loop are handled.

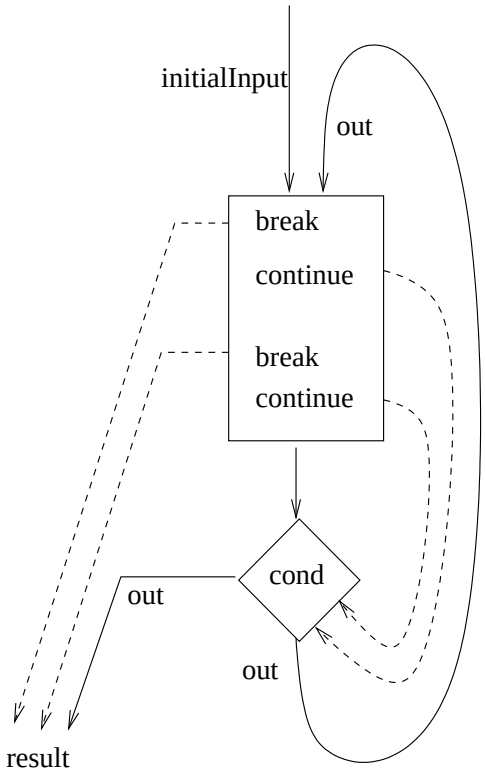
One important thing to note is that the handling of breaks takes as input the output set of processing the condition and not the newly generated input for the fixed point iteration. This is so because the loop has to execute at least once and hence the initialInput can never be part of the final result. Hence at Statement 13 in the pseudo-code shown in Figure 6.5(a) the input to `handleBreak` is the result of Statement 9 which contains the out-set of processing the `Do-While` condition. Once any break sets have been merged the result is the output of processing the `Do-While` loop.

`Unconditional-While` Construct

In an `Unconditional-While` loop the body of the loop keeps executing until there is a `break` out of the loop. Hence the only way out of the loop is through one or more `break` statements in the `Unconditional-While` body as shown in Figure 6.6(b). The processing of the loop is shown in Figure 6.6(a). The fixed point iteration starts off by processing

```
1 process_DoWhile(ASTDoWhileNode node,
2                   Object input){
3   initialInput = clone(input)
4   do{
5      lastin = clone(input)
6      out = processBody(node,input)
7
8      out = handleContinue(out,node)
9      out = processCondition(condition,out)
10
11     input = merge(initialInput,out)
12  } while(lastin != input)
13  result = handleBreaks(out,node)
14  return result
15 }
```

(a) Pseudo-code



(b) Graphical Representation

Figure 6.5: *The Structural Flow-Analysis Algorithm of DoWhile Construct.*

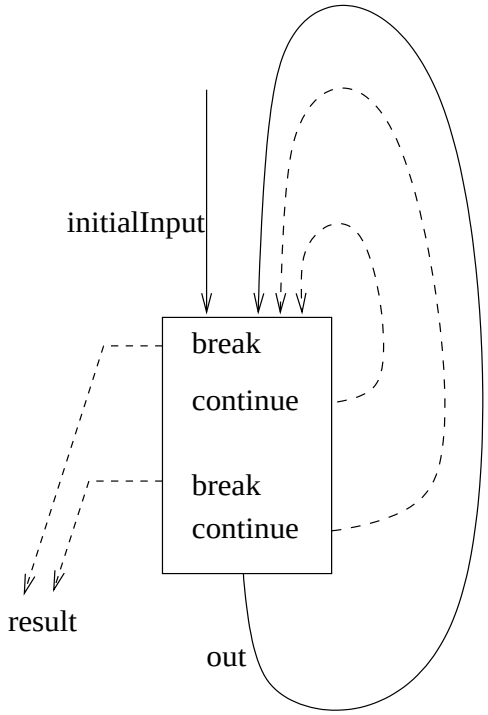
the body of the loop. Then any `continue` flow-sets are handled. Then the initial input is merged to create the input set for the next iteration of the loop. Once the fixed point has been achieved the `break` flow-sets are merged together to create the result of processing the `Unconditional-While` loop.

Notice that the result of processing the `Unconditional-While` body is sent as input to the `MergeBreaks` method. This is only used to retrieve the list of `break` flow-sets stored in the in-set and does not get included in the result since the only way out of the `Unconditional-While` is through a `break` statement.

```
process_UnconditionalLoop(
    ASTUnconditionalWhileNode node,
                    Object input){
  initialInput = clone(input)
  do{
     lastin = clone(input)
     out = processBody(node,input)
     out = handleContinue(out,node)

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

(a) Pseudo-code



(b) Graphical Representation

Figure 6.6: *The Structural Flow-Analysis Algorithm of Unconditional-While Construct.*

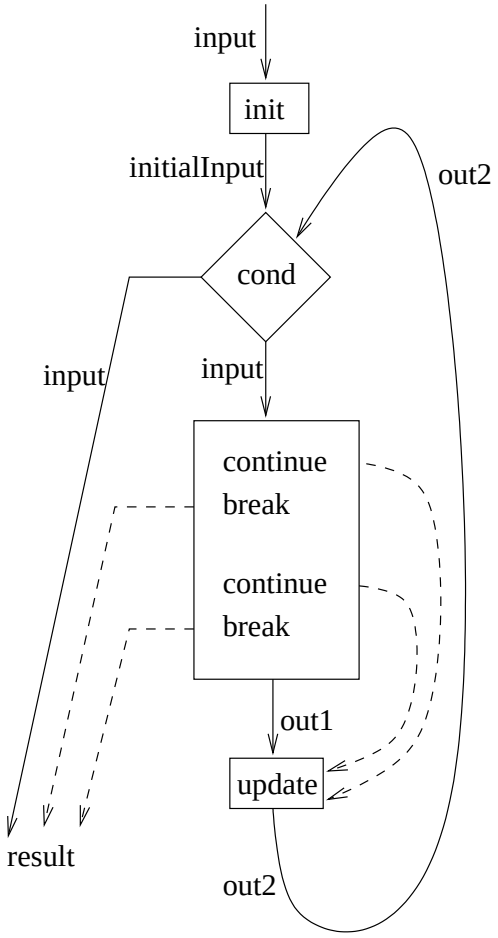
For Loops

The semantics of the `For` loop are discussed in Section 7.1.1. Briefly, when a `For` loop is encountered first the initializations are carried out followed by the evaluation of the condition. If the condition evaluates to `true` the body of the loop is executed followed by any updates to be performed. `Break` statements result in the termination of the loop and `Continue` statements target the update component of the loop (Figure 6.7(b)). The processing of the `For` loop is shown in Figure 6.7(a). First the `init` component is processed. Since this contains a sequence of statements it should be processed in the same way as any other `Statement-Sequence` block would be. Hence the `Statement-Sequence` flow analysis algorithm is invoked from within the algorithm of the `For` loop. Once this has been completed the condition of the loop is processed. The output of processing the condition becomes the input to the algorithm which computes the fixed point for the body of the loop.

```
process_for(ASTForNode node,
                    Object input){
  input = processInit(node,input)
  initialInput = clone(input)
  input = processCondition(condition,input)
  do{
     lastin = clone(input)
     out1 = processBody(node,input)
     out1 = handleContinue(out1,node)

     out2 = clone(out1)
     out2 = processUpdate(node,out2)
     //merge cond evaluating to false
     input = merge(initialInput,out2)
     input = processCondition(condition,input)
  } while(lastin != out2)
  result = handleBreaks(input,node)
  return result
}
```

(a) Pseudo-code



(b) Graphical Representation

Figure 6.7: *The Structural Flow-Analysis Algorithm of For Construct.*

This is done by first processing the body. This is then followed by handling any `continue` statements that might be targeting the update component of the loop. Once the `continue` flow-sets have been handled it is time to handle any update statements. The update part of the `For` loop can be empty hence the output produced by handling the `continue` statements is first copied into a new flow-set which is then used to process the update statements. The update statements are a sequence of statements and are processed by internally invoking the `Statement-Sequence` flow analysis algorithm. Once done, the input set for the next iteration is created by merging the initial input set to the output of processing the update

statements (Statement 14 in Figure 6.7(a)). As the regular execution *i.e.*, when no `break` is encountered always terminates at the evaluation of the condition the condition is processed again. Once the fixed point is reached any `break` statements targeting this loop are handled by merging their `break` sets together. The output from this becomes the output of handling the `For` loop.

`Switch` Construct

The processing of the `Switch` statement is shown in Figure 6.8. The algorithm starts off by processing the switch key. Since this component is always executed, the output from the processing of the key becomes the initial input to all the possible cases of the `Switch` statement.

The algorithm continues forward by first retrieving the different cases of the `Switch` statement. Then for each case the case Body is processed. The input set for these bodies is the merge of the initial input, if the case is the first case to be executed, and the previous case's output, since Java cases can have fall throughs (as shown in Figure 6.8(b)).

After the processing of a case the output set of each set is stored in the `caseBreakSet` list. This information is needed since it is the out-set of each case that stores all the `break` and `continue` sets which will be handled later in the algorithm.

While processing the `Switch` statement cases, another possibility that is checked is whether the `Switch` statement has a default case. If one is found the out-set of the default case is also stored (Statements 12 and 13 in Figure 6.8(b)).

A number of different execution paths can be taken for a `Switch` statement. Firstly it is possible that a `Switch` statement has no cases. Then the initial input `flow-set` should be the out-set of processing the `Switch` statement (Statements 16 and 17 in Figure 6.8(a)). If the `Switch` statement does contain one or more cases then there are two possibilities. First, a default case is present meaning that if no case matches the default case will be executed. Hence in this case the output becomes the merge of the default case with the conservative out-set of having processed all the cases of the statement. The latter out-set is available as the output of the processing of the last case of the `Switch` statement.

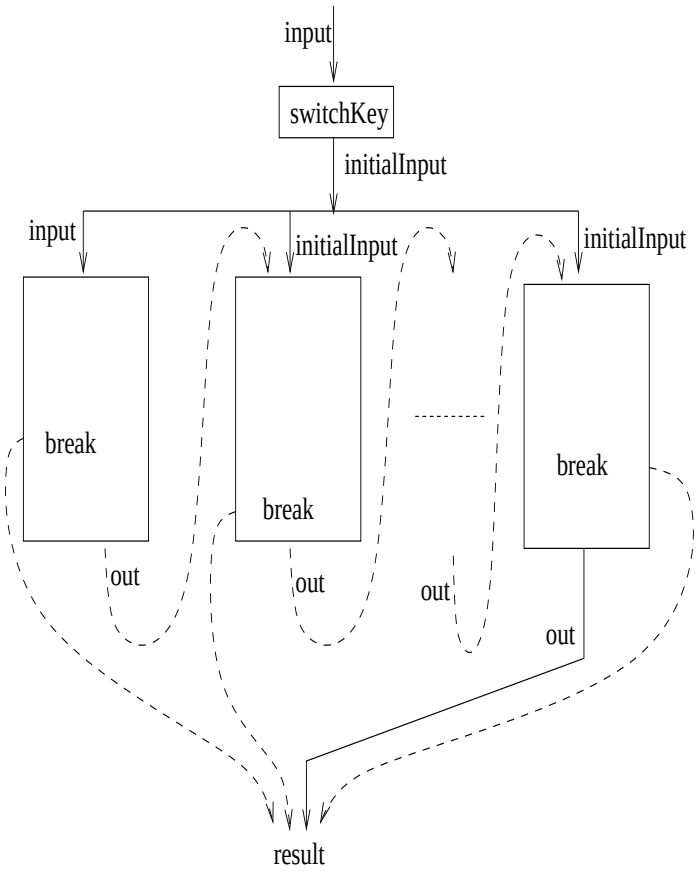
The second possibility is that if there is no default case present then it is possible that none of the cases in the `Switch` statement match the key. In this case the output is the

```
process_switch(ASTSwitchNode node,
                   Object input){
  input = processSwitchKey(key,input)
  initialInput = clone(input)

  Object default = null
  List caseBreakSet
  List cases = node.getSwitchCases()
  for each case, c in cases{
      out = processBody(c, input)
      caseBreakSet.add(clone(out))
      if(case is default case)
         default = out
      input = merge(out,initialInput)
  }
  if ( cases.size()==0 )
     output = initialInput
  else{
     if(default != null)
        output = merge(default,out)
     else
        output = merge(initialInput,out)
  }

  Object finalOut = output
  for each break set s in caseBreakSet{
      set = handleBreaks(s,node)
      finalOut = merge(output,set)
  }
  return finalOut
}
```

(a) Pseudo-code



(b) Graphical Representation

Figure 6.8: *The Structural Flow-Analysis Algorithm of Switch Construct.*

initial Input (since no additional code is executed). To handle the instance when a case does match, the output is the merge of the initial Input with the last out-set of the different cases.

Once we have the `output` from normal processing of the `Switch` statement the last thing to do is handle any `break` statements. This is done in Figure 6.8(a) Statements 25 to 29. The `break` sets stored for each case of the `Switch` statement (Statement 11) are retrieved. The `handleBreak` method is invoked on each individual `break` set to handle all possible `break` statements that might be present in that particular case. After merging the possibly different sets the result (`set` in Figure 6.8(a) Statement 27) is merged with the output of the regular processing. This is repeated for all the cases in the `Switch` statement (Statements 26 to 29). The output of this merging becomes the final output of processing a `Switch` statement.

`Try-Catch` Construct

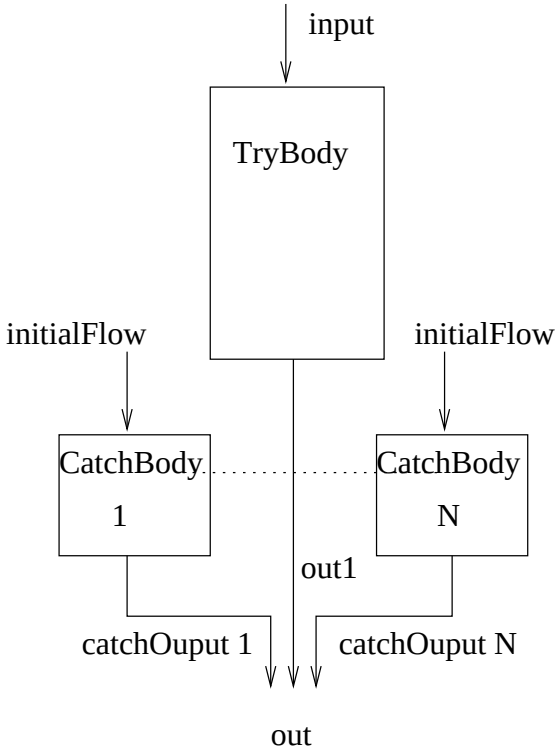
In the case of a `Try-Catch` block the algorithm needs to conservatively assume that either the try body will run to completion or one of the caught exceptions and the corresponding code will be executed. Also, since the code enscapsulated in the try component of the `Try-Catch` block (from here on called the try body) or any of the exception handlers (from here on called the catch bodies) can contain `break` statements these need to be handled correctly.

The algorithm starts out with processing the try body of the `Try-Catch` block. Then it processes each of the catch bodies. Notice that the input to each catch body is taken by invoking the `newCatchBodyInitialFlow` method (Statement 6 in Figure 6.9(a)). This method is one of the abstract methods declared by the flow-analysis framework and the analysis writer is required to provide an implementation for it. The purpose of the method is to take as input a conservative approximation for the input set of the catch bodies. Since it is not possible to predict which statement in the try block might cause an exception, it is prohibitively expensive to store each possible flow-set which could be the input to a catch body. Hence a conservative approximation is the best that can be handled in any reasonable amount of time and memory. Implemented analyses in Chapter 7 discuss possible conservative sets for some analyses. (Note: `newCatchBodyInitialFlow` is different from the

initial flow-set used to initialize an analysis. The initial flow-set, used as input to process a method, is a safe set for the analysis of a method whereas the `newCatchBodyInitialFlow` is a conservative approximation to the input of catchBodies occurring within a `Try-Catch` construct).

```
1  process_Try(ASTTryNode node,Object input){
2   tryBody = node.getTryBody();
3   tryBodyOutput = processBody(tryBody,input)
4
5   List catchBodyOutput
6   inputCatch = newCatchBodyInitialFlow()
7   for each catchBody , c in node{
8     in = clone(inputCatch)
9     out = processBody(c,in)
10    catchBodyOutput.add(out)
11  }
12  mergedOut = tryBodyOutput
13  for each out-set, catchOut in catchBodyOutput
14     mergedOut = merge(catchOut,mergedOut)
15
16  mergedOut = handleBreaks(tryBodyOutput,node)
17  for each catchOutput in catchBodyOutput{
18    breakout = handleBreaks(catchOutput,node)
19    mergedOut = merge(mergedOut,breakOut)
20  }
21  return mergedOut
22 }
```

(a) Pseudo-code



(b) Graphical Representation

Figure 6.9: *The Structural Flow-Analysis Algorithm of Try-Catch Construct.*

Also note that the same `catchBodyInitialFlow` set is cloned and passed as input to each of the catch bodies processed (Statements 8 and 9 in Figure 6.9(a)). The reason being

that only one of these catch bodies will ever be matched and hence the input should always be the same flow-set for all catch bodies. The result of processing the catch bodies are stored within the catchBodyOutput list (Statement 10 in Figure 6.9(a)). Once all the catch bodies have been processed the out-sets of these and the out-set if no exception is thrown (tryBodyOutput from Statement 3 in Figure 6.9(a)) are merged in Statements 12 to 14.

The last step then is the merge of all the possible `break` statements. Again it is important to remember that if the `Try-Catch` node has a label on it then either the try body or any of the catch bodies can target this label. Hence statements 16 to 20 ensure that all stored breaksets for the tryBodyOutput as well as the catchOutput's are correctly handled for `break-sets`. The result of merging the break-sets with the execution of the tryBody and/or one catch bodies becomes the final output of processing the `Try-Catch` block.

Chapter 7
AST rewriting using Structure-based Flow Analyses

In this section we discuss some structural analyses, and transformations that use information from structure-based analyses, to further improve code readability and comprehension. With the structure-based flow analysis framework, as described in the previous chapter, we now have the resources to gather any additional information required for more complex transformations. More precisely, we are now able to follow the flow of data through the AST and make conservative assumptions regarding the reachability, execution *etc.* of certain areas of the code. Figure 7.1 shows the internals of the back-end stage where flow analyses are performed, the results of which are then used to enable further transformations on Dava's AST. This stage is an iterative process since the application of a transformation may enable further transformations.

Figure 7.2 shows the implemented analyses (rectangles) and the transformations (diamonds) using information gathered by these analyses.

The analyses implemented (reaching definitions, reaching copies, must/may assign and constant propagation) are all well-known compiler flow analyses. An interesting observation is that usually these analyses are used by optimizing compilers for performance improvements. However, in the context of Dava we have used these analyses for code simplification.

In the remaining sections of this chapter we discuss the different analyses implemented

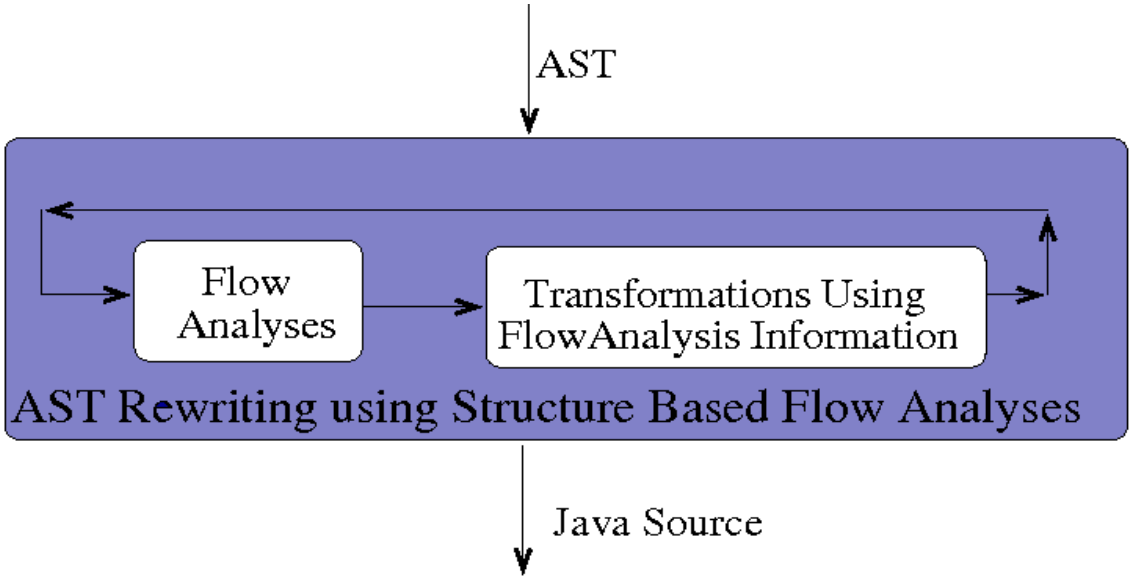


Figure 7.1: *AST rewriting using Structure-Based Flow Analyses*

along with transformations enabled because of these analyses.

7.1 Reaching Definitions

The reaching definition analysis is the basis of other structure-based flow analyses and is also used in transformations. *A definition d: x = <expr> reaches a point p in the program if there exists a path from p such that there is no other definition of x between d and p.* The analysis is a forward flow analysis and gathers sets of definitions that reach each program point.

The analysis is started by invoking the `process` method of the `StructuredAnalysis` class. The `process` method takes as input the body to be processed, in this case the method being analyzed, followed by the initial input set. The initial input set for reaching definitions is the empty set since no definitions reach the start of a method. The invocation of the `process` method is shown in Figure 7.3.

As seen from Figure 7.3 the merge operation is set union since the definitions reaching a particular point p is the combination of definitions reaching from all paths leading to p.

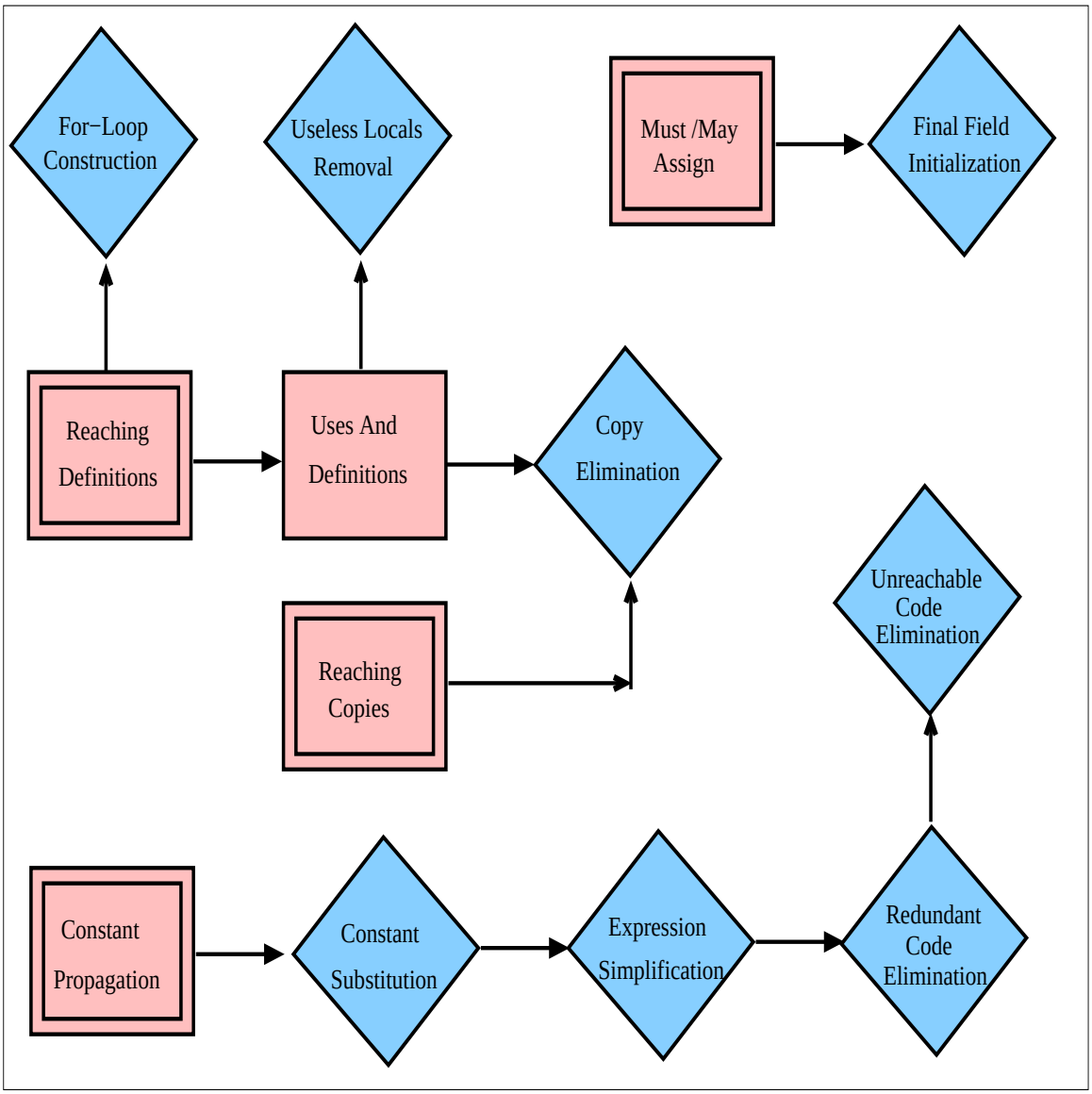


Figure 7.2: *Implemented Flow Analyses and transformations*

```
public class ReachingDefs extends StructuredAnalysis{
   ASTMethodNode toAnalyze;

   /*
    * Invoke the main process method to start processing the ''toAnalyze'' method node.
    * Notice the process method is sent an emptyFlowSet as initial input since the safe assumption
    * for reaching definitions is that no definition reaches the start of a method body.
    */
   public ReachingDefs(Object toAnalyze){
     super();
     this.toAnalyze = (ASTMethodNode)toAnalyze;
     DavaFlowSet temp = (DavaFlowSet)process(toAnalyze, emptyFlowSet());
   }

   //Implementation of inherited abstract method
   public DavaFlowSet emptyFlowSet(){
     return new DavaFlowSet();
   }

    // Implementation of inherited abstract method. Setting merge operator to UNION
    public void setMergeType(){
        MERGETYPE=UNION;
    }
```

Figure 7.3: *Initializing the Reaching Definitions Flow Analysis*

New reaching definitions are generated whenever a local variable is assigned a value. Hence, whenever such an assignment statement is encountered the current flow-set's information needs to be augmented with this new reaching definition. However, before this addition, any previous definitions of the same variable that are currently present in the flow-set need to be removed. Figure 7.4 shows how this is carried out by over-ridding the `processStatement` method of the `StructuredAnalysis` class. Briefly, the current reaching definitions in the flow-set are searched to find any that match the local variable be-

ing redefined. Any such definitions are removed from the flow-set. Then the new definition statement is added to the flow-set.

```
public Object processStatement(Stmt s, DavaFlowSet inSet){
   if(! (s instanceof DefinitionStmt))
      return inSet;

   DavaFlowSet toReturn = (DavaFlowSet)cloneFlowSet(inSet);
   Value definedVar = ((DefinitionStmt)s).getLeftOp();
   if(definedVar instanceof Local){
      // KILL any previous reaching defs of definedVar
      List currentReachingDefs = toReturn.toList();
      Iterator listIt = currentReachingDefs.iterator();
      while(listIt.hasNext()){
         //each entry is a reaching definition
         DefinitionStmt reachingDef = (DefinitionStmt)listIt.next();
         //we know this is a definition of a local
         if(definedVar.getName().compareTo(
                  reachingDef.getLeftOp().getName())==0){
            //need to kill this from the list
            toReturn.remove(reachingDef);
         }
      }
      //GEN: add stmt s to the toReturn flow set
      toReturn.add((DefinitionStmt)s);
      return toReturn;
   }
}
```

Figure 7.4: *Generating new Reaching Definitions and killing previous ones*

The extension of the `StructuredAnalysis` class also requires the programmer to provide an implementation of the `abstract newCatchBodyInitialFlow` method. As discussed in the previous chapter, this is the conservative assumption used in processing the

catch Bodies of any `Try-Catch` block found in the code. The `newCatchBodyInitialFlow` for the reaching definitions analysis is the universal set of all definitions in the method body. This is obtained using the `AllDefinitionsFinder` traversal discussed in Section 3.4 as shown in Figure 7.5. Once all the definitions are obtained, the initial input flow-set is populated with these definitions and becomes the input to the catch Bodies.

```
/*
 * Implementation of inherited abstract method. The Initial flow into catch bodies is
 * the universal set of all definitions in the method being analyzed.
 */
public Object newCatchBodyInitialFlow(){
  DavaFlowSet initial = emptyFlowSet();
  // Use an already implemented traversal routine to find all definitions in the method
  AllDefinitionsFinder defFinder = new AllDefinitionsFinder();
  toAnalyze.apply(defFinder);
  List allDefs = defFinder.getAllDefs();

  //allDefs is the list of all definition statements in the method
  Iterator defIt = allDefs.iterator();
  while(defIt.hasNext())
    initial.add(defIt.next());

   //initial is now the universal set of all definitions
  return initial;
}
```

Figure 7.5: *Input to catch Bodies for Reaching Definitions Flow Analysis*

The universal set of all definitions is used as input for catch bodies since during analysis we are not sure which statement of the try body will result in the exception being thrown. Figure 7.6 shows pseudo-code explaining this. Any statement in the try body can potentially throw an exception which can result in the execution of the catch body. Hence at the start of the catch body we don't know the exact flow set. One way of creating the correct

flow-set would be to merge data sets for all the statements of the try body and use that conservative assumption as the input set for the catch body. However, this requires a lot of memory. Therefore, it is better to be even more conservative and assume that all definitions reach the catch body.

```
1    d: x = ....
2    try{
3      ......... the definition d reaches this area
4      .........
5    }
6    catch(....){
7      .... the definition d might not reach this area
8    }
```

Figure 7.6: *Conservative reaching definitions assumption for input to catch bodies*

The results of the reaching definition analysis are used to compute uD-dU chains. The uD chain is a mapping of all definitions for a use of a variable. The dU chain gives all uses of a variable where a particular definition might reach. The uD-dU chains are useful while looking for complicated patterns. For example, modifications to the code that moves variable uses around needs this information since we need to make sure that the correct definitions of variables reach each use at all times.

A direct advantage of having this information is that looking at the dU chain we can find definitions which will never get used. These definitions can simply be removed as long as the definition does not have any other side effects *e.g.*, invocation of a method to assign to a field. In the next section we discuss the creation of `For` loops which wouldn't be possible without uD-dU information.

7.1.1 For Loop Construction

Certain conditional `While` loops can be represented more compactly as `For` loops. Programmers generally prefer to use `For` loops instead of `While` loops particularly when the

loop has a consistent increment on a particular variable. A `For` loop has four important components:

- **Init:** This is the part of the `For` loop where variables to be used in the loop body can be declared and initialized. The init is invoked once before the first iteration of the loop.
- **Condition:** The loop continues to execute as long as the condition of the loop evaluates to true. The condition is evaluated each time before the iteration of the loop.
- **Update:** This part of the `For` loop is executed at the end of each iteration. It is here than any updates of the variables can be done.
- **Body:** The body of the `For` loop consists of the code which is to be executed as long as the condition evaluates to true.

We define natural `For` loops as those loops where all four components of the `For` loop contain at least one expression/statement. The `While` to `For` transformation looks for patterns which can be converted into natural `For` loops. The pattern is shown in Figure 7.7(a).

(a) Unreduced

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

(b) Reduced

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

Figure 7.7: *The `While` to `For` conversion*

The general form of the reduction is shown in Figure 7.7(b). However, there are a number of restrictions on the different components and the transformation succeeds only if all restrictions are fulfilled. The procedure and the restrictions can be best explained by going through the algorithm for the transformation. Algorithm 9 outlines the steps taken to transform a `While` loop into a `For` loop. The body of an ASTNode is searched for a sequence of statements followed by a `While` loop. The statement sequence is the combination of Body A and Init Stmts in Figure 7.7(a). These statements are then analyzed to retrieve the init using the `GetInit` function.

Algorithm 9: The `While` to `For` conversion

```
Input: ASTNode node

body ← GetBody(node)
Iterator it ← body.iterator()
while it.hasNext () do
    node1 ← it.Next()
    node2 ← GetNextNode(node1)
    if node1 is a series of statements and node2 is a conditional while loop then
        init ← GetInit(node1)
        update ← GetUpdate(init,node2)
        newStmts←removeInitStmts(node1,init)
        stmtsNode ← ASTStatementSequenceNode(newStmts)
        condition ← GetCondition(node2)
        whileBody ← GetBody(node2)
        forNode ← ASTForLoop(init,condition,update,whileBody)
        Replace node1 and node2 by stmtsNode and forNode in body
    end
end
```

The `GetInit` function goes through the sequence of statements and gathers all statements that are initializing any variables. Once all such statements have been gathered they are analyzed to check whether the initialized variables are only used within the `While` loop

body. This information is readily available through the uD-dU chains created using the reaching defs flow analysis discussed in the previous section. If all uses of variables initialized in the `init` are present only in the `While` body then we know that the variable is live only within this body and hence the initialization is converted into a loop-local declaration and initialization statement.

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

If an `init` and `update` list are successfully retrieved then we can create the `For` loop. The first step is to create the sequence of statements that will replace the existing sequence (the combined Body A and Init stmts node of Figure 7.7(a)). This is achieved by the `RemoveInitStmts` function which goes through the statements and keeps only those which do not belong to the `init`. Basically we are left with Body A which is then used to create a new statement sequence node.

The `For` loop is then created with the condition of the `While` loop as its condition and the body of the `While` loop as its body minus the update statements which becomes the update part of the `For` loop. The new statement sequence node and the `For` loop then

replace the old statement sequence node and `While` loop in the AST.

```
Function: GetUpdate

Input: List init, ASTWhileNode node
Output: List update

body ← GetBody(node)
lastNode ← GetLastNode(body)
if lastNode is a statement sequence then
    stmt ← GetLastStmt(lastNode)
    if stmt is a definitionStmt then
        definedLocal ← GetDefinedLocal(stmt)
        if definedLocal occurs in init then
            update.add(stmt)
        else
            condition ← GetCondition(node)
            if definedLocal occurs in condition then
                update.add(stmt)
            end
        end
    end
end
return update
```

7.2 Reaching Copies

Copy statements are defined as statements of the form a = b where both a and b are variables. The reaching copies analysis, as implemented in Dava, tracks copy statements where both a and b are local variables. Fields were excluded from this analysis since tracking field values requires an inter-procedural context-sensitive analysis to be able to gather information useful enough to justify the cost of the analysis.

The analysis gathers sets of reaching copies where a copy statement reaches a program

point p if all paths leading to p pass through the copy statement a = b and the values of a and b are not changed between the copy statement and the statement p. For each copy statement, a = b, the analysis stores a local variable pair (a,b). It is a forward analysis which uses intersection as the merge operation since we are only interested in copy statements which definitely reach the program point p. When some local variable is assigned a value then any previous entries in the flow set are removed since the value of the variable is now changed. If the assignment to the local variable is from anther local *i.e.*, it is a copy statement then a new entry of the form (a,b), is added to the flow set.

The initial input to the method body is the empty set since no copies reach the start of the method. The input to the catch bodies is also the empty set since we cannot safely assume that a certain copy statement reaches a program point p within the catch body.

7.2.1 Copy Elimination

The copy elimination algorithm aims to remove useless copy statements from the code. In doing so it also minimizes the number of variables used in the program. A copy statement a = b is useless if at all places where variable a is used we could have used the variable b instead. We can use variable b instead of variable a if the value of a and b has not changed between the copy statement and its use. This information is available from the reaching copies analysis discussed in the previous section.

The transformation starts by looking for copy statements. When a copy statement is found it uses the dU chain, created using the reaching definitions analysis in Section 7.1, to find all the potential places that this definition might get used. Then the reaching copies analysis is used to find out whether at each potential use of this definition the flow set contains this definition as a reaching copy. If it does that means that the values of a and b have not been changed between the copy statement and its use. We can therefore remove the copy statement and use the variable b wherever there is a use of variable a.

Two real-world examples of copy elimination, from our benchmark suite, are shown in Figure 7.8. The unreduced form of the code in Figure 7.8(a) shows a copy statement `x=a;` which gets eliminated in the reduced version due to copy elimination. The use of variable x (line 3 in Figure 7.8(a)) has been replaced by the use of variable a in line 2 of

Figure 7.8(c). Similarly the copy statement of line 5 in Figure 7.8(b) is useless since the next line is the only use of this copied variable and there is no reason why we can't use the original variable in this use. Hence the use of r1 in line 6 of Figure 7.8(b) can be replaced by the use of variable e. The copy statement (line 5) is then useless and is removed from the code.

(a) Unreduced

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

(c) Reduced

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

(b) Unreduced

```
1  try{
2    BodyA
3  }
4  catch(Exception e){
5       r1 = e;
6       r1.printStackTrace();
7  }
```

(d) Reduced

```
1 try{
2   BodyA
3 }
4 catch(Exception e){
5   e.printStackTrace();
6 }
```

Figure 7.8: *Copy Elimination*

7.3 Constant Propagation

A constant propagation analysis aims to remove unnecessary use of variables in expressions. If the value of a field or local can be statically determined there is no reason why the code should not use that value instead of the variable. A more important advantage of constant propagation is that sometimes valuable information can be obtained regard-

ing conditional expressions in the code. For instance, in figure 7.9, the condition checks whether a variable x is less than the constant 5. The constant propagation data flow analysis can determine that at statement 2, before evaluating the condition, the value of x is 3. Hence statically it can be confirmed that the condition will evaluate to true. Therefore the condition need not be evaluated and code can flow straight to the target of the condition, side stepping the actual evaluation of the condition (in Figure 7.9 this means removing the `If` statement 2) and immediately executing code A after statement 1). Similarly, if code A does not change the value of x, then condition at statement 5 evaluates to false since constant propagation will know that x is still 5. Therefore, statements 5 to 7 can also be removed from the code.

(a) Original Code

```
1  x = 3;
2  if(x<5){
3    <code A>
4  }
5  if(x ==1){
6    <code B>
7  }
```

(b) Transformed Code

```
1  x =3
2  <code A>
```

Figure 7.9: *Advantages of constant propagation*

Second-generation obfuscators, those which go further than just renaming class members and local variables, rely heavily on confusing decompilers by producing complicated code guarded by opaque predicates. One form of opaque predicates is the use of conditions which never evaluate to true. Constant propagation can sometimes help the decompiler confirm that the condition is always false and the conditional statement along with its body can be discarded as dead code. Section 7.3.5 discusses this in more detail.

7.3.1 The analysis

Constant propagation is a forward data flow analysis. The analysis collects sets of local value pairs. *A local has a constant value at a program point p if on all program paths from the start of the method to point p the local has been assigned this constant value and this definition has not been modified from its definition point to the use at program point p.*

The merge operation is defined as the pair wise intersection using the following rules:

Value 1	Value 2	Result
$\perp$	$\perp$	–
C	$\perp$	C
C1	C2	C1 if C1 == C2 else $\top$
$\top$	$\perp$ / C / $\top$	$\top$

Table 7.1: *Intersection for Constant Propagation. ($\perp$ indicates unknown value and $\top$ represents a non-constant value)*

The flow equations for the flow analysis deal with assignment statements of the form *x = expr* where x is a local. The statement kills any known belief about the values of x in the current flow-set. The information obtained from the statement (hereafter called the gen set) contains an entry if one of two conditions is satisfied:

- *expr* is a constant value, C. In this case the gen set is the pair (x,C).
- *expr* is a local variable which has a constant value pair present in the current flow set. Supposing *x = y* is the statement and (y,C) belongs to the current flow set. Then the gen set contains the pair (x,C).

These flow equations, however, are not general enough and miss many opportunities to gather useful information. An example of this can be seen in Figure 7.10. In the figure the merge of the out-sets of B2 and B3 (the in-set of B4) will require the intersection of the pairs (j,2) from B2 and (j,$\top$) from B3. This means that the in-set for B4 will contain (j,$\top$)

according to our merge rules (Table 7.1). This is because the analysis does not interpret the relatively simple aggregated expression *i +1* and gives the value of $\top$ to j in B3.

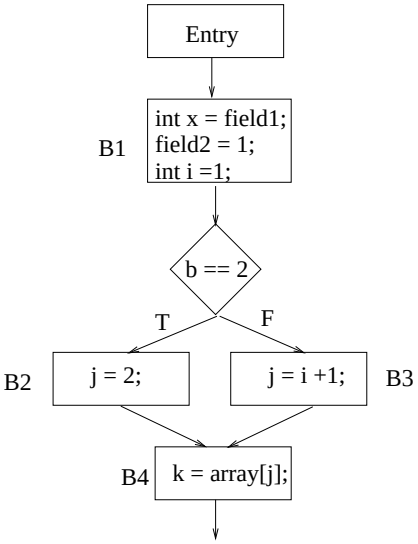


Figure 7.10: *Using constant field information during Constant Propagation*

The flow equations are strengthened by adding equations for assignment statements with expressions of the form *expr1 op expr2* on the RHS. Briefly: the new equations check whether expr1 and expr2 are constant values or have constant entries in the current flow set. If yes and if the operation is one of addition, subtraction or multiplication the operation is performed and this value is used to generate a pair for the local being assigned. Hence in Figure 7.10 the assignment statement *j = i+1* will result in the pair (j,2) since expr1 is i which has an entry, (i,1), in the in-set and expr2 is the constant value 1. Now the merge of (j,2) from B2 and (j,2) from B3 results in (j,2) to be present in the in-set of B4 which comes useful during the array access in B4.

A special case of this are the increment and decrement statements (i++ and i- -). In this case if the in-set before processing the statement contains a constant value for i the out-set contains the incremented/decremented value.

The initial flow set, when entering the method body, is the set of local value pairs with values for all locals set to $\perp$ since locals have no initial value and must be defined before use. However, values for formals of the method, which are also local variables, are

assigned the value $\top$ since they receive their values from calling sites for which we have no information. The input to the catch bodies is the set where formals and locals are all set to $\top$. This is so since we need to be conservative in our analysis and assume that none of the variables are assigned constant values.

7.3.2 Extensions

Using only local variables and only checking for simple expressions on the RHS of the assignment statement, as opposed to also looking for simple aggregated expressions, does not fully utilize the potential of constant propagation and gives weak results. Extensions to the analysis were implemented trying to gather a larger data set with information about more local variables.

Using constant value fields

The constant values field finder analysis of Section 3.5 can be used to increase the amount of information available to the analysis. To recap, this analysis gathers a list of all fields in the application which are either final fields, hence their value is constant throughout the program, or are fields which always get the default value. It is therefore logical to add this set of, known, constant fields to the initial in-set. Notice that this does not mean that the analysis is now an analysis on both fields and locals. All this extension allows is the presence of some additional information when deciding to create the gen set for an assignment statement. Figure 7.10 shows an example of this. Suppose *field1* is part of the constant value list provided by the constant primitive value finder analysis in Section 3.5. If we were not to use this information in our analysis then the gen set for the assignment statement *int x =field1;* in B1 would contain the pair (x,$\top$) since *field1* is a field and we do not track field values. However, if the in-set contained information about the constant value fields then the gen set would for this statement would be (x,0) since (field1,0) will be present in the in-set.

One thing to remember is that the only time a pair (x,const) where x is a field is added to the in-set is the entry to a method. All such pairs are created from the list of constant value fields provided by the constant primitive value finder analysis. In particular the statement

field2 = 1; contains an assignment to a field and is NOT added to the in-set. Although field information can help, in the same way as local information can, in the general case it is harder to track values of fields.

Conditional Expression results

Vital information about variables can be obtained from the conditional expressions in conditional statements: `If` and `If-Else` and the `Switch` construct. For instance, in Figure 7.10, the true branch of the `If` statement is taken only if the local b has the value 2. Hence while entering the basic block B2 we know that (b,2) is valid. Although this information is short lived *i.e.*, valid only within the basic block it can help gather information regarding other locals which might be valid even after the basic block ends. Depending on the type of conditional expression different beliefs can be generated. These are as follows:

- In an `If` statement if the conditional expression is a boolean variable then the variable holds the value true within the body of the `If` statement.
- If the conditional expression of an `If-Else` statement contains a boolean variable then one of two things can occur:
  1. If the variable is not negated, using the `!` symbol, then the boolean variable is true in the **then** branch and false in the **else** branch.
  2. If the variable is negated then the boolean variable is false in the **then** branch and true in the **else** branch.
- If an `If-Else` statement contains a binary comparison operation using the == or != comparison operators some information can be inferred about the operands. Assuming the conditional expression is *expr1 op expr2* then the types of inferences possible are shown in Table 7.2.

  Similar inferences can be made for the `If` statement for the == operator. One important point to be careful of is that if there is a previous constant belief about a local used in a conditional expr then that belief should get preference over any belief that might get added due to the conditional expression. The reason being that a belief

expr1	op	expr2	Result
constant	== / !=	constant	no information
constant	==	local	add (local,constant) to **then** branch
constant	!=	local	add (local,constant) to **else** branch
local	==	constant	add (local,constant) to **then** branch
local	!=	constant	add (local,constant) to **else** branch
local1	==	local2	if (local1,const) $\in$ in-set add (local2,const) to **then** branch else if (local2,const) $\in$ in-set add (local1,const) to **then** branch
local1	!=	local2	if (local1,const) $\in$ in-set add (local2,const) to **else** branch else if (local2,const) $\in$ in-set add (local1,const) to **else** branch

Table 7.2: *Strengthening Constant Propagation using Conditional comparison operations*

which is not generated within a condition has the chance to hold true after the condition whereas a belief generated by a condition only holds true within one of the branches of the condition. Figure 7.11 shows a code snippet which illustrates this.

```
1 a=2;
2 if (a==3){
3    <code A>
4 }
5 <code B>
```

Figure 7.11: *Preference to existing constant values*

In Figure 7.11 using constant propagation we know that the out-set of statement 1 will contain (a,2). The conditional expression in statement 2 will generate (a,3) for code A. However, if we were to add this pair to the in-set then the merge at the end of the `If` statement will try intersecting (a,2) with (a,3). This will generate the pair (a,$\top$) in the out-set which causes loss of information. In fact the condition in statement 2 will always evaluate to false and is dead code. Section 7.3.5 discusses

more on this. In short, a belief is only generated from a conditional expression if there is no existing belief regarding the variables involved prior to the evaluation of the expression.

The `Switch` statement can also give some information for the value of a local. Suppose the key for a `Switch` statement is a local variable. Then within a particular case of the `Switch` statement the value of the local is the same as the value checked in the case statement. Again if any previous constant entry exists then the previous entry gets preference since we know for sure that the particular case with a different constant value than the entry in the in-set will never get matched and is essentially dead code (Section 7.3.5).

7.3.3 Constant Substitution

The information gathered by the extended constant propagation analysis are used by a transformation routine which searches for uses of locals in the code. At each such use the constant propagation analysis results are queried to check whether we can statically determine the value of this local at this point. If such an entry is found the use of the local is replaced by the constant value. Some key things to keep in mind are:

- For querying the results of constant propagation on loops one needs to retrieve and query the out-set of the loop. This is because only the entries in the out-set hold true at all stages of the loop (first iteration, any middle iteration or when the exit condition holds).
- In a `For` loop any locals used in the *init* must be queried in the in-set of the `For` loop whereas the condition and the *update* should be checked using the out-set. The reasoning is the same as the case above.
- Conditional statements (`If` and `If-Else`) and all other statements in the code use the in-set for the statement to query for constant values for locals.

Immediately after applying constant substitution new uD-dU chains are created, using the reaching definitions analysis introduced in Section 7.1. This allows the application of useless local variable removal. Since local uses might have been substituted for constant val-

ues there is a good chance that some variable is declared and initialized but never used. All these are removed from the code.

7.3.4 Expression Simplification

A direct effect of applying constant propagation is that expressions can be simplified. Figure 7.12 illustrates this. The code in Figure 7.12(a) shows original code which is compiled and then decompiled with constant propagation enabled. The output is shown in Figure 7.12(b). It is clear that the local variable cleaner is not doing its job. The reason being that the implementation of the local variable cleaner only looks for definitions with locals or constants on the RHS. In Figure 7.12(b) we see that the RHS of statements 2 to 5 contain aggregated expressions. However, it is obvious that these statements can be simplified. An expression simplification pass of the AST is made after applying constant substitution. This results in code shown in Figure 7.12(c). Here the expressions were simplified by applying the operations being performed between different constants. The resulting statements were all of the form *local = constant*. The local variable cleaner then removes all of these statements. The expression simplification checks for binary operations of the form constant1 op constant2 where the operation can be addition, subtraction or division. The conversion is then made by evaluating the result of the operation and the binary operation is replaced by the constant value result. This is applied moving upwards from the lowest subtree of an expression tree all the way to the root resulting in the ability to simplify an expression with multiple operations.

A specialized form of expression simplification is conditional expression simplification. The aggregation patterns of Chapter 5 can create complex aggregated conditions. Constant propagation on these aggregated conditions can help replace some of the locals with constants. It is important to simplify conditions as much as possible since they play a vital role in program understanding. A number of simplification strategies are applied. These are briefly discussed below:

Simplifying unary boolean constants: This converts conditions of the form !true to false and !false to true.

Simplifying binary conditional expressions: These involve expressions of the form

(a) Original Code

```
1 int a = 2;
2 int b = a*3;
3 int c = a-b;
4 int d = c + a;
5 int e = 5;
6 int x = a +b +c +d +e;
7 System.out.println(a+b+c+d+e+x);
```

(b) After Constant Propagation

```
1 int i1, i2, i3, i5;
2 i1 = 2 * 3;
3 i2 = 2 - 6;
4 i3 = -4 + 2;
5 i5 = 2 + 6 + -4 + -2 + 5;
6 System.out.println(2 + 6 + -4 +
7                    -2 + 5 + 7);
```

(c) After Expression Simplification

```
1 System.out.println(14);
```

Figure 7.12: *Advantages of constant propagation*

expr1 op expr2 where the operation can be any of the relational operations (==,>=,>,<=,<,!=). If expr1 and expr2 are constants then the comparison is carried out and the binary expressions is replaced by its truth value obtained on evaluation. For instance 2==3 is replaced by false.

Simplifying complex aggregated conditions: These involve conditions aggregated together using && or || symbols. Aggregated conditions using the && aggregation symbol are first matched against Table 7.3. The first four rows are the truth table for boolean truth values for the && operator. The remaining for rows deal with && aggregation when one of the two conditions is a constant and the other an expression to be evaluated.

If Expr 1 is a constant boolean but Expr 2 is an expression to be evaluated then the result of the simplification is Expr 2 if the boolean constant is true (since now the RHS has to be evaluated) or is false if the boolean constant is false (since RHS will never be evaluated). In the case Expr 2 is a boolean constant Expr 1 is always evaluated. The condition can be simplified by removing the LHS constant if it is true but in the case the constant is false

Expr 1	Expr 2	Result
true	true	true
true	false	false
false	true	false
false	false	false
true	Expr 2	Expr 2
false	Expr 2	false
Expr 1	true	Expr 1
Expr 1	false	Expr1 && false

Table 7.3: *Simplifying the && condition*

we cannot remove the constant as the condition itself is always false. The reason is that even though we know that the condition is false we cannot simplify the condition to just the boolean constant false is because of any potential side-effects that might be caused by the evaluation of Expr 1. If basic tests can show that Expr 1 does not have any side-effects then this can also be removed to further simplify the condition.

Table 7.4 gives a similar simplification for the || operator. Reasoning about the simplification when dealing with one boolean constant and one expression is the same as that for the && operator. Using tables 7.3 and 7.4 the complex aggregated conditions are simplified as much as possible. As a last effort, if the condition still contains aggregation, we apply DeMorgans law. The law states that:

$$!A \;\&\&\; !B \equiv !(A \;||\; B)$$
$$!A \;||\; !B \equiv !(A \;\&\&\; B)$$

An example of this is shown in Figure 7.13.

7.3.5 Removing Redundant Conditional Statements

Once the conditional expressions have been simplified, after the application of constant propagation, it is sometimes possible to remove redundant `If` and `If-Else` statements

Expr 1	Expr 2	Result
true	true	true
true	false	true
false	true	true
false	false	false
true	Expr 2	true
false	Expr 2	Expr 2
Expr 1	true	Expr 1 || true
Expr 1	false	Expr1

Table 7.4: *Simplifying the || condition*

(a) Original Code

```
1 if (a && b || c && d)
2     return true;
```

(b) Decompiled Code

```
1 if ( (z0 && z1) ||
2        (!( !(z2) || !(z3))) )
3     return true;
```

(c) After DeMorgans simplification

```
1 if ( (z0 && z1) || (z2 && z3) )
2     return true;
```

Figure 7.13: *Simplifying conditions using DeMorgans Law*

from the code. If the conditional expression is an `If` statement, and if we know that the condition has been simplified to a boolean constant, one of two things can occur:

1. The condition is the constant: `true`. In this case we know that the body of the `If` statement will always be executed. However, removing the `If` statement and copying its body into the parent node can produce incorrect Java code. In Figure 7.14(a) one would assume that since the code inside the `If` statement is always executed there is no need to check the condition and the code inside can simply be moved out of the `If` statement. However, as seen in Figure 7.14(b) this can result in potentially uncompilable code since the code labeled `codeA` is dead code because of the return statement copied out of the `If` statement. Hence removal of such conditional statements is always followed by the analysis discussed next (Section 7.3.6). This analysis looks for unreachable pieces of code and removes it from the AST.

(a) Original Code

```
1 public void foo(){
2    <snip>
3    if(true){
4       return;
5    }
6    <codeA>
7 }
```

(b) Incorrect Transformation

```
1 public void foo(){
2    <snip>
3
4    return;
5    //javac compiler will give an error at this
6    //point due to unreachable code
7    <codeA>
8 }
```

Figure 7.14: *Removing always true `If` statement*

2. The constant condition is false. In this case the `If` statement along with its body is dead code and is removed from the code.

Similar to the `If` statement, if on simplification an `If-Else` statement contains a boolean constant one of two things are possible:

1. The constant condition is true. This implies that the **then** branch always executes. Hence the `If-Else` statement is removed and is replaced by the code in the **then** branch of the statement. Again simply moving the **then** body out of the `If-Else` statement can cause potential compilation errors due to reasons similar to those of removing the code out of a `If` statement with a `true` condition. The unreachable code analysis discussed in Section 7.3.6 is applied right after this transformation to remove any dead code produced.

2. The constant condition is false. This means that the **else** branch will be executed. The same pattern as the above is applied *i.e.*, the `If-Else` statement is removed and replaced by the **else** branch of the statement. The unreachable code elimination transformation discussed is applied immediately afterwords to remove dead code.

7.3.6 Unreachable code Elimination

The unreachable code detection is carried out using a structure-based flow analysis. A program point p is considered unreachable if there is no path from the start of the method which can lead to program point p. SOOT already includes a dead code eliminator which eliminates any dead code present in the bytecode read from the class file. However, certain analyses like the redundant condition elimination discussed in the previous section can produce unreachable code. The analysis traverses the AST flowing `canReach` information as it processes different Java constructs. The flow set of the analysis always contains one entry which is `true` if this path is reachable and `false` otherwise. Abrupt statements *i.e.*, `break`, `continue` and `return`, change the `canReach` information to `false`. The merge operation is the `OR` operation *i.e.*, if both flow sets contain `false` then the output is `false`. In all other cases the flow set contains `true`.

One interesting thing about the analysis is that the processing of the loops does not need a fixed point computation. The processing rules for some of the interesting constructs are listed below:

- If a loop is reachable then the construct following the loop is always reachable. This is in accordance with the Java language specifications. In the case of conditional

loops since the loop condition might not evaluate to `true` hence the construct following the loop is always reachable if the loop itself is reachable. For unconditional loops either the loop is intended to be an infinite loop or the next construct is reachable from a `break` from within the loop.

- If an `If` statement is reachable then the construct following this statement is also reachable. Again since the condition within the `If` statement might not evaluate to `true` the next construct is reachable.

- For an `If-Else` statement the construct following the `If-Else` statement is reachable as long as one of the branches of the `If-Else` statement targets the natural fall through of the `If-Else` construct. In Figure 7.15(a) CodeC is reachable as long as codeB does not end with an abrupt statement. Figure 7.15(b) shows how codeC can be unreachable since both branches of the `If-Else` statement sidestep the execution of codeC.

- Any labeled construct if targeted by a reachable `break` statement is itself reachable.

7.3.7 Program Deobfuscation

A practical use of constant propagation along with the expression simplification and dead code elimination is seen in the case of decompiling obfuscated code. As mentioned in Section 7.3 second-generation obfuscators introduce complicated code into the program being obfuscated. This code is never executed since it does not actually do anything meaningful. One way of preventing the code from executing is to place the code within an `If` statement whose condition never evaluates to true. An example of this is shown in Figure 7.16(a). The code is the Dava output without constant propagation for a program obfuscated using the Zelix KlassMaster [Klaa] obfuscator. Statements 23-28 is code guarded by the boolean flag z0. An inspection of the program shows that the only place z0 is assigned a value is statement 8. Tracking the value of c which is being assigned to z0 shows that this is in fact a boolean field which is never assigned a value. Since a field which is never assigned a value receives the default value this implies that c and hence z0 after statement 8 have the

(a) CodeC is Reachable

```
1 if(cond){
2    CodeA
3    return;
4 }
5 else{
6    codeB
7 }
8 CodeC
```

(b) CodeC is Unreachable

```
1 label1:{
2    if(cond){
3       CodeA
4       return;
5    }
6    else{
7       codeB
8       break label1;
9    }
10   CodeC
11 }
```

Figure 7.15: *Reachability analysis for the `If-Else` statement*

default value false. Hence the condition in statement 23 always evaluates to false and the code 24-27 is never executed and is dead code.

Figure 7.16(b) shows the effects of applying constant propagation followed by local variable cleaning. The boolean variable z0 is detected to hold a constant value, false, after Statement 8 in Figure 7.16(a). Hence all uses of z0 in the code (Statements 14, 18 and 23) are replaced by the constant false. Statement 8 of Figure 7.16(a) becomes useless and is removed from the program.

Looking at Figure 7.16(b) we see that condition simplification will simplify the condition in Statement 13 to true. This means that the `If` Statement 13 is un-needed and in Figure 7.17(a) has been removed from the code by replacing it with its body. Similarly Statement 17 of Figure 7.16(b) contains the condition false. The code is dead code and is removed from the program. Looking at the condition in Statement 22 again we see that the `If` Statement 22-27 will never be executed as the condition is false. Hence this code is also removed from the output. Once dead code elimination has been applied to the program there is a strong chance that the AST transformations (Chapter 5) might be able to simplify

(a) Decompiled obfuscated Code

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

(b) Code After constant propagation

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

Figure 7.16: *Advantages of constant propagation*

(a) Dead Code Elimination

```
class a{
  private Vector a;
  public static boolean b,c;
  int a(String r1){
    boolean $z2, z3;
    int i0, $i2, i3;
    String r2;

    i0 = 0;
    label_1:{
      label_0:
      while (i0 < a.size()){
        r2 = (String) a.elementAt(i0);
        z3 = r2.equals(r1);
        i3 = z3 ? 1 : 0;
        $i2 = i3;
        if (i3 == 0) i0++;
        else{
          a.remove(i0);
          return i0;
        }
      }
      $i2 = -1;
    } //end label_1:
   return $i2; }
```

(b) Reapplying AST Transformations

```
class a{
  private Vector a;
  public static boolean b,c;
  int a(String r1){
    boolean z3;
    int $i2, i3;
    String r2;
    for (int i0=0; i0<a.size();i0++){
      r2 = (String) a.elementAt(i0);
      z3 = r2.equals(r1);
      i3 = z3 ? 1 : 0;
      if (i3 != 0){
        a.remove(i0);
        return i0;
      }
    }
    $i2 = -1;
    return $i2;
  }
}
```

Figure 7.17: *Dead code Elimination and AST Transformations*

the resulting AST. Hence the set of transformations are reapplied to the AST. The resulting output is shown in Figure 7.17(b). Notice that the labeled blocks have been removed since dead code elimination removed the abrupt edges targeting these labels. Also notice that the `If-Else` statement (Statements 17 to 21 in Figure 7.17(a)) has been converted to an `If` statement using the abrupt `If-Else` splitter analysis in Section 5.3.1.

Also the `While` loop (Statements 12 to 22 in Figure 7.17(a)) has been converted to a `For` loop since the transformation discussed in Section 7.1.1 was matched.

One other interesting thing to note is that Statements 17 and 18 of Figure 7.17(b) suggest that reapplying constant propagation after AST transformations will simplify the code further. However, in our opinion the costs of constant propagation are high enough that this should not be included within a fixed point computation of the AST. We therefore leave these statements unchanged.

7.4 Must and May Assign

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

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

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

7.4.1 Final Field Initialization

The Java Language specifies that all instance variables of a class that are declared `final` should be initialized at the time of construction of the object. Static final fields have to be initialized as part of the declaration or in the static initializer block. Non-static final fields need to be initialized as part of the declaration or within all constructors of the class defining this instance variable. If the initialization of a final field takes place within a code body *i.e.*, not as part of the declaration statement then the field needs to be declared on ALL paths within the code body.

When decompiling code produced by a Java compiler all field initializations are handled correctly as the bytecode necessarily contains initializations of the fields either as part of the constructors or as class attributes that can be retrieved.

Things start to get tricky when the bytecode being decompiled originates from a different source than a standard Java compiler. At the bytecode level there is no restriction for final fields to be necessarily initialized. Hence decompiling bytecode produced from a bytecode optimizer like Soot or code generated by other compilers, such as AspectJ, can easily lead to decompiled output which violates the Java specifications. Java obfuscators in fact exploit this by introducing uninitialized fields in the bytecode since they will lead to uncompilable code once decompiled. Figure 7.18(a) shows such an example. The field `myField` is declared final, but is never initialized. A Java compiler will not compile this code since it violates the language specifications. Figure 7.18(b) shows decompiled pseudo-code which can be produced when decompiling bytecode produced using an AspectJ compiler. In this case the Java language specifications are violated since the field `myField` is not initialized on all paths in the method `foo`.

In order to generate recompilable code for bytecode produced by compilers other than the standard `javac` compiler and to thwart obfuscators we have written a transformation which relies heavily on `mustInitialize` and `mayInitialize` analyses, discussed in the previous section. The aim of this transformation is to ensure that if a field is declared as final then it is always must initialized in the constructors of the class. We discuss this transformation in the following sections.

(a) No assignment to final field

```
1 class FinalField{
2    public final int myField;
3
4    public FinalField(){
5    }
6 }
```

(b) Final field not initialized on all paths

```
1 final int myField;
2 public void foo(){
3    BodyA
4    if(cond){
5      BodyB
6      myField = <assignment>;
7      BodyC
8    }
9    BodyD
10 }
```

Figure 7.18: *Example of final field not initialized on all paths*

The Indirect Assignment Algorithm

To ensure that all final fields are always initialized the `processField` algorithm is invoked for each field. The static modifier of the field is checked. If the field is static then the algorithm only proceeds forward if the current method is the static initializer. Similarly if the field is non-static then the algorithm proceeds only if the current method is a constructor of the class.

A check is then made to see whether the field has a tag associated with it. These tags are created by SOOT from class attributes and contain information about the constant values in the application. If the field has a tag associated with it then the value for the field is retrieved from the tag. If no tag is found then this indicates that the assignment to the field is being carried out either in the static-initializer or the constructors of the class. In this case there is a need to confirm that the field is initialized on all paths.

The `isMustInitialized` method of Algorithm 10 checks to see whether the final field is initialized on all paths of the method being processed. The method uses information from the structure-based `must` assign flow analysis, discussed in the previous section. Using the

must assign analysis the method returns true if the final field is in fact assigned on all paths in the method being analyzed. This guarantees that at compile time the code will not result in a "final field not initialized error". If isMustInitialized returns false then we know that compilation of this method would result in a compilation error as the field has not being initialized on all possible paths.

Algorithm 10: processField

Input: SootField *field* , ASTMethodNode *method*

```
if !isFinal(field) then
    return;
if !isStaticInitializer(method) || !isStatic(field) then
    return;
else if !isConstructor(method) || isStatic(field) then
    return;
if hasTag(field) then
    return;
if isMustInitialized(field) then
    return;
if !isMayInitialized(field) then
    addDefaultAssignmentStatement (method, field);
else
    defs ← getDefs(field);
    handleAssignOnSomePaths (method, field, defs);
```

In the case that a field is not must assigned a may assign analysis is applied (Section 7.4) using the isMayInitialized method. Let us first consider the case where the isMayInitialized method returns false. This indicates that there is no assignment of a field on any path through the method. In this situation the decompiler adds a statement assigning a default value to the field. This is achieved using the addDefaultAssignmentStatement function. The function checks the type of the field and accordingly adds to the AST a default assignment statement of that type: object fields are assigned null, integers the value 0, booleans are set to false etc.

Now considering the case when isMayInitialized returns true: this indicates that there is an assignment to the field on at least one path through the program. Given that the

field is not `must` initialized (`isMustInitialized` returned false), we need to transform the AST such that the `must` initialize condition is fulfilled. This is handled by `handleAssignOnSomePaths` which we now discuss.

Algorithm 11: handleAssignOnSomePaths

```
if defs.size() != 1 then
    cancelFinalModifier(field);
    return;
end
allUses ← getUses(field);
if allUses != null && allUses.size() != 0 then
    cancelFinalModifier(field);
    return;
end
clonedMethod ← clone(method);
newMethod ← createIndirection(cloneMethod,field);
if isMethodCallSafe(newMethod) then
    replaceMethodBodies(method,newMethod);
return;
```

The aim of the `handleAssignOnSomePaths` function is to rewrite the AST such that the field under observation, currently satisfying the `may` initialize property, be `must` initialized. The only reason why a variable is `may` initialized is that all assignments to the variable are nested within some control flow path which might or might-not be taken. This is shown in Figure 7.19(a) where the field is `may` assigned since its assignment is within the `If` statement. The approach followed by the algorithm is to delay the assignment within the nested control flow as much as needed such that it lies on the `must` initialize path. In the case of Figure 7.19(a) this means delaying the assignment to "field" until after the `If` statement.

A few things have to be kept in mind while doing such a delayed assignment. One of them being the value of the field if the path that did assign to the field is not taken. In our example what should be the value to field if `cond` evaluates to false? The suggested transformation to delay the assignment of the field is to use a dummy variable of the same

(a) May assigned field

```
1 public void foo(){
2   BodyA
3   if(cond){
4     BodyB
5     field = <assignment>;
6     BodyC
7   }
8   BodyD
9 }
```

(b) Delayed Assignment makes field Must assigned

```
1 public void foo(){
2   <Field Type> tempField;
3   tempField = <default>
4   BodyA
5   if(cond){
6     BodyB
7     tempField = <assignment>
8     BodyC
9   }
10  field = tempField;
11  BodyD
12 }
```

Figure 7.19: *Delaying assignment of a final field*

type as the field being assigned (variable tempField in Figure 7.19(b)). This variable is then assigned a default value depending on the type of the field (object types get null, booleans get `false` etc). Then the assignment to the actual field is substituted by an assignment to the just created dummy field. A position in the code is then found where the original field is assigned the value from the dummy field (in Figure 7.19(b) this position is right after the end of the `If` statement). By doing this we have moved the assignment of the field to a `must` assign path. In the case that the `may` assign path is taken, the field is assigned the intended value. On the other hand if that path is not taken (cond evaluates to false) then because of the default assignment to the temporary variable the field is also assigned the default variable.

Delaying such initialization is tricky and we only deal with the cases where there is only one assignment of the field that has to be delayed. Also, if in the original code the field is used after it has been defined we are unable to delay the assignment since then it is essential that the delayed statement be above all uses of the field. In our transformation we

delay the assignment to JUST as much as is needed to put the field assignment on the `must` initialize path.

The `May Assign` structure-based analysis not only tells us whether a particular variable may have been assigned on some path in the program but also stores the different definition (assignment) statements that might be executed. The `handleAssignOnSomePaths` checks whether there are more than one definition statements of the field within the code. If there are more than one definitions, then the analysis gives up and invokes the `cancelFinalModifier` method which will remove the final keyword from the field's declaration (Remember that only final fields `must` be assigned values). If there is only one definition then the algorithm checks whether there is any use of this field within the body of the method. If there are any uses then the algorithm gives up trying to delay the assignment to the field. In this case also the final keyword is removed from the field.

However, if there is only one definition of the field and the field is not used after its definition then the algorithm continues with its "delaying of assignment" approach. This is achieved by invoking the `createIndirection` method. Once the delayed method body has been created one last thing that needs to be checked is that there is no method call between the original assignment of the field and the new position of assignment. This is necessary since we are delaying the assignment of a field which might be accessed by other methods. Conservatively we restrict the transformation to only those instances in which there is no method invoked between the old and new position of assignment.

Let us look in more detail the workings of algorithm to create the indirection. Algorithm 12 shows the pseudo-code for the creation of delayed assignment. Briefly explained the algorithm works like this:

- Create a new local variable with the same type as the final field
- Add this variable to the list of locals in the method under process (Statementt 1 in Figure 7.19(b))
- Create a default assignment statement for this new local variable
- Add default assignment statement to method body (Statement 2 in Figure 7.19(b))
- Modify the current assignment statement of the field by assigning the value to the

new local (Statement 3 in Figure 7.19(b))

- Create new indirect assignment statement of field using new variable
- Find the correct position in the method body to place this statement (Statement 4 in Figure 7.19(b))

The last part of this algorithm deserves further discussion. Our aim is to delay the assignment to the field till as late as it is necessary. `createIndirection` does this by trying to place the new assignment statement in the parent of the node in which it originally existed. If this does not result in `must` initialization the algorithm tries the grandparent and rechecks `must` initialize. If that does not work then the great-grandparent is checked and so on. Using this algorithm we are guaranteed that the first ancestor at which `isMustInitialized` returns true will be used to place the new assignment to the field.

With the help of this transformation Dava is able to ensure that there are no compilation errors resulting from final fields not being initialized. If a final field is not assigned on all paths then either the final keyword is removed or in some cases the assignment is delayed to the point that the field is in fact assigned on all paths.

Algorithm 12: createIndirection

Input: SootField *field* , ASTMethodNode *clonedMethod*

```
// Create and add local for indirect assignment
localType ← getType(field);
newLocal ← new JimpleLocal(uniqueName,localType);
addNewLocal (clonedMethod,newLocal);
// Initialize newly created local to default value
initStmt ← createDefaultStmt(newLocal);
index ← 0
addStatement (clonedMethod,initStmt,index);
// Assign required value for field to new local
defStmt ← getDef(field); defStmt.setLeftOp(newLocal);
// create indirect field assignment statement
assignStmt ← new AssignStmt(field, newLocal);
// Add indirect assignment at the first possible place
parent ← getParentOf(defStmt); grandParent ← getParentOf(parent);
while ! isMustInitialized(field) do
    if isMethodNode(grandParent) then
        throw new DavaError("Unable to must-initialize");
    ancestor ← getParentOf(grandParent);
    ancestorSubBody = ancestor.getSubBodyContaining(grandParent);
    index ← ancestorSubBody.indexOf(grandParent);
    addStatement(ancestorSubBody,assignStmt,index);
    if ! isMustInitialized(field) then
        // problem not solved remove the stmt just added
        ancestorSubBody.removeStatement(assignStmt);
        // we should put assign in one level above than current
        grandParent ← getParentOf(grandParent);
    end
end
return clonedMethod;
```

Chapter 8
Naming Mechanism

Local variable names present in Java source code may be lost at compile time. At the same time the most common obfuscation technique is to rename all identifiers in an application to meaningless and often confusing names. Until recently Dava had a very naive naming strategy for allocating names for local variables in the decompiled code, the result being source code with hard to follow variable names.

The new Dava back-end now contains a naming stage where all identifiers in an application (class names, methods, fields and local variables) can be renamed. The reason for including non-local variables as part of the namer stems from the fact that obfuscators most often use name obfuscation to confuse the code. With a naming mechanism for all identifiers in the application we hope to be able to build some contextual information of the program and convey that to the programmer via identifier names.

8.1 Heuristic-based naming

There are many attributes that contribute to how a programmer names a variable. Some basic ones that are easily identifiable are used to provide rudimentary renaming to variables in Dava. The future work (Section 11.1) discusses ideas on further improving the naming mechanism.

- **Variables used in `For` loops:** It is common practice to use variables named i, j or k for driving variables in `For` loops.

```
for(int i=0;i<var;i++){
   //for loop code
}
```

Figure 8.1: *`For` loop driving variables*

- **Variables used as flags:** Variables that have `boolean` types are usually used as flags. They can be used to terminate `While` loops or used in `If`/`If-Else` statements. When used in a `While` loop they represent code as shown in Figure 8.2(a). The variable `notDone` is used as a flag to terminate the `While` loop when a certain condition is satisfied. Such variables can be called `flags`.

```
while(notDone){
   //while loop code
}
```

```
if(isFinished){
    //then code
}
```

Figure 8.2: *Conditional Flags*

- **Variables used to hold size or length of a data structure:** In Java many classes, implementing data structures, contain the method `size` or a field `length`. Hence a variable with the same name can give good contextual information regarding the data it is holding.

- **Variables declared `final`:** It is common programming practice to name `final` fields with names with capital letters (Figure 8.3(b)).

- **Variables whose exact names can be obtained:** The use of `get` and `set` methods in Java gives additional hints regarding the use of a variable. Since method names are conserved during compilation, an assignment from a `get` method can be used to

```
int length = classObject.length;
int size = classObject.size();
```

```
final int DIRECTION=1;
final int SIZE = 10;
```

Figure 8.3: *Heuristics for size/length and* `final` *variables*

name a variable. Similarly an argument to a `set` method can be given the name of the `set` method.

```
id = classObject.getId();
name = classObject.getName();
index = classObject.getIndex();

classObject.setSize(size);
classObject.setX(x);
```

Figure 8.4: *Using* `get` *and* `set` *methods to get variable names*

- **Exception Names:** It is common practice to name exception variables with the first letters of each identifier making up the exception converted to lower case. For example a variable of type FileNotFoundException can be named fnfe and an IOException variable can be named ioe.
- **Main method argument:** A rather trivial heuristic, only applicable to the `main` method of an application, this heuristic looks for the main method and names the argument of the method to `args`.
- **Arrays:** If a better name for an array variable is not available then one can append the type of the variable to the string "array" to convey to the programmer that this is an array. Hence we can have variables with names intArray or nodeArray *etc.*.

- **Local assignment using fields:** Since compiled code contains field names, a local variable assigned a field value can be given a name similar to the field.
- **Object type:** If a local is assigned the result of creating a new object or if an object is cast to a particular type then the type of the variable can be used to decide on the name of the variable.
- **Remove confusing characters:** Confusing symbols should in all cases be removed from variable names. These include the use of $ symbols, generated by SOOT for internal (stack) variables. At the same time obfuscators tend to add other confusing characters such as a sequence of underscores or combinations of the letter S and the digit 5. The renamer looks for such sequences and removes them.

8.2 Displaying qualified types

Java bytecode represents objects with their fully qualified types. For instance, if a class extends the `Thread` class the class definition would contain "`extends java.lang.Thread`". Similarly, a field or local of type `String` would have the definition `java.lang.String`. This ensures that all types are explicit and no confusion occurs when executing code for objects of the same class names, but belonging to different packages. An example of this can be the use of `Timer` objects in Java. The `java.util` package and the `javax.swing` package both contain a `Timer` class. Hence in this case, or in any application that uses different classes with the same name, it is critical that there be no type ambiguities.

Type ambiguities are handled by restricting the Java compiler to only allow unambiguous types at compilation. Hence in the presence of only one imported `Timer` class it is legal to use "`Timer t`" to define a `Timer` object with name t which belongs to whichever type is imported in the class definition (In Figure 8.5(a) the timer object t has type `java.util.Timer` since that is the imported class). However, if multiple Timer classes have been imported then the user has to explicitly refer to each type. The code in Figure 8.5(b) shows a Java program which will produce compile time errors since the packages `java.util` and `javax.swing` both have Timer classes. Statements 4 and 5 define ambiguous Timer objects and need to be fully qualified in order for the program to compile.

(a) Legal variable declarations

```
1 import java.util.Timer;
2 public class TimerTest{
3    Timer t;
4 }
```

(b) Illegal variable declarations

```
1 import java.util.*;
2 import javax.swing.*;
3 public class TimerTest{
4    Timer swingTimer;
5    Timer utilTimer;
6    <code>
7 }
```

Figure 8.5: *Qualified Variable types*

Another related, and important, restriction is that in the case of importing two classes with the same name it is illegal to import the fully qualified class names. Figure 8.6(a) shows two illegal import statements (Statements 1 and 2). If classes from different packages but with the same name have to be imported then instead of importing the classes the packages need to be imported. Figure 8.6(b) shows the correct version of the code. Notice that the Timer objects (Statements 4 and 5) are created using the fully qualified type name.

(a) Illegal Class Imports

```
1 import java.util.Timer;
2 import javax.swing.Timer;
3 public class TimerTest{
4    <code>
5 }
```

(b) Legal Imports

```
1 import java.util.*;
2 import javax.swing.*;
3 public class TimerTest{
4    javax.swing.Timer swingTimer;
5    java.util.Timer utilTimer;
6    <code>
7 }
```

Figure 8.6: *Importing classes with the same name*

When decompiling bytecode, the original Dava front-end always produced code with fully qualified type names even though most of the time the types are unambiguous. This resulted in verbose code. A back-end transformation has now been implemented which converts unambiguous types to their truncated form. An important requirement for deciding when a type is ambiguous is knowing exactly which classes have been imported. Hence, the first step for this transformation is to detect all Java classes that need to be imported. This is done by processing the entire Java class being decompiled and storing all references to library and application classes. Note that we do not store the list of packages to be imported but the individual classes that are needed by the Java class. This is necessary since we intend to look for cases when two classes with the same Java class name but belonging to different Java packages are imported.

The removal of fully qualified class names occurs at the time the decompiled code is being output. The transformation implemented checks whether a class type is being printed. At this time the truncated name of the type being printed is checked with the list of imported classes. If the import list contains multiple classes matching the truncated name then the removal of the fully qualified name for this type will result in an ambiguity. If only one match is found then the qualified name can be truncated. Hence, looking back at Figure 8.5(a) when the decompiler is printing statement 3, the declaration of the `java.util.Timer` object, the type name can be truncated since the import list only contains one `Timer` class.

If an ambiguity exists *i.e.*, the import list contains two classes belonging to different packages with the same name, then not only can we not truncate the fully qualified name of the class but we also need to import the entire package instead of explicitly importing the class. This is shown in Figure 8.6(b). Statement 4 and 5 are two declarations of `Timer` objects belonging to different packages. When the decompiler creates the import list both `java.util.Timer` and `javax.swing.Timer` will be present in this list. When the types of the declaration statements are being printed the list will be searched for the truncated name, `Timer`. Since multiple occurrences of this name will be found the type names in statements 4 and 5 will be left un-truncated. At the same time the import statements for the two `Timer` classes are marked such that instead of printing the explicit imports to the two

`Timer` classes their respective packages are imported, as seen from statements 1 and 2 in Figure 8.6(b). Using this transformation most of the types (fields, formals, locals etc) get truncated names since ambiguities rarely exist.

Chapter 9
Testing and Empirical Results

The key requirement in our implementation has always been the correctness of the transformations. Previously, Dava produced semantically correct but complicated output. The newly introduced back-end aims to improve the code quality but should not do so at the expense of producing incorrect code. Great care has been taken to ensure that the semantics of the program don't change because of the transformations performed. This requires not only confidence in the correctness of the transformation but also testing the semantic equivalence of the AST before and after transformations and the interaction of transformations when applied iteratively to a program.

We performed two types of experiments. The first kind performs unit testing for each implemented transformation and analysis (discussed in the next section). Since the goal of the back-end is to simplify the code we needed to evaluate the effects of the transformations. We designed a set of metrics that give insight to the complexity and comprehensibility of the code. The second set of experiments computes these metrics for a set of benchmarks. In Section 9.2 we discuss the metrics and benchmarks used in our experiments. Empirical data and its discussion can be found in Sections 9.4 and 9.5.

9.1 Unit Testing

As each transformation was implemented, we created test cases that checked that the transformation was sound. These stress cases check for bugs in the implementation and ensure

that the transformations result in the desired control flow. Since the transformations apply pattern matching techniques another very important set of tests were the cases where a pattern does not get matched. Hence by checking both cases: when the pattern should get matched and when it shouldn't we are sure that the transformations will not change the program behavior. Another advantage of using test cases is they can be used to reason about the control flow.

9.2 Complexity Metrics

We experimented with a wide variety of metrics and in this section we present those metrics that we found to be most useful for the purposes of evaluating the quality of code produced by decompilers.[1] We first present the simplest metrics for size and counting relevant constructs. One of the key differences among decompilers is their treatment of conditional expressions and hence we define a *conditional complexity* metric designed to expose those differences. Finally, a special problem introduced in decompilation and obfuscation is the naming of identifiers. Hence, we introduce an *identifier complexity* metric to measure the complexity of identifier names.

All of the metrics were computed using specialized traversals over the abstract syntax tree (AST) representation of Java source as produced by the polyglot-based SOOT front-end.

9.2.1 Program Size

A simple program size metric is useless in comparing two *different* programs other than to say one is larger than the other. However, this metric can be very useful in comparing two representations of the *same* program. Arguably, more verbose code is more complex and this metric is a good high-level measurement to see if decompilers produce unnecessarily verbose code and if obfuscators inserted useless code.

For our purposes, we define *program size* to be the number of nodes in program's AST

[1] The design and implementation of these metrics has been done jointly with Micheal Batchelder from the School of Computer Science, McGill University, who is currently working on the JBCO obfuscator

representation. Measuring size in this way discounts comments, spurious parentheses and any program formatting issues.

9.2.2 Number of Java Constructs

Another simple metric for the comprehensibility of a Java program is the frequency of different Java constructs in the code. Of course it is necessary to identify which constructs are strong indicators of complexity. After considering empirical results, we narrowed our attention to four categories:

- `If` and `If-Else` statements (Simple Conditionals)
- Abrupt control flow (`break` and `continue`)
- Labeled blocks
- Local variables

Simple conditionals help to indicate the amount of decision-making in a program. A more complex program will have more branching and therefore more `If` and `If-Else` statements.

Abrupt control flow directives are even more indicative of complex programming. It is argued that the use of these statements decreases the tractability of control flow and therefore increases code complexity.

Labeled blocks are compound statements which are explicitly labeled. While programmers will often section their code using blocks, the existence of a label suggests the block is used for controlling execution flow (through the use of a explicitly labeled `break` or `continue`). Other than exception handling, this is one of the most complex control flow mechanisms in Java.

Local variable counts can also indicate complexity. The more information one must consider when reading code the harder it is to understand. Programmers don't usually create unnecessary identifiers, but tools like decompilers and obfuscators often do.

9.2.3 Conditional Complexity

Boolean expressions which decide control flow in a program (*i.e.*, those deciding `If`, `For`, and `While` branching) play a particularly crucial role in analyzing code. Aside from boolean constants (`true` or `false`), the simplest conditional expressions consist of a unary boolean literal - a boolean variable. This is assigned a complexity weight of 1. However, conditional expressions can be aggregations or nestings of simpler expressions. A boolean literal can be reversed with the negation operator, ! or relational operators ($<$, $>$, $<=$, $>=$, $==$) can be used to compare expressions. We argue that these operators, while more complex than a single boolean, are still fairly easy to understand and therefore we give them a weight of 0.5. Expression aggregation using the && or || operators requires the reader of code to evaluate the meaning of two subexpressions and then to combine the two - arguably a more complex task - so we define the weight for these operators to be 1.

The complexity for each boolean expression in a program is simply the sum of all the weights described above. Taking the subtree that represents the expression, the leaves of the tree are boolean literals (increasing the complexity by 1 each) and every internal node is either an unary, relational, or binary operation (increasing the complexity by 0.5, 0.5, or 1, respectively).

Given this description, the expression `a<b && !done` would be assigned a complexity of 5. `a<b` refers to two variables (weight of 1 each) and the relational operator giving it a complexity of 2.5. `!done` is a boolean with a negation operator and is given 1.5. The aggregation (`&&`) adds another 1 to the overall complexity for a total of 5.

Average conditional complexity for a program is simply the average of the conditional complexities over all boolean expressions in the program.

9.2.4 Identifier Complexity

The name used for an identifier can provide valuable insight into the context in which the variable is used. This in turn can ease a programmer's task of understanding the code. Indeed, most obfuscators garble identifiers in a program. We compute the complexity of identifiers by calculating a sum of complexities for all identifiers where each is weighted by a relative importance. An identifier x has it's importance factor $I(x)$ defined as follows:

I(x) is 4 if x is a method name, 3 if it is a class identifier, 2 if it is a field, 1.5 if it is a formal and locals have 1 as the importance factor.

We argue that method names are particularly important for program understanding so we give them the highest importance value. Each identifier's complexity is computed as the sum of token and character complexities (described below) multiplied by their importance factor. Total identifier complexity is then calculated as a sum over all individual identifier complexities.

Token complexity is a measure of recognizable language. Alpha tokens are parsed and delimited by non-alphas and uppercase alphas. For example, `getASTNode` is split into `get`, `AST` and `Node`. Notice `ASTNode` is split into two tokens, the second one starting with a capital alpha). Similarly, `___Junk$$name` is broken into `Junk` and `name`. Tokens are then counted and the *token complexity* is defined as the ratio of total tokens to those found in a dictionary.[2] If the dictionary contains the tokens `get` and `Node` but not `AST` then token complexity for `getASTNode` will be 1.5.

Character complexity is a ratio of total characters to those classified as non-complex. Non-complex characters are those which are *not* part of a sequence of non-alphas of length greater than 1. The character complexity for the identifier `___Junk$$name`, for example, is 1.625 as there are five complex to 8 non-complex characters (_ , _ , _ , $, $ and J, u, n, k, n, a, m, e, respectively). Note that a sequence of non-alphas of length one is not considered as complex since it very likely exists as a word separator, as in `get_Socket`.

9.3 Benchmarks

The benchmarks have been culled from a graduate-level compiler optimizations course where students were required to develop interesting and computation-intensive programs for comparing the performance of various Java Virtual Machines. Each one was written in the Java source language and compiled with `javac`. The following is a brief description of each.

[2] The dictionary used in our experiments was a standard English language dictionary. However, one could use a special-purpose dictionary that also contained domain-specific identifiers.

Asac: is a multi-threaded sorter which compares the performance of the Bubble Sort, Selection Sort, and Quick Sort algorithms.

Chromo: implements a genetic algorithm, an optimization technique that uses randomization instead of a deterministic search strategy. It generates a random population of chromosomes. With mutations and crossovers it tries to achieve the best chromosome over successive generations.

Decode: implements an algorithm for decoding encrypted messages using Shamir's Secret Sharing scheme.

FFT: performs fast fourier transformations on complex double precision data.

Fractal: generates a tree-like (as in leaves) fractal image.

LU: implements Lower/Upper Triangular Decomposition for matrix factorization.

Matrix: performs the inversion function on matrices.

Probe: uses the Poisson distribution to compute a theoretical approximation to pi for a given alpha.

Sliding: solves the well-known Sliding Block Puzzle Problem.

Traffic: is an animation of a road intersection controlled by a traffic signal. It uses multithreading to simulate cars moving through the intersection.

Triphase: performs three separate numerically-intensive programs. The first is linpack linear system solver that performs heavy double precision floating-point arithmetic. The second is a heavily multithreaded matrix multiplication algorithm. The third is a multithreaded variant of the Sieve prime-finder algorithm.

The benchmarks we selected are not large (our size metric is shown in Figure 9.1), but are quite varied and exhibit many different properties and coding styles.[3]

[3]We would have liked to experiment with some larger benchmarks as well, but in order to do so in a rigorous manner all of the decompilers and obfuscators would have to work correctly on those benchmarks. This appears not to be the case. As the other tools mature and become more robust on larger applications, it will be possible to experiment with larger programs.

9.4 Evaluation of Decompiled Code

We discuss the results obtained from measuring the decompiled output of different decompilers. Each benchmark was decompiled using four different decompilers: the original Dava decompiler (henceforth referred to as Dava(Original)), the improved version of Dava (referred to as Dava(Improved)), Jad[Jad] and SourceAgain[Sou].

9.4.1 Program Size

Since each decompiler has its own source code formatting style, we normalized all output with a style formatter (JRefactory's JavaStyle [JRe]) in order to remove these differences. The formatter ensures that the AST contains the same number of AST nodes for the same constructs (an `If` block with one statement in its body is calculated the same whether brackets exist, distinguishing the block as a compound statement, or not). Figure 9.1 shows the number of nodes in the AST for all benchmarks. Traffic is largest with triphase, sliding, and chromo following it.

It is interesting to note that the output produced by different decompilers does change

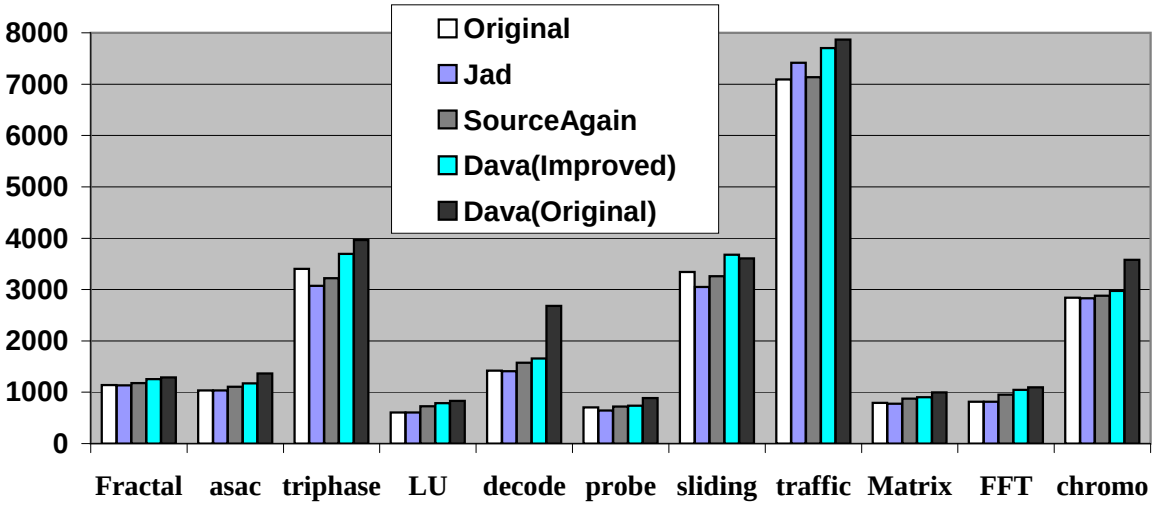


Figure 9.1: *Program size for decompiled code*

the size of the code. Dava(Original) *i.e.*, Dava without its back-end enabled produces the

largest size AST. However, once the back-end is enabled this AST decreases in size (mostly because of the removal of abrupt statements and labeled blocks and the aggregation of conditional statements using the boolean && and || operators). Usually the output produced by Jad and SourceAgain matches very closely to the original source code. This being an expected result since the decompilers use pattern matching to reverse the code generated by the compiler used.

9.4.2 Conditional Statements

Since Dava(Original) did not deal with short-circuit control flow created by && and || operators, it produces more `If` and `If-Else` statements. Dava(Improved) implements numerous aggregation transformations, greatly reducing the number of conditionals, as supported by the metrics in Figure 9.2 attests to this fact.[4]

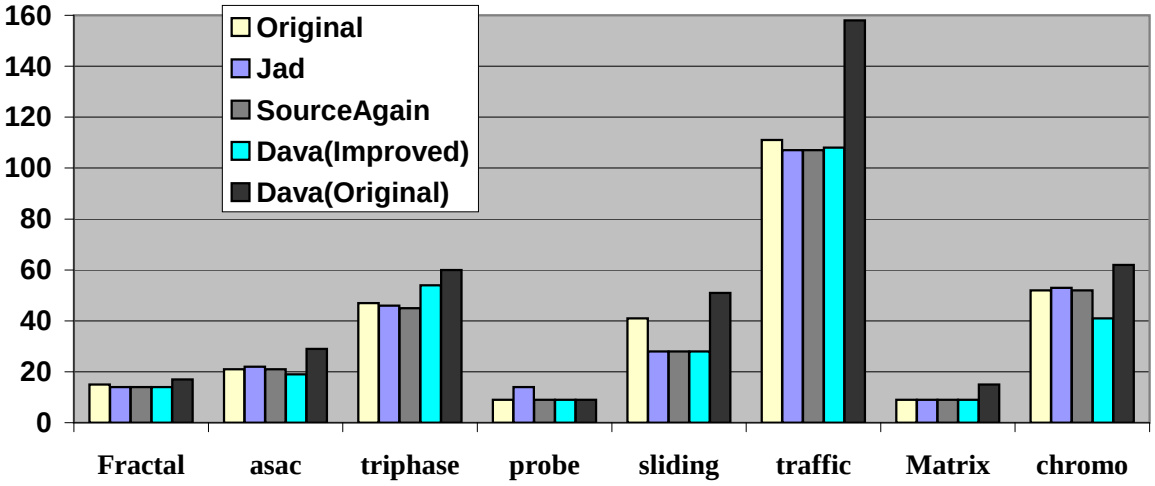


Figure 9.2: *Conditional statements for decompiled code*

The largest peaks for the number of conditionals are from Dava(Original). With Dava(Improved), however, there is a drastic drop in these constructs which, in most cases, matches that of the other decompilers. Interestingly, all decompiler output (except Dava(Original)) for the sliding benchmark contain fewer conditionals than the original

[4]Note that in this and subsequent graphs we do not show results for benchmarks for which the metrics are the same, or nearly the same, for all versions of the benchmark.

source. This would indicate that the benchmark's original code used very simple non-aggregated conditional statements and was perhaps written by a novice programmer. An examination of this benchmark proved this. Figure 9.3(a) shows a code snippet from the original source code of the sliding benchmark. Statement 3 in the code is an `Unconditional-While` loop and statement 4 and 5 are the exit condition for the loop. In the decompiled code produced by Dava we see that the `Unconditional-While` loop has been replaced by a conditional loop by pulling in the condition from statement 4 into the loop body (use of aggregation pattern discussed in Section 5.2.2). Another bad programming instance is detected at statements 9 and 10 of Figure 9.3(a) where the `If-Else` statement contains an empty **if** body. This has been converted by Dava to an `If` statement with the condition negated (Statements 8 and 9 in Figure 9.3(b)).

An interesting observation is that the general strategies in Dava(Improved) sometimes find more aggregation opportunities than Jad and SourceAgain (asac and chromo), and sometimes find fewer (triphase). This demonstrates that different decompilation strategies can impact the quality of the output.

9.4.3 Condition Complexity

Conditional complexity is a measure of the complexity of the boolean expressions within conditional constructs (`If`, `If-Else`, and loop constructs). Conditional complexity increases as boolean subexpressions are aggregated using the && or || operators. At the same time the use of negations (!) also increases conditional complexity. Figure 9.4 shows conditional complexity for the benchmarks.

For most benchmarks Jad and SourceAgain produce code with almost the same measure as the original. Small variations occur when a boolean flag is represented using the negated flag and vice versa.

An exception to this is the sliding benchmark. Here we see that all the decompilers increase the complexity by almost the same amount. This again strengthens our belief that the benchmark was written by a novice programmer who used simple non-aggregated boolean expressions. The decompilers merely detect the chance to aggregate the different conditions and in doing so increase the conditional complexity and reduce the number of

(a) Original Source code

```
public static int search(Problem p) throws Exception {
   nodes.add(new Node(p.getStartState()));
   while (true) {
     if (nodes.size() == 0)
        throw new Exception("No solution found!");
     <snip>
     for (i = 0; i < succ.size(); i++) {
        Node toInsert = (Node) succ.elementAt(i);
        if (FindCycle(toInsert, x, y)) ;
        else nodes.add(toInsert);
     }
   }
}
```

(b) Dava(Improved) output

```
public static int search(Problem p) throws Exception{
  nodes.add(new Node(p.getStartState()));
  while (nodes.size() != 0){
      n = Astar.removeBest();
      <snip>
      for (i = 0; i < succ.size(); i++){
         toInsert = (Node) succ.elementAt(i);
         if ( ! (Astar.FindCycle(toInsert, x, y)))
             nodes.add(toInsert);
      }
   }
   throw new Exception("No solution found!");
}
```

Figure 9.3: *Detecting simple non-aggregated conditional statements in original Source*

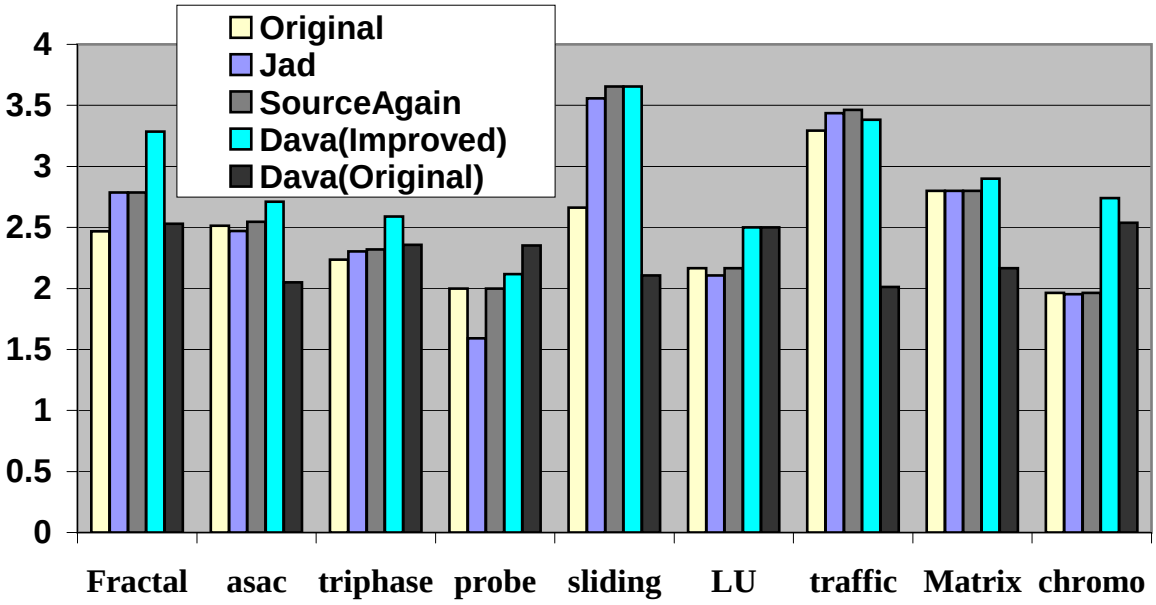


Figure 9.4: *Average Condition Complexity for decompiled code*

`If` and `If-Else` statements.

Comparing Dava(Improved) and Dava(Original) we see that apart from the probe benchmark there is a definite increase in conditional complexity implying the aggregation of conditions. When we investigated the probe code, we noticed that whereas Dava(Original) was creating conditions of the form "!flag" Dava(Improved) was able to switch the bodies to have conditions of the form "flag". Further, there was no chance of aggregation in the code. Thus, the removal of negation decreases the complexity and we see this in the complexity values for probe.

By examining the values for the original metrics, we see that a conditional complexity between 2 and 3 is normal. In the future, a metric-aware Dava could use its aggregation transformation sparingly in an attempt to maintain this level.

9.4.4 Abrupt Control Flow

Eliminating `Break` and `Continue` statements is one of the key transformations implemented in Dava(Improved). We argue that these abrupt control flow devices, of all Java

constructs, add the most complexity to source code because they represent disjoint execution flow. The more abrupt edges there are in a program, the less the code reads sequentially. This makes it difficult for a programmer because it increases the the number of scoping levels that must be kept track of, as well as the cohesion of disparate code chunks.

Out of all the benchmarks, sliding and traffic were the only ones which had a sizable number of `break` statements. All decompilers end up introducing some abrupt flow but this number is usually very low for javac-specific decompilers, Jad and SourceAgain, as seen in Figure 9.5. Again, this is due to the matching of code patterns to obtain concise output.

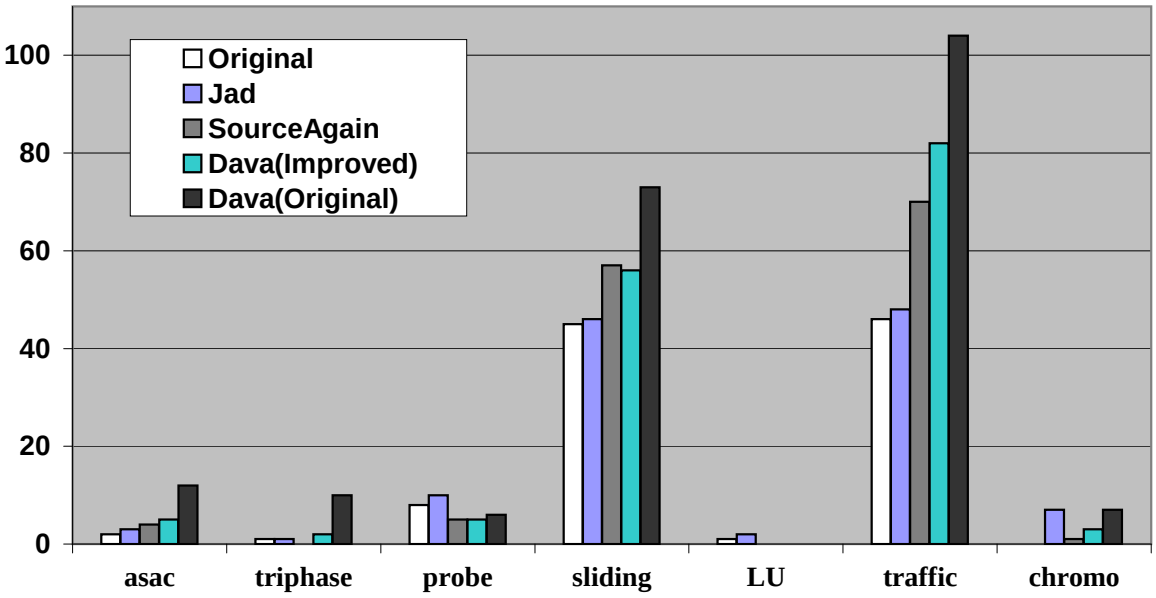


Figure 9.5: *Abrupt statements for decompiled code*

Dava(Original), on the other hand, suffers greatly by producing code with many complicated `break` statements nested within `Labeled-Block` constructs. This is because the low-level bytecode represents all of its control flow through only `If` and `goto` instructions; a naive decompiler will take the simplest route and transform these into abrupt `breaks`. The impact of more complex abrupt flow transformations, as implemented in Dava(Improved), can be seen in the reduction of abrupt statements for Dava(Improved) as compared to Dava(Original). In many cases Dava is able to produce fewer, if not the same, number of abrupt statements as Jad and SourceAgain. However, sliding and traffic

are two benchmarks which still show there is room for improvement. On inspection of the code it becomes obvious that these `break` statements can be removed by applying more generalized patterns on the AST. A few of these are discussed as future work.

An interesting anomaly is noticed in the metric values for the chromo and probe benchmarks. The abrupt statement counts for the output produced by Jad is higher than that produced by SourceAgain or Dava(Improved). On inspection it was noticed that Jad sometimes produces unnecessary `continue` statements. Figure 9.6(a) shows code produced by Jad. The `continue` statement can be avoided by negating the condition of the `If` statement (Statement 2 in Figure 9.6(a)) and adding Statement 4 as the new body of the `If` statement. This is exactly what SourceAgain and Dava(Improved) do, as shown in Figure 9.6(b).

(a) Jad output

```
1 for(int j1 = 0; j1 < i; j1++){
2     if(d < a1[j1].cfitnessGet() || d >= a1[j1 + 1].cfitnessGet())
3           continue;
4     a1[j1 + 1].copyChromosome(a2[i1]);
5 }
```

(b) Dava(Improved) output

```
1 for (i2 = 0; i2 < i0; i2++){
2     if (d1 - r1[i2].cfitnessGet() >= 0 && d1 - r1[i2 + 1].cfitnessGet() < 0){
3           r1[i2 + 1].copyChromosome(r2[i7]);
4     }
5 }
```

Figure 9.6: *Unnecessary `continue` statements produced by Jad*

9.4.5 Labeled Blocks

Directly related to abrupt statements are the number of labeled blocks present in decompiled code. Labeled blocks are especially bad programming practice and, in fact, they exacerbate the previous problems with abrupt control flow by allowing more disjoint execution jumps than available with unlabeled `break` statements. Unsurprisingly, no labeled blocks appear in the original source of any of the benchmarks. Jad and SourceAgain are able to maintain this minimum. The general restructuring algorithm in Dava(Original), on the other hand, produces a high number (Figure 9.7). Figure 9.7 also shows that 75% of these labeled blocks, introduced by Dava(Original), are removed by the pattern-matching based transformations implemented in Dava(Improved).

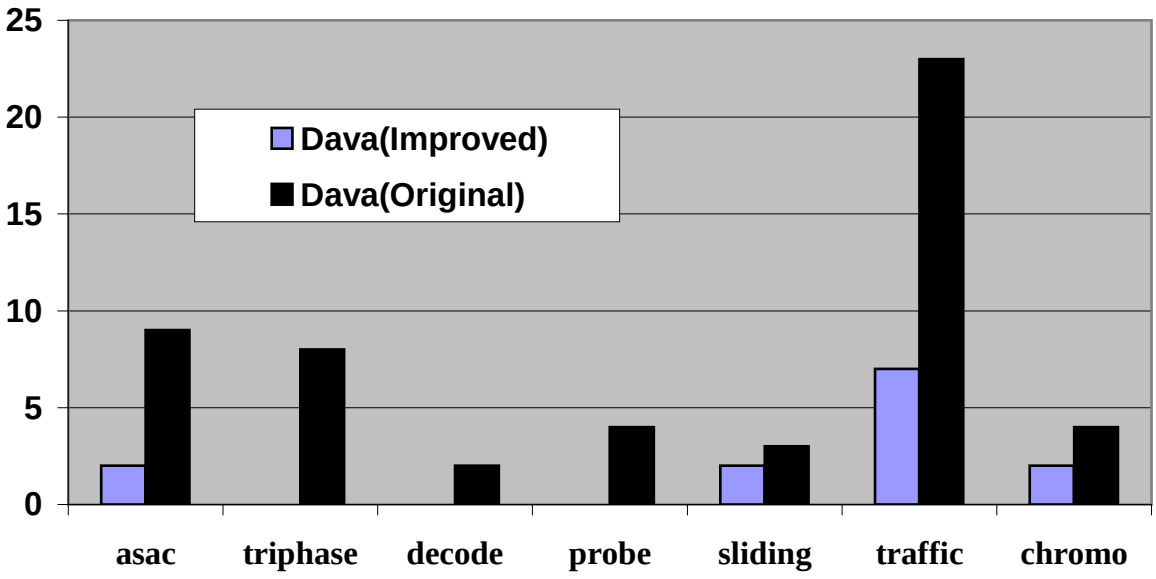


Figure 9.7: *Labeled Blocks for decompiled code*

9.4.6 Local Variables

Dava(Original) produces many local variables in its output. This is because Dava takes its input from `grimp` which has been computed from the low-level Soot IR which uses many local variables in order to get simple and precise compiler analyses.

Although local variable webs are collapsed while creating `grimp` the reduction in the number of locals is not as much as one would like. With Dava(Improved), copy elimination (Section 7.2.1) and constant substitution (Section 7.3) considerably reduce the use of intermediate local variables. Figure 9.8 shows the number of local variables for some of the benchmarks. Jad and SourceAgain output is, again, very close to the original for this metric.

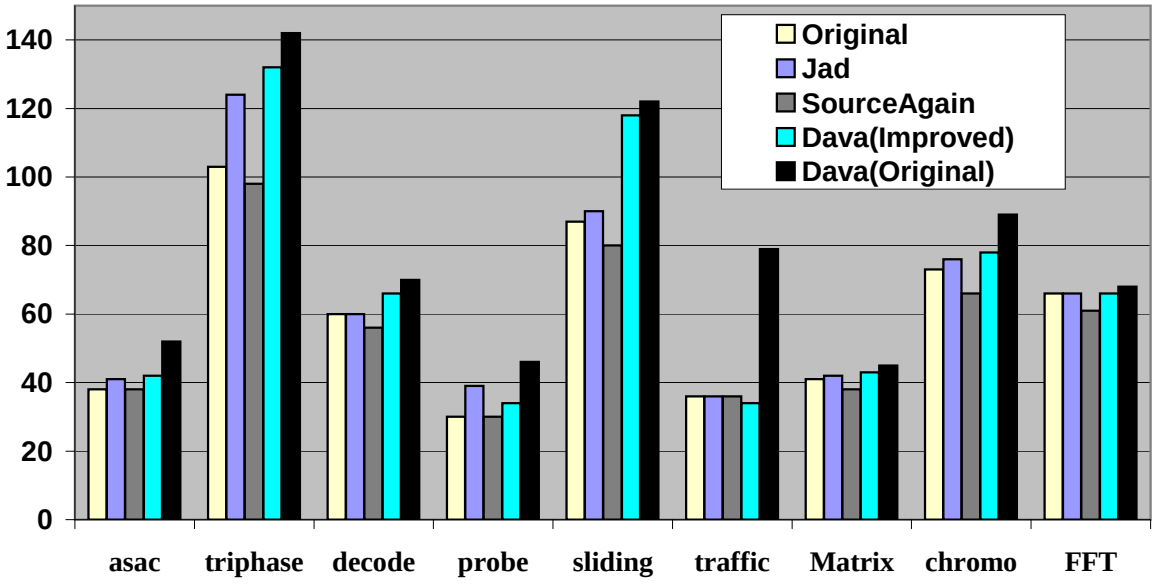


Figure 9.8: *Number of Locals for decompiled code*

An exception to this is triphase where we see an abnormally high number of local variables for Jad and Dava. Inspection of the decompiled code produced by Jad for triphase shows that it is unable to handle aggregated floating point and double precious calculations. These are broken down into 3-address statements where each statement introduces a new local variable. On inspection of code produced by Dava(Improved) we noticed that the increase in number of locals was mainly due to the presence of shortcut `If` statements in the original code. An example of this is shown in Figure 9.9(a). Whereas SourceAgain and Jad are both able to produce this shortcut construct Dava fails to detect the pattern and produces output shown in Figure 9.9(b). In the triphase benchmark the shortcut `If` statement occurs numerous times and this explains the higher number of local variables in Dava.

(a) Original source code

```
final double abs(double d) {
     return (d >= 0) ? d : -d;
}
```

(b) Dava(Improved) output

```
final double abs(double d0){
   double $d1;
   if (d0 - 0.0 < 0)
      $d1 = (- (d0));
   else
      $d1 = d0;
   return $d1;
}
```

Figure 9.9: *Reason for an increase in local variable count in Dava*

Dava(Improved) shows decent amount of improvement over Dava(Original) (particularly for traffic). The difference of local variables for Dava(Improved) with Original source code is now with an acceptable range in most case.

9.4.7 Loop Count

Table 9.1 shows the breakdown of different loops within the decompiled outputs of the different decompilers as compared to the original source code. Both Jad and SourceAgain aggressively create `For` loops which we think is a good feature to have since `For` loops are inherently easier to understand than their `While` counterparts. Previously, Dava was unable to generate `For` loops and represented all loops using one of the three flavors of `While` loops (`While`, `Do-While` or `Unconditional-While`). With the implementation of the `For` loop construction transformation (Section 7.1.1) Dava is now able to generate `For` loops. However, in Dava we restrict the conversion of a `While` loop to a `For` loop to cases where all the four components of the `For` loop can be determined (Section 7.1.1).

The sliding benchmark shows some interesting results. The original source code contained an `Unconditional-While` loop which has been converted to a `While` loop in Dava(Improved). This was previously illustrated in Figure 9.3 where we see that the con-

	Do	For	While	UnConditional
triphase(Original)	0	43	3	0
triphase(Jad)	1	45	0	0
triphase(SourceAgain)	0	44	2	0
triphase(Dava-Original)	0	0	46	0
triphase(Dava-Improved)	0	45	1	0
decode(Original)	0	29	1	0
decode(Jad)	0	30	0	0
decode(SourceAgain)	0	29	1	0
decode(Dava-Original)	0	0	30	0
decode(Dava-Improved)	0	29	1	0
sliding(Original)	0	21	2	1
sliding(Jad)	1	22	1	0
sliding(SourceAgain)	0	21	3	0
sliding(Dava-Original)	0	0	24	0
sliding(Dava-Improved)	0	22	2	0
Matrix(Original)	0	21	0	0
Matrix(Jad)	0	21	0	0
Matrix(SourceAgain)	0	20	1	0
Matrix(Dava-Original)	0	0	21	0
Matrix(Dava-Improved)	0	20	1	0

Table 9.1: *Breakdown of Loops for decompiled code*

dition of a nested `If` statement is pulled into the `While` loop as it's condition.

Another interesting thing to note in the results for sliding are that even after the conversion of the `Unconditional-While` loop to a `While` loop both the original code and Dava(Improved) have the same number of `While` loops. The reason being that one of the `While` loops in the original code can be better represented as a `For` loop. This conversion is illustrated in Figure 9.10. Figure 9.10(a) shows the original code snippet. Since this `While` loop contains a condition which checks on a pointer's `null` value, which is consistently

updated within the loop body, Dava converts the `While` loop into a `For` as illustrated in Figure 9.10(b).

(a) Original source code

```
1 ptr = n;
2 while ((ptr.getParent()) != null) {
3    store.addElement(ptr.getblockType());
4    store.addElement(ptr.getopcode());
5    ptr = ptr.getParent();
6 }
```

(b) Dava(Improved) output

```
1 for (r4 = r0; r4.getParent() != null; r4 = r4.getParent()){
2    r3.addElement(r4.getblockType());
3    r3.addElement(r4.getopcode());
4 }
```

Figure 9.10: *Converting a `While` loop to a `For` loop*

9.4.8 Overall Complexity

In order to provide one summary metric, we experimented with a variety of composite metrics. We found a good overall complexity metric that is defined by first expressing each component metric as a normalized value with respect to the value for the original Java benchmark, and then combining the normalized values, each component multiplied by a constant representing that metric's importance. The sum of the constants is 1, so that when comparing the original javac source to itself will always result in an overall metric of 1.

For example, for the size component we compute the normalized value by (size of decompiled benchmark)/(size of original benchmark) and we multiply this normalized value

by 0.2. Figure 9.11 gives the result using 0.2 * size + 0.2 * if_count + 0.2 * cond_complexity + 0.1 * num_abrupt + 0.1 * num_labeled + 0.2 * num_locals, where each component of this metric corresponds to normalized values of the metrics as presented in Section 9.2.

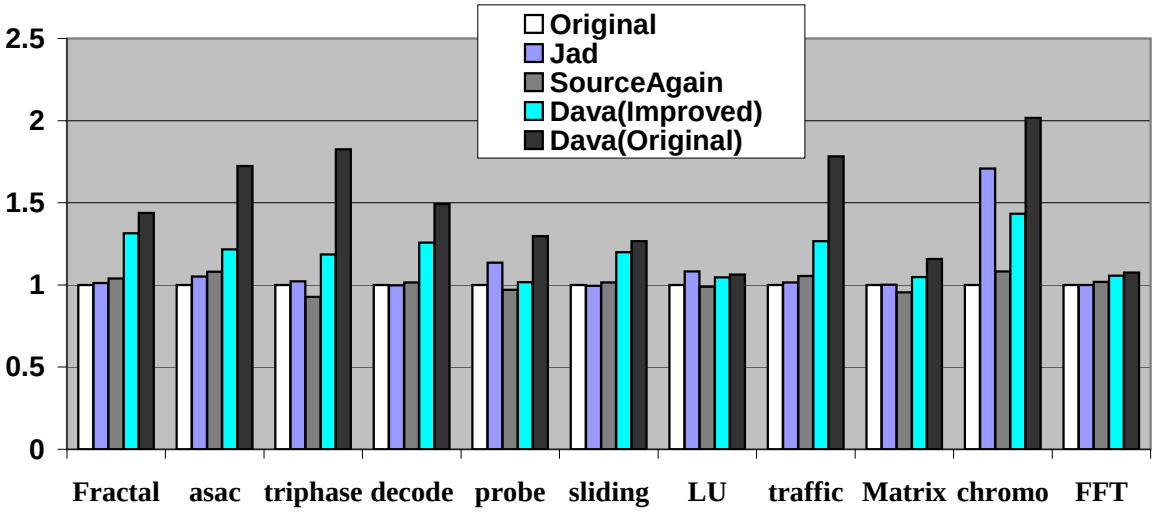


Figure 9.11: *Overall complexity for decompiled code*

Using this overall metric we can see that Jad and SourceAgain produce decompiled code that is close to the original code (remember that these benchmarks have not been obfuscated and thus javac-specific decompilers work well for them). We can also observe that Dava(Original) does in fact produce (ugly) code that is not as similar to the original code, but that the additional transformations implemented in Dava(Improved) do improve upon this substantially.

9.5 Evaluation of Obfuscated Code

The experiments in this section were performed as follows. We created our baseline by first compiling the application using an ordinary `javac` compiler to produce the class files and then decompiled those class files with our Dava decompiler, with all of the advanced transformations turned on. This option is labeled Dava(Improved) in subsequent figures. Notice that in decompiling obfuscated code we only use Dava and not Jad or SourceAgain.

Both Jad and SourceAgain are not able to decompile much of the obfuscated code. Dava on the other hand is robust enough to be able to decompile code after first- and second-generation obfuscations. In order to be able to obtain metrics we need compilable Java source and hence our choice of decompiler.

To create the obfuscated versions of the source code we first applied the obfuscators (Klassmaster and JBCO) to the class files to produce obfuscated class files. We then decompiled the obfuscated class files using Dava. We used Dava in two configurations, the *Original* one, and the *Improved* one where all simplifications are applied. In the subsequent figures JBCO(Improved) refers to the case where we obfuscated with JBCO and then decompiled with Dava(Improved) and JBCO(Original) refers to the case where we obfuscated with JBCO and then decompiled with Dava(Original). Similarly, we created two versions for the Klassmaster obfuscator.

By comparing the Dava(Improved) versions with JBCO(Improved) and Klassmaster(Improved) one can observe the impact that the two obfuscators had on the metrics. By comparing the Klassmaster(Improved) to Klassmaster(Original), and similarly comparing JBCO(Improved) to JBCO(Original), we can observe the impact of the advanced Dava simplifications in undoing some of the obfuscations introduced by the obfuscators. These include some identifier renaming optimizations, control-flow simplifications, copy elimination and advanced dead-code elimination.

Although we computed all the metrics for both obfuscators, we only show results for Klassmaster in many of the figures. This is because JBCO has no effect on some of the metrics since we enable only two obfuscations: renaming identifiers and moving library calls into new methods with obfuscated names.

9.5.1 Benchmark Size

Figure 9.12 shows the program size metric. It is clear that both JBCO and Klassmaster increase the size in all cases. Comparing the two obfuscators we see that the size increase is greater for Klassmaster. This is expected because Klassmaster adds dead code guarded by opaque predicates which can therefore not be removed by the static analyses performed by Dava. JBCO size increases are due to the addition of methods which are used to in-

voke library calls through an extra level of indirection. Therefore, the difference between the unobfuscated Dava(Improved) case with the JBCO(Improved) case is directly proportional to the number of unique library methods called in the program. A smart decompiler could apply a refactoring algorithm to overcome this obfuscation through re-inlining these unneeded indirections.

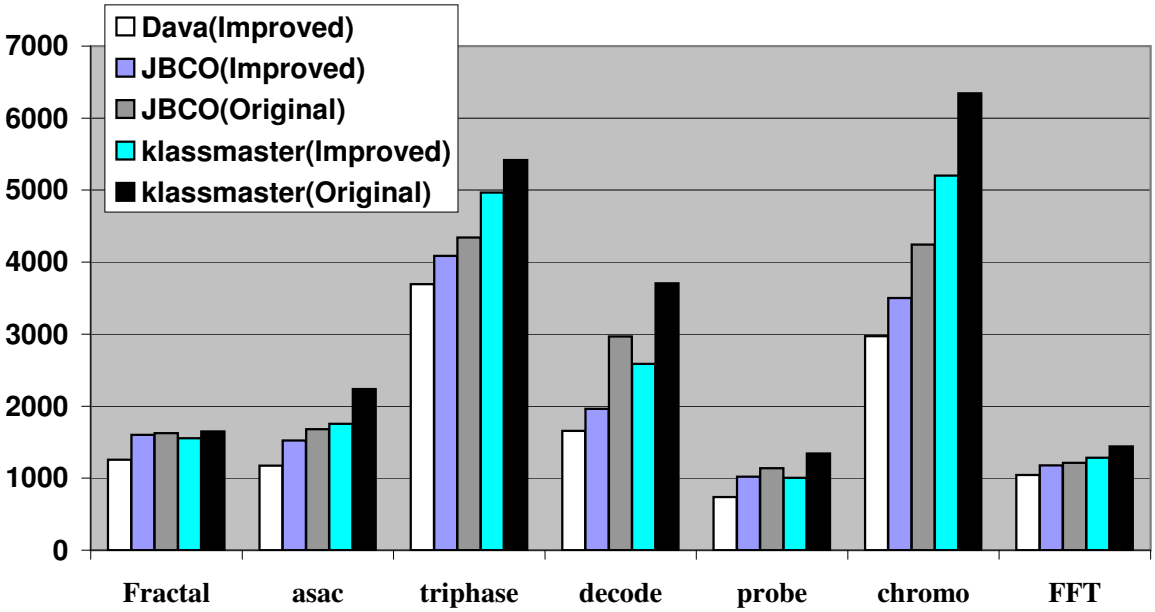


Figure 9.12: *Program size for obfuscated code*

Also interesting is the difference between Klassmaster with and without Dava's advanced simplification analyses, Klassmaster(Original) versus Klassmaster(Improved). This difference is most obvious for the decode and chromo benchmarks. In these cases the Dava dead code elimination removes a large amount of code introduced by Klassmaster. Nevertheless, not all dead code is removed because much of it is guarded by opaque predicates. Dava is unable to statically detect the values of these predicates and hence the code remains. A much more powerful context-sensitive flow analysis would be required to remove the remaining dead code.

9.5.2 Conditional Statements

Figure 9.13 demonstrates a large increase in the number of conditional statements after obfuscation by Klassmaster. This is consistent with Klassmaster's technique of introducing redundant or dead code enclosed by simple `If` statements. Dava attempts to aggregate many of the conditionals and can sometimes remove some redundancies, as illustrated by the difference between Klassmaster(Original) and Klassmaster(Improved). However, a large number of these conditions still remain.

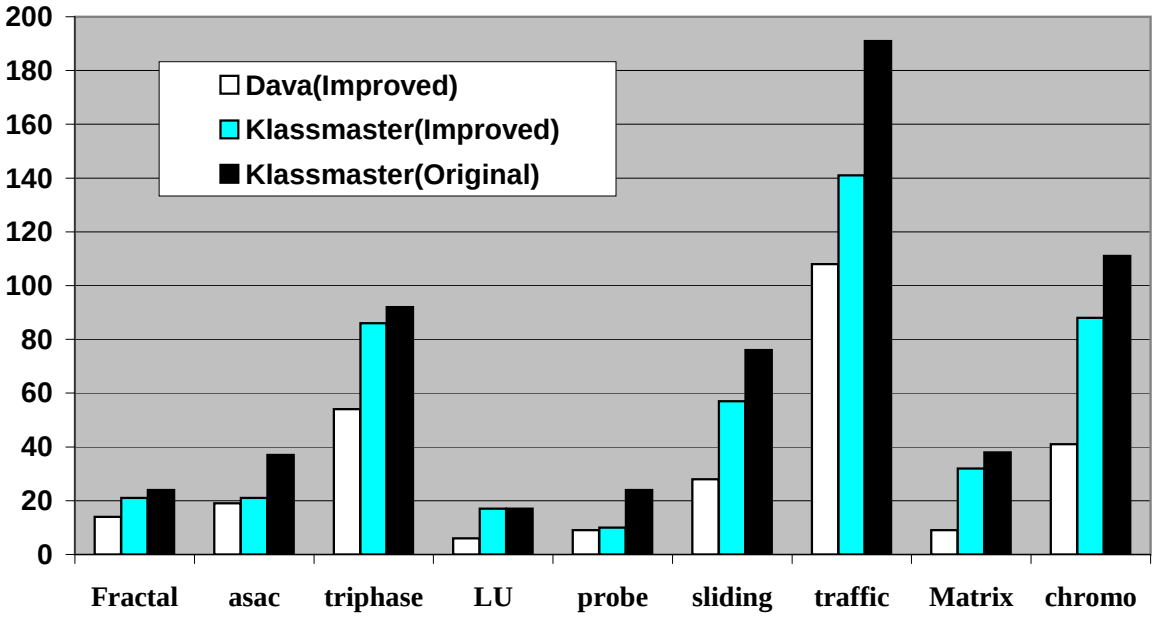


Figure 9.13: *Simple conditional statement count for obfuscated code*

9.5.3 Conditional Complexity

Conditional complexity is shown in Figure 9.14. Here, the decrease in complexity is mainly due to the fact that Klassmaster introduces its own conditional constructs which are simple un-aggregated boolean expressions. Hence, although the number of conditional constructs increases, the average conditional complexity decreases. An additional possible reason for the drop in complexity is that the original bytecode is intermixed with obfuscation code.

This inhibits the pattern-based simplifications and therefore results in fewer conditional aggregations. The increase seen in Klassmaster(Improved) versus Klassmaster(Original) is due to the aggregation of conditions. Some benchmarks show a decrease which most likely occurs due to removal of dead code which included complex conditionals.

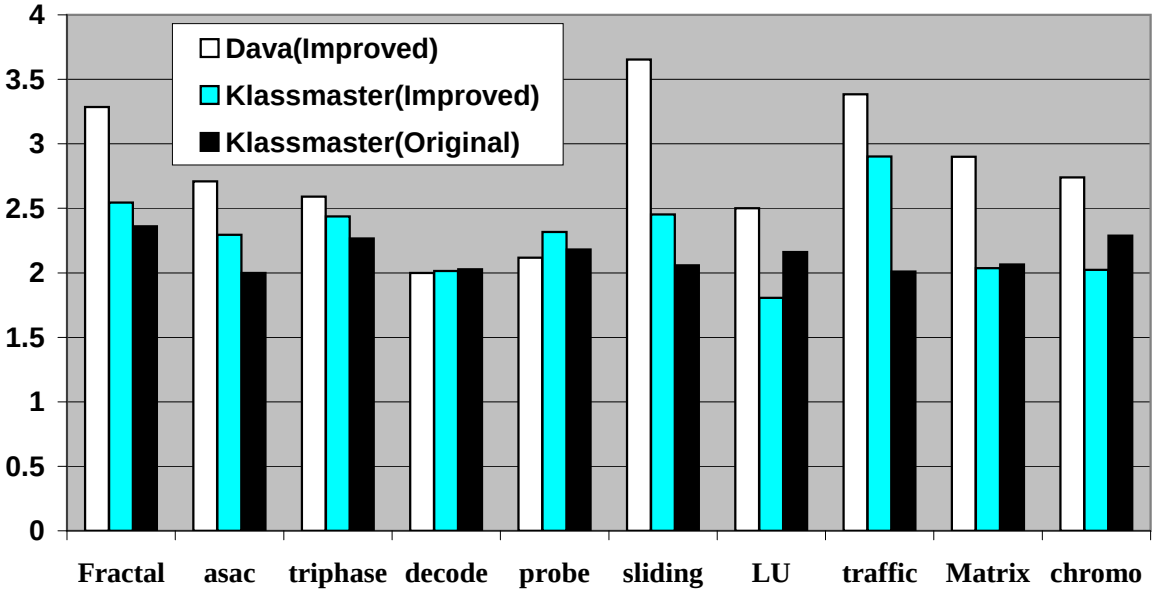


Figure 9.14: *Average conditional complexity for obfuscated code*

9.5.4 Abrupt Control Flow

The count of abrupt statements (`break` and `continue`) for the obfuscated code as compared to the un-obfuscated code is shown in Figure 9.15. We can see a marked increase in abrupt statements (particularly in triphase, decode and chromo).

The abrupt metric is particularly useful in identifying obfuscated code. Abrupt edges in the flow graph of a program are a direct result of control-flow obfuscation techniques and it clearly worsens the readability. As stated earlier, a programmer has a lot to keep track of when trying to follow abrupt control, especially when execution jumps directly out of multiple nesting levels. Thus, programmers tend to make sparse use of complex abrupt control-flow, whereas obfuscators intentionally add them in to complicate the control flow.

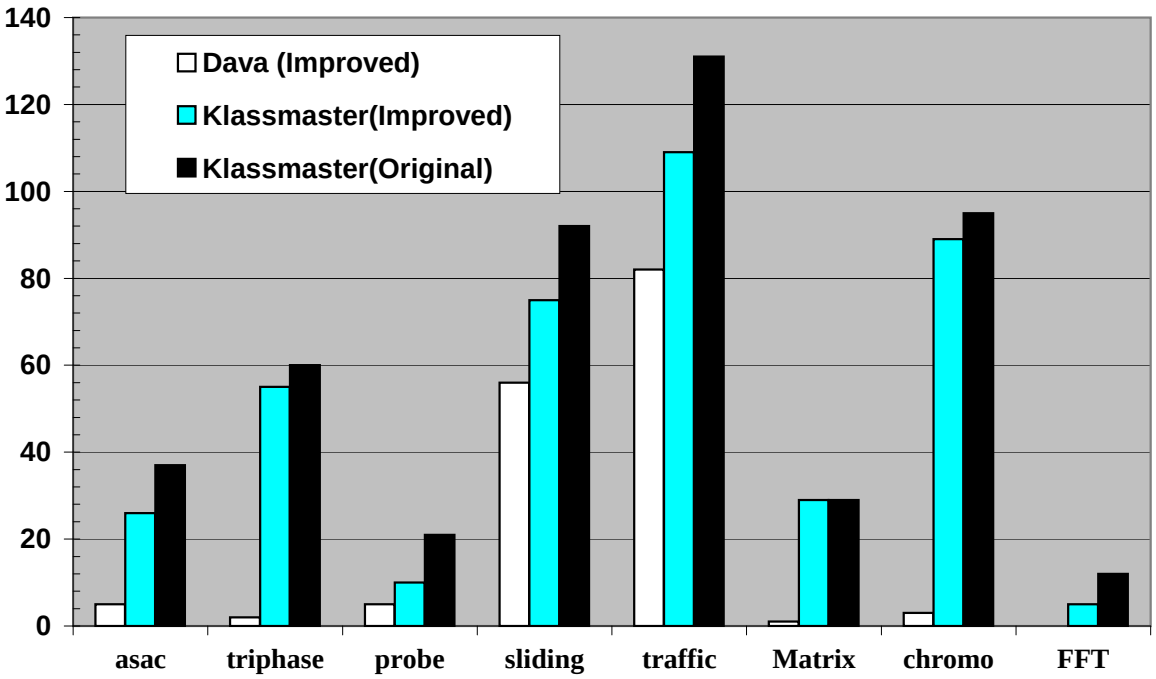


Figure 9.15: *Abrupt control flow count for obfuscated code*

It is interesting to note that javac-specific decompilers such as Jad and SourceAgain often fail to decompile such code because the control-flow in the class files does not correspond to any known structured Java control flow pattern. Dava succeeds in decompiling and reducing the number of abrupt control flow statements due to its use use of graph-based restructurings.

As demonstrated by comparing Klassmaster(Original) to Klassmaster(Improved), the Dava simplifications are able to restructure some of the code to reduce abrupt control flow in many of the benchmarks, but not all cases of abrupt control-flow can be removed. We suggest some more transformations in our future work to further decrease the number of abrupt statements, but it seems unlikely that all abrupt flow introduced by obfuscation could be eliminated.

9.5.5 Labeled Blocks

Labeled blocks are shown in Figure 9.16, correlating closely with the number of abrupt statements. The Klassmaster(Original) case has a large number of labels but Klassmaster(Improved) shows that Dava's simplifications can reduce these to a more acceptable level. For some benchmarks (FFT and probe) all labeled blocks can be removed. Over the whole benchmark suite 65% of the labeled blocks are removed. With the addition of further transformations discussed in the future work section it seems likely that even more labeled blocks can be removed from the code.

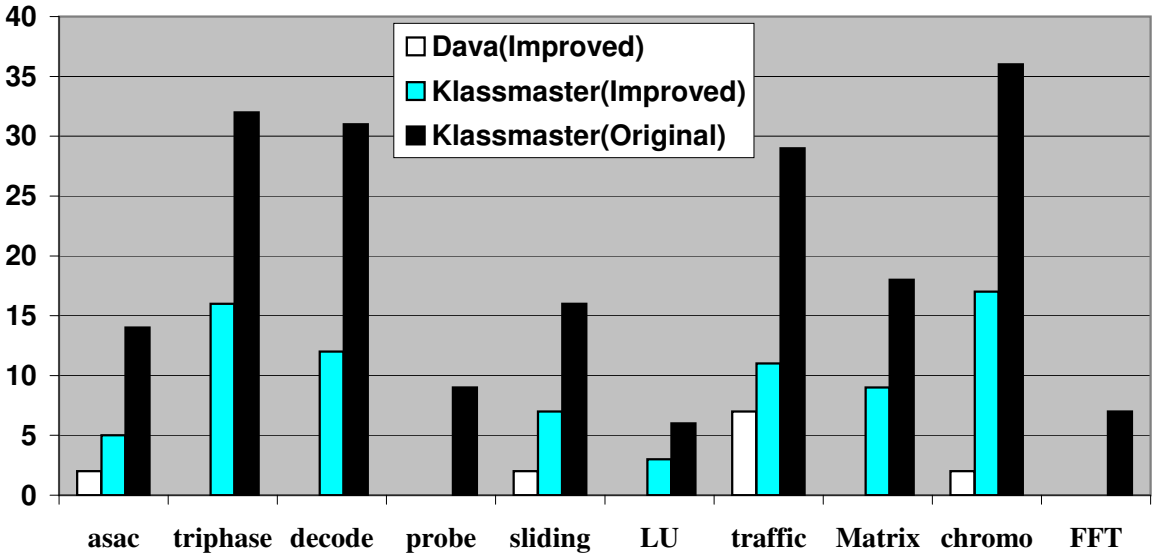


Figure 9.16: *Labeled block count for obfuscated code*

9.5.6 Identifier Complexity

Identifier obfuscation is a very important metric for evaluating obfuscators. Nearly all obfuscators perform identifier obfuscation and it is perhaps the only technique that is truly irreversible [BGI+01]. Figure 9.17 shows that JBCO performs good identifier obfuscation based on our metric. Klassmaster also does well, though a difference between the Klassmaster(Original) and Klassmaster(Improved) values can be seen due to a basic local

variable renaming algorithm implemented in Dava. Also, removal of dead code reduces local variables, some of which have complex names, hence decreasing the complexity.

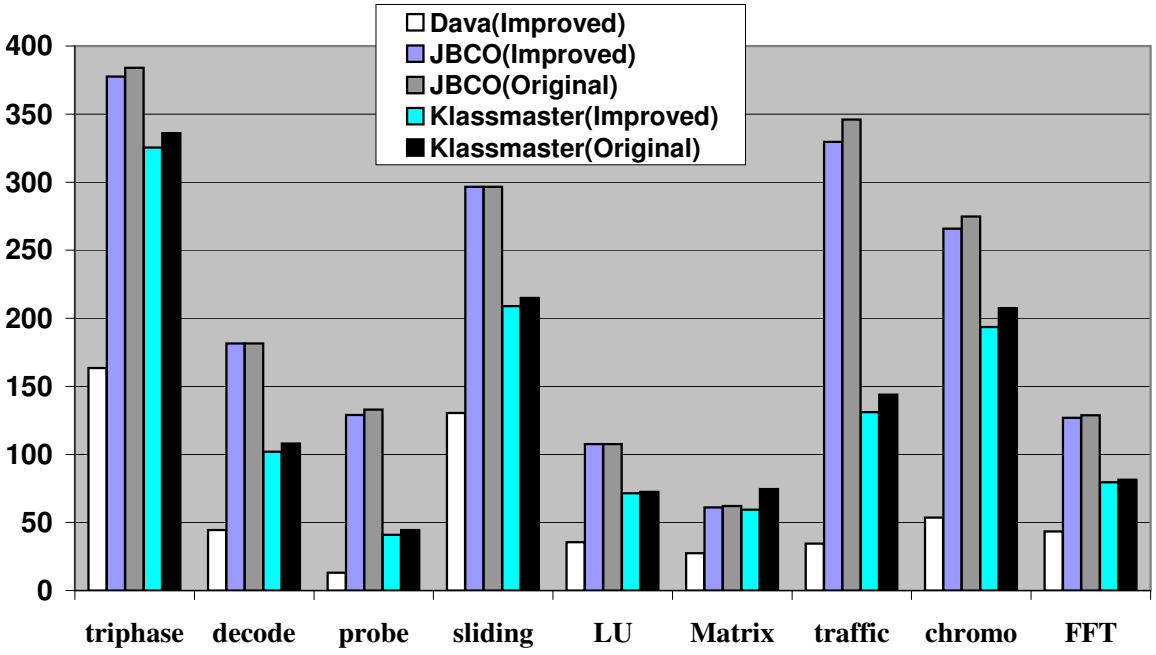


Figure 9.17: *Identifier complexity for obfuscated code*

9.5.7 Overall Complexity

Figure 9.18 reports the overall complexity metric. Note that this metric does not include identifier complexity, so one should really consider both the identifier complexity presented in figure 9.17 and the overall metric in figure 9.18 which summarizes control-flow like obfuscations, when considering the effect of obfuscators.

Considering these two figures we can see that, as expected, the effect of JBCO[5] is mostly on identifier obfuscation, whereas Klassmaster shows significant impacts on the structure of the code. It is also interesting to note that the Klassmaster(Improved) is closer

[5]In these experiments we used a preliminary version of JBCO, the final version of JBCO will support many more control flow obfuscations

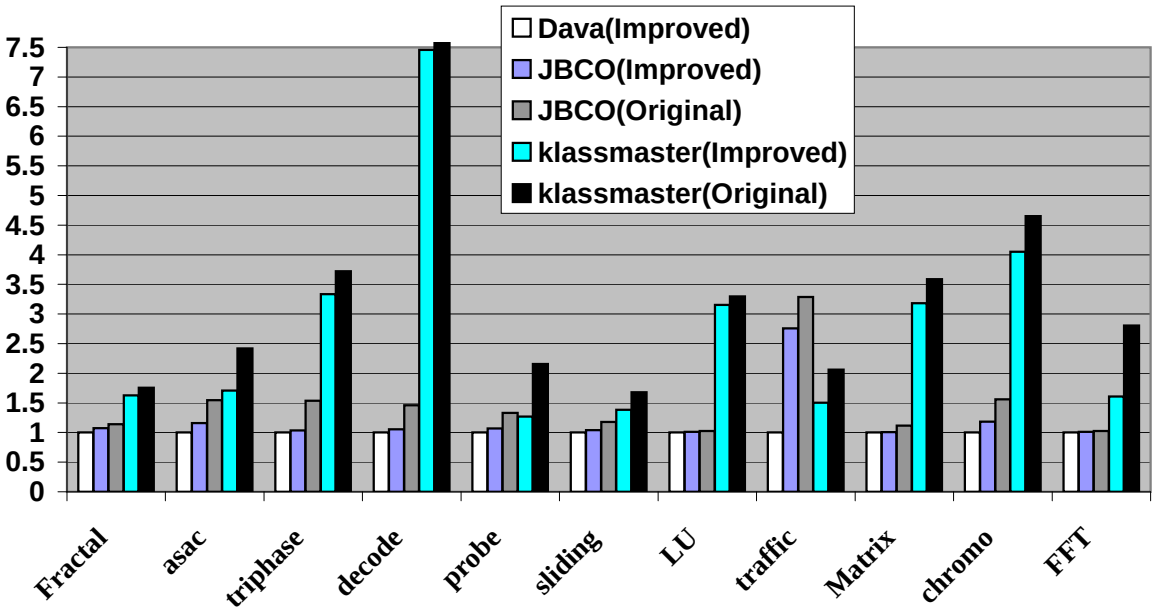


Figure 9.18: *Overall complexity for obfuscated code*

to the unobfuscated code than Klassmaster(Original), indicating that the advanced transformations in Dava do help to clean up the code.

Chapter 10
Related Work

To the best of our knowledge Dava is the only available tool-independent decompiler for Java. It is therefore difficult to compare methodologies used in Dava to other decompilers since the issues encountered for Dava are more complex than the simple reversing of code generation carried out by other decompilers.

10.1 Decompilers

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

SourceAgain is a commercial decompiler with an online version available to test its capabilities. The decompiler creates a flow graph representation from which it detects Java constructs. Due to the use of a flow graph representation it does a better job at decompilation than Jad. Although SourceAgain claims to be able to decompile obfuscated code our tests have shown that it is only able to handle name obfuscation(by converting these to indexed names) and fails when control flow obfuscation has been carried out.

10.2 Obfuscators

To test Dava's capabilities in decompiling and simplifying obfuscated code we used the Zelix Klassmaster [Klaa] obfuscator. Although Java obfuscation has become popular in recent years, both in academic and commercial communities, there aren't many obfuscators which do more than name obfuscation. Zelix Klassmaster stands out since it applies complicated control flow obfuscations by adding predicates guarding "presumably" undecompilable code. The Klassmaster documentation states:

> *"The obfuscator makes slight changes to the bytecode that obscures the control flow without changing what the code does at runtime. Typically, selection (e.g. if...else...) and looping constructs (e.g. while and for loops) are changed so that they no longer have a direct Java source code equivalent"* [Klab].

Our tests with the obfuscator indicate that the changes made are not "slight". Large chunks of code, which includes loops, are added to confuse the decompilers (Section 9.2). This creates convoluted code but at the expense of a slow down in the application runtime. Since one key selling point of Dava has always been its general applicability to verifiable bytecode, in most of our test cases, and all the benchmarks selected, Dava was able to correctly decompile the obfuscated code. KlassMaster claims to be the only "Second generation" Java obfuscator, a term coined for obfuscators performing strong control flow obfuscation. Our experiments with obfuscated code using different obfuscators attest to this claim. Zelix Klassmaster does reflect the latest technology, available for non-academic use, in the field of Java bytecode obfuscation.

The second obfuscator used was JBCO (Java Bytecode Obfuscator) which is still under development at the Sable Research Group at McGill University. Using the Soot [Soo] Java bytecode analysis framework, the same framework used by Dava, this obfuscator promises to be a top-notch Java bytecode obfuscator. JBCO's philosophy is to introduce the least, if any, amount of dead code and to incur minimum runtime slowdowns. JBCO's proposed transformations take into account the minute details of the Java language specification in order to exploit little-known options in bytecode representation. Although bytecode is relatively high level there is still a large gap between Java bytecode and Java source. Utilizing

the additional expressiveness of the bytecode, transformations are proposed which will have no, or very complicated, Java source code equivalent.

10.3 Visitor Design Pattern

The inspiration for the extended version of the visitor design pattern, now implemented for Dava's AST, was taken from Sablecc [GH98]. SableCC generates compilers (and interpreters) in the Java programming language from a given specifications grammar. The key features of SableCC include the use of object-oriented techniques to automatically build a strictly-typed abstract syntax tree and the generation of tree-walker classes for the generated AST. It is this implementation of the traversal routines, enabling the implementation of actions on the nodes of the AST using inheritance, that we have borrowed for use within Dava.

10.4 Structure-Based Flow Analysis

As the analyses for the decompiler are performed on the AST it is best to use a syntax-directed method of data flow analysis such as structural analysis. Structural Flow analysis initially presented by Sharir[Sha80] is ideal for data-flow analysis using a structured representation of the program. The advantage of using this technique is that it gives, for each type of high level control-flow construct in the language, a set of formulas that perform data flow analysis. This technique has been successfully used in creating an optimizing compiler which uses a hierarchy of structured intermediate representations [HDE+93]. Work done by Emami et. al. [Ema93] for gathering alias and points-to-analysis information for the McCAT C compiler matches very closely to what was required for Dava. Dava's flow analysis framework is an implementation of the same approach utilized in McCAT along with handling of complexities introduced by Java.

10.5 Complexity Metrics

There has been much research into software complexity and many metrics have been proposed and embraced by the software engineering community throughout the years. Classic examples are McCabe's cyclomatic number [McC76], and Halstead's programming effort measures [Hal77]. More recent efforts have been geared towards quality analysis for large-scale software projects and processes [LSP05, Con04].

These complexity measures are interested in measuring effectiveness, code reliability, programming effort, and clarity (or cognitive expressibility / representability) [Tai84]. What we are interested in within this research is this specific idea of cognitive expressibility. When a decompiler sets out to recover the higher-level source code of a binary program it is effectively attempting to recover a cognitive representation - a human-readable (or at least programmer-readable) version of the program that is semantically equivalent to the binary. Likewise, when an obfuscator sets out to garble a program it is attempting to decrease the cognitive representability of the program by adding complexity of some kind.

Because the quality of the cognitive representation is our key interest, some well-developed metrics in the literature are somewhat useless here. McCabe's Cyclomatic number, for example, shows the complexity of the control flow through a piece of code. It is the number of linearly independent paths through a program. However, if a program segment S is compiled into a binary B and then decompiled into a source code segment S' then S and S' will have the same cyclomatic number regardless of how the decompiler chooses to represent the loops and other branching instructions in the program. Therefore the metric shows us nothing of the differences between the cognitive representation of S and S'.

Similarly, Halstead's metrics are not all suitable for our case. They are often used during code development in large projects in order to track complexity trends. A spike in Halstead metrics can signify a highly error-prone module, for example. However, this is not our concern. We wish to use metrics to compare two high-level representations of a program, both with the same semantics. Halstead's metrics do not lend themselves well to this problem.

Program volume, for example, is a measure of the minimum number of bits required for coding a program. In the case of Java, non-local variables (either class fields or statics) and

method names are preserved in the compiled bytecode. A common Java obfuscation technique is to rename these identifiers, often with shorter and more incomprehensible names. This effectively reduces the program volume but also reduces the ability of a decompiler to recover the full cognitive representation of the original program.

Indeed, many metrics are designed to compare large software projects in a very abstract way in order to predict maintainability, reliability and/or programming effort. Most of these are not useful to the particular problem at hand.

However, some of the criticism that Halstead's measures have seen over the years - specifically the argument that they are a bad measure because they consider lexical and textual complexity rather than the structural complexity of a program [HF82] - is a key ingredient to our own proposed metrics. The high-level measures of lexical and textual structure, and complexity are in fact exactly what we wish to measure, along with control flow complexity.

We are much more interested in the high-level human-readable source code representation of the program's methods. This makes the approach in [RCC91] a good starting point as they measure such intricacies as identifier length, nesting depth, and decision node complexity.

Chapter 11
Future Work and Conclusions

11.1 Future Work

Although we have improved the output produced by Dava there are clear indications of areas where work should be carried out.

11.1.1 Abstract Syntax Tree Expansion

Currently Dava works on a per-method basis. Each method is separately decompiled and an AST, with an ASTMethodeNode as the root of the tree, is created. Although per-method decompilation works well for general decompilation, a class-based decompilation can provide additional avenues for analyses. It would be useful to modify the abstract syntax tree representation within Dava to handle per-class, instead of per-method decompilation. This can be achieved by the creation of an ASTClassNode data structure which could then hold all methods and fields of the class. This would help streamline and modularize some of the interprocedural analyses implemented as part of this thesis. The biggest advantage, however, would be the ability to retrieve and produce inner classes within the decompiler output. Also handling of field-aware analyses would become much easier if the fields were represented as elements of the abstract syntax tree.

11.1.2 Transformations

More aggressive `Labeled-Block` removal transformations are also needed. Currently the transformations apply to small patterns. Larger, more general patterns can and should be implemented to remove the complexity introduced by `Labeled-Block` constructs. As seen in the results section, although the output has been greatly simplified, the numbers for abrupt control flow constructs show room for further improvement.

We have not fully explored all possible analyses and transformations that could help remove local variables. The increase in the number of local variables seen in Dava is due largely because of the effect of removing local variable webs from the bytecode. Also stack locations are allocated intermediate local variables which result in an increase in the number of locals at the `jimple` level. `grimp` does a decent job of aggregating expressions and in doing so removes a large number of stack variables. However, there is still a large gap between the actual number of variables used in the original source and that produced by Dava. Although it would be an interesting experiment to see by how much the number of locals can be reduced, the effect of having fewer locals in a program on program comprehensibility is a grey area. Where too many locals (indirections) might be confusing to the programmer too few locals might also be a complicating factor since then the programmer has to track the current value stored in a local. We think there is a need to find the right balance in dealing with the number of locals such that they don't cause any added complications.

The aggregations carried out in Chapter 5, the for-loop detection patterns (Section 7.1.1) and the breaking of the `If-Else` statement discussed in Section 5.3.1 are some of the design decisions which are related to a certain style of programming. By allowing a customizable Dava back-end, where the user gets to decide which transformations to apply, the output could be transformed to best suit the individual rather than the general programming community. Work on this was already started by setting flags for advanced transformations *e.g.*, constant substitution and aggressively producing potentially more complicated but compilable code (Sections 7.3 and 4.8). Making Dava's back-end more adaptable would generate code which is customized to a programmer's personal likes and dislikes. A related idea is to have a formating tool (Jalopy [Jal], JRefactory [JRe]) which could take the output

produced by Dava and pretty print it using customized formatting rules.

11.1.3 Adding comments to decompiler output

Since the goal of decompilation is program comprehension we think that being able to convey any additional information to the user helps in program understanding. A feature to add comments within the decompiler output would be really useful. The best way of achieving this would be to have a `COMMENT` tag associated with each AST node. This tag could be given a value if there is a need to insert some comment into the decompiled output. Tags could be specialized to hold single line comments using the // symbol or the long C style comments if there is a large comment to be added.

A lot of times the application bytecode comes along with the API, in the form of javadocs. Another proposed idea is to parse the javadocs information available and place them in the decompiled output as javadocs style comments.

11.1.4 Stronger refactoring analyses

Now that we have an efficient traversal mechanism and a flow analysis framework the decompiler can use these to implement refactoring. One possible refactoring is method inlining. Using whatever heuristics that seem fit (number of lines of code to be inlined, number of method call sites etc) it might be useful to inline methods. A known obfuscation technique used in JBCO is that library calls are moved to methods with confusing names. This is done since renaming library methods/fields is not possible. By putting a level of indirection the obfuscator is able to confuse the programmer since the new method invocation does not give any clues that this is in fact a library call. In such situations by selectively inlining methods the understandability of the code can be increased.

Another related refactoring is to move methods from one class to another. This again counters an obfuscation technique in which a method is moved to an unrelated class. This creates difficulties for the programmer to reason about the component structure of the application. By using heuristics (method only invoked from methods of a particular class, method performs operations on data of a certain class *etc.*) it might be reasonable to move a method to a different class.

Method de-inlining, moving similar code to a separate method and replacing it with invocations to the newly created method, can also help in simplifying program output. Although this type of obfuscation has not been seen in the obfuscators tested so far, the implementation of this refactoring transformation might simplify un-obfuscated code also.

11.1.5 Identifier Renaming

A naming stage has been included in Dava's back-end. However, so far the naming strategy uses very simple heuristics to name local variables. Firstly, the naming mechanism needs to be modified to include naming classes and methods for obfuscated bytecode. Since current obfuscators mostly concentrate on name obfuscation, having even a basic naming strategy for identifiers in the program will help a programmer understand code.

The Java class libraries provide the best starting point to naming identifiers. Variables assigned from library invocations can be allotted names created using the name of the method invoked. Similarly classes extending or encapsulating library data types can have names which include the library type that they utilize.

Another interesting idea, worth exploring, is doing a "flow-analysis" looking for variable names. The analysis would gather potential names for each identifier, as data flows through a program. Once the sets of potential names is obtained the best possible name is alloted to the identifier. Obviously the "best" name is very subjective but a heuristic-based decision can be made to pick a name from a set of possible names.

Currently we perform naming intra-procedurally. Including interprocedural heuristics, results of a method performing some computation, retrieval of a field from an object *etc.* can add to the set of potential names.

11.2 Conclusions

In dealing with arbitrary bytecode, Dava uses a control flow graph representation of the bytecode to generate valid Java programs. Previously the output of the decompiler was verbose and difficult to understand because of the use of complicated control flow using `break` statements and labeled blocks. Also the absence of boolean expression aggregation

resulted in a large number of conditional constructs making the code output harder to track.

This thesis introduces a new back-end to Dava based on matching patterns to simplify the control flow of the decompiled output. Our philosophy for writing transformations has been that a smaller number of conditional statements and less verbose code are easier to understand. Transformations, implemented using the Visitor design pattern implementation for the AST, perform semantically-equivalent rewrites of the AST.

More complicated transformations have been enabled using a newly implemented structure-based flow analysis framework. The implementation of the framework was a non-trivial task resulting from the complexities introduced by complex Java constructs *e.g.*, `Try-Catch`, `Switch` and dealing with `break` and `continue` statements. The framework is extensible, hence making it possible for researchers to test new simplification and refactoring techniques for the decompiler.

Currently Dava uses the flow analysis framework to implement flow analyses often used for compiler optimizations. This is a novel application of compiler analyses that have been traditionally used for execution performance improvements. Certain more complicated transformations were implemented using these flow-analyses to help Dava simplify code generated for obfuscated bytecode.

We have developed metrics that identify code complexity in terms of code layout and number of constructs. We chose several benchmarks and performed compile-decompile experiments on them. We observed the change in values of complexity metrics with the code simplification transformations enabled or disabled. It was quite obvious that Dava's new back-end reduces the complexity of the output making it more comprehensible. Another set of experiments following the compile-obfuscate-decompile pattern was also carried out. The results of these experiments showed that along with being able to correctly decompile obfuscated code, the complexity of the decompiled code can be greatly reduced with the new back-end transformations enabled.

Dava, with its general applicability for Java bytecode along with its AST rewriting transformations, is a robust decompiler compared to its peers. We feel that with more work on programming idioms support, and the suggestions mentioned as future work, Dava will stand out as the decompiler of choice for reverse-engineering applications from Java bytecode.

Bibliography

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

[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *Aspect-Oriented Software Develipment Conference*, 2005, pages 87–98.

[AMP] Axiomatic Multi-Platform C compiler suite. <http://www.axiomsol.com> .

[asp03] The AspectJ home page, 2003. <http://eclipse.org/aspectj/> .

[BGI+01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139, 2001.

[Cav] Cavaj Java Decompiler. <http://www.bysoft.se/sureshot/cavaj/> .

[Con04] Richard Conn. A reusable, academic-strength, metrics-based software engineering process for capstone courses and projects. In *Proceedings of the*

35th SIGCSE Technical Symposium on Computer Science Education, Norfolk, Virginia, USA, 2004, pages 492–496.

[Dec] Decafe Pro.
<http://decafe.hypermart.net/> .

[DJJ] Dj Java Decompiler.
<http://members.fortunecity.com/neshkov/dj.html> .

[Ema93] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, School of Computer Science, McGill University, August 1993.

[Fro] Frontend Plus.
<http://www.softpile.com/development/java/review_03171_index.html> .

[GH98] Etienne M. Gagnon and Laurie J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, 1998, page 140. IEEE Computer Society, Washington, DC, USA.

[GHM00] Etienne M. Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium 2000*, June 2000, Lecture Notes in Computer Science, pages 199–219.

[GJS97] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1997.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[HDE+93] Laurie J. Hendren, Chris Donawa, Maryam Emami, Guang R. Gao, Justiani, and Bhama Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, 1993, pages 406–420. Springer-Verlag.

[HF82] Peter G. Hamer and Gillian D. Frewin. M.H. halstead's software science - a critical examination. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, Tokyo, Japan, 1982, pages 197–206. IEEE Computer Society Press, Los Alamitos, CA, USA.

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

[Jal] Jalopy: the source code formatting tool. <hhttp://jalopy.sourceforge.net/> .

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

[Jav] Java Programming Language. <http://java.sun.com> .

[JRe] A Java Refactoring Tool. <http://jrefactory.sourceforge.net/> .

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *European Conference on Object-oriented Programming*, 2001, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer.

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

[Klab] Zelix KlassMaster - The second generation Java Obfuscator. `http://www.-zelix.com/klassmasterfeaturesFlowObfuscation.html`.

[LSP05] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pages 214–223.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[McC76] Thomas J. McCabe. A complexity metric. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[MH01] Jerome Miecznikowski and Laurie Hendren. Decompiling Java using staged encapsulation. In *Proceedings of the Working Conference on Reverse Engineering*, October 2001, pages 368–374.

[MH02] J. Miecnikowski and L. J. Hendren. Decompiling Java bytecode: problems, traps and pitfalls. In R. N. Horspool, editor, *Compiler Construction*, 2002, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127. Springer Verlag.

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

[RCC91] Pierre N. Robillard, Daniel Coupal, and François Coallier. Profiling software through the use of metrics. *Softw. Pract. Exper.*, 21(5):507–518, 1991.

[Sha80] Micha Sharir. Structural analysis: A new approch to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.

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

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

[Sun] Sun Microsystems. <http://www.sun.com> .

[Tai84] Kuo-Chung Tai. A program complexity metric based on data flow information in control graphs. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, Orlando, Florida, United States, 1984, pages 239–248. IEEE Press, Piscataway, NJ, USA.

[VRGH+00] 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*, March 2000, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34. Springer, Berlin, Germany.

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