The BLOAT Book

David M. Whitlock

October 20, 1999

Introduction

Nate Nystrom is the **man**! No question about it. In less than one year he sat down, designed, and implemented BLOAT, the Bytecode-Level Optimizer and Analysis Tools. Yes, it's a horrible acronym for software that is supposed to make Java run faster, but it works. In that year, Nate managed to writing BLOAT, run some benchmarks on BLOATed code to demonstrate its worth, compose he thesis, and escape to Silicon Valley.

A couple of months later I came on the scene as a new graduate student. Nate's and my advisor, Tony Hosking, pointed me at the BLOAT source code and said, "Okay. Learn it." So, there I was, presented with 50,000 lines of Java code...that were completely undocumented! Now, Nate is a great guy and all, but in his documentation skills leave something to be desired. I'll never forget the first time I met Nate and he tried to convince me that his code was "self-documenting". Right.

After spending several months trying to figure out what BLOAT was all about, I finally decided to get serious and write this book. I knew that if I was ever going to use and improve BLOAT, I would need a solid understanding of the optimizations it performs and how they are implemented. I also wanted to make BLOAT more approachable so that other people could make use of its modeling and optimization facilities. That, and the fact that I'm a stickler for documentation.

It is assumed that the reader has a working knowledge of the Java Virtual Machine Specification. If not, I recommend the appropriately-named *Java Virtual Machine Specification* by Tim Lindholm and Frank Yellin. *The BLOAT Book* was written for BLOAT 0.8.0 and is organized into two parts. The first part focuses on how BLOAT models Java classes. Chapter 1 describes BLOAT's class reflection mechanism and describes how BLOAT loads classes from files. Chapter 2 gives a brief overview of `java.util` package whose classes are used extensively in BLOAT and describes BLOAT's own utility classes. Chapter 3 shows how BLOAT can be used to edit Java classes.

BLOAT's expression tree representation of Java instructions is discussed in Chapter 4. Chapter 5 describes the control flow graph used to model Java methods. Chapter 6 covers Static Single Assignment form. Chapter 7 shows how BLOAT generates Java bytecodes from a control flow graph.

The second part of the book gives details about the optimizations that BLOAT performs. Both the optimization algorithms and their implementations are discussed. Chapter 8 covers a number of optimizations including dead code elimination, value numbering, constant and copy propagation, and peephole optimizations. Chapter 9 describes type inferencing and type-based alias analysis. The book concludes with an in-depth description of the partial redundancy elimination algorithm which is at the heart of BLOAT. Throughout the book I have tried to give examples of what BLOAT does. To save ink, I have ommitted "`EDU.purdue.cs`" from the names of BLOAT's classes. Each chapter concludes with a brief summary.

BLOAT is an amazing piece of software. It does a lot of neat things and it is obvious that Nate put a lot of time and energy into it. And now, it's understandable.

David Whitlock
October 2, 1999

Contents

I Modeling Java Classes **11**

1 Modeling the Java Classfile **13**

1.1 The Reflection Mechanism 13
1.1.1 Interfaces in the Reflection Package 13
1.1.2 The Constant Pool 14
1.1.3 Exception Handlers 15
1.1.4 Debugging Information 15
1.2 Loading Classes from Files 16
1.2.1 Modeling a Classfile 16
1.2.2 Modeling Attributes 17
1.2.3 Modeling Fields 18
1.2.4 Modeling Methods 18
1.3 Summary 19

2 Utility Classes **21**

2.1 The `java.util` package 21
2.1.1 Utility Interfaces 21
2.1.2 Implementations 22
2.2 BLOAT Utility Classes 23
2.2.1 Representing a Directed Graph 23
2.2.2 More Collection Classes 24

3 Editing Java Classfiles **25**

3.1 Editing Many Classes 25
3.2 Editing Pieces of a Class 26
3.2.1 Modeling Parts of the Classfile 26
3.2.2 Editing Class Information 27
3.2.3 Editing Field Information 27
3.3 Modeling Methods 28

3.3.1 Java Virtual Machine Instructions 28
3.3.2 Editing a Method 30
3.4 Summary 31

4 Expression Trees 33
4.1 A Node In an Expression Tree 33
4.2 Visiting the Nodes In an Expression Tree 34
4.3 Simulating the Operand Stack 34
4.4 Node Types 35
4.5 Expressions 35
4.5.1 Basic Expressions 36
4.5.2 Expressions For Calling Methods 37
4.5.3 Expressions That Check Things 37
4.5.4 Boolean Expressions 38
4.5.5 Expressions That Define Local Variables 38
4.6 Statements 39
4.6.1 Statements That Change Control Flow 40
4.6.2 ϕ-Statements 42
4.7 Interfaces Used In the Expression Tree 42
4.8 Constructing the Expression Tree 43
4.8.1 Adding Instructions to the Tree 43
4.8.2 Visiting Instructions and Building the Tree 44
4.9 Summary 52

5 Control Flow Graphs 53
5.1 Background 53
5.2 Basic Blocks 54
5.3 Exceptions 55
5.4 Modeling Control Flow Graphs 56
5.5 Constructing the Control Flow Graph 57
5.5.1 Building Basic Blocks 57
5.5.2 Dealing With Exception Handlers 57
5.5.3 A Quick Regroup 59
5.5.4 Building Expression Trees 59
5.6 Initializing the Control Flow Graph 61
5.6.1 Building the Dominator Tree 61
5.6.2 Computing the Dominance Frontier 62
5.6.3 Preparing for ϕ-statement Insertion 63
5.6.4 Splitting Irreducible Loops 64
5.6.5 Splitting Reducible Loops 65

5.6.6 Determining the Types of Blocks 66
5.6.7 Building the Loop Tree 66
5.6.8 Peeling Loops 66
5.6.9 Removing Critical Edges 68
5.6.10 Inserting Stores after conditional statements 68
5.6.11 Inserting Stores Before Protected Regions 71
5.6.12 Verifying the Correctness of the Control Flow Graph . 71
5.6.13 Committing Changes to the Control Flow Graph . . . 72
5.6.14 Looking at Control Flow Graphs 72
5.7 Control Flow Graph Examples 76
5.7.1 A Simple Example 76
5.7.2 Stack Variables 77
5.7.3 Exceptions 78
5.7.4 A Finally Clause 80
5.8 Summary 80

6 Static Single Assignment Form 85
6.1 Background 85
6.1.1 Placing ϕ-functions 85
6.1.2 Naming Variables In SSA Form 86
6.1.3 Deconstructing SSA Form 87
6.1.4 Other ϕ-functions 87
6.2 Constructing SSA Form 89
6.2.1 Placing ϕ Statements 90
6.3 Renaming SSA Variables 91
6.3.1 Modifying the Blocks 93
6.4 Examples of SSA Form 93
6.4.1 An Example ϕ_c-statement 95
6.5 Summary 97

7 Code Generation 99
7.1 Liveness Analysis 99
7.2 Register Allocation 100
7.3 Code Generation 102
7.3.1 Auxiliary Methods 102
7.3.2 Replacing ϕ Statements 102
7.3.3 Simplifying Control Flow 103
7.3.4 Allocating Subroutine Return Addresses 104
7.3.5 Generating Code 105
7.4 Summary 110

II Optimizing Java Classes **111**

8 Program Transformations **113**
8.1 Array Initialization Compaction . . . 113
8.2 Dead Code Elimination . . . 115
8.2.1 An Example of Dead Code Elimination . . . 116
8.3 Value Numbering . . . 117
8.3.1 The SSA Graph . . . 117
8.3.2 Implementation . . . 120
8.4 Value Folding . . . 125
8.5 Expression Propagation . . . 125
8.5.1 An Example of Expression Propagation . . . 126
8.6 Eliminating Persistent Checks . . . 127
8.7 Peephole Optimizations . . . 129
8.8 Summary . . . 131

9 Type-Based Alias Analysis **133**
9.1 BLOAT's Class Hierarchy . . . 133
9.2 Type Inferencing . . . 134
9.2.1 An Example of Type Inferencing . . . 137
9.3 Type-Based Alias Analysis . . . 139
9.3.1 Implementation . . . 140
9.4 Summary . . . 141

10 Partial Redundancy Elimination **143**
10.1 Background . . . 143
10.1.1 Φ-insertion . . . 144
10.1.2 Renaming . . . 146
10.1.3 Computing Down Safety . . . 148
10.1.4 Will Be Available . . . 151
10.1.5 Finalize . . . 152
10.1.6 Code Motion . . . 154
10.1.7 But, wait. There's more! . . . 155
10.1.8 PRE for Access Paths . . . 158
10.2 Implementation . . . 158
10.2.1 Kills . . . 159
10.2.2 Modeling Φ-statements . . . 159
10.2.3 The Worklist . . . 159
10.2.4 Is an Expression First Order? . . . 160
10.2.5 Constructing the Worklist . . . 161

10.2.6 Inserting Φ-statements 162
10.3 That's all, folks 163
10.4 Examples of PRE 163

Part I

Modeling Java Classes

Chapter 1

Modeling the Java Classfile

Before BLOAT can analyze and optimize Java classes, it must read the class's raw bytecode. Two packages in BLOAT, `bloat.reflect` and `bloat.file` are used to read a Java class file from disk and model it using various classes.

1.1 The Reflection Mechanism

BLOAT provides a reflection mechanism to abstractly model access to a Java class. It is provided so that information about a class may be obtained independently of how it is stored. Currently, classes are only loaded into BLOAT from files, but the reflection mechanism allows classes to be loaded from a network or another virtual machine. As such, `bloat.reflect` provides several interfaces through which a class may be accessed and several classes that represent fundamental parts of a Java class such as the constant pool.

1.1.1 Interfaces in the Reflection Package

There are four interfaces in `bloat.reflect` that specify a set of methods through which a class may be accessed and modified. `bloat.reflect.ClassInfo` grants access to information about a class's superclass, the interfaces it implements, its fields and methods, its modifiers, and its constant pool. `bloat.reflect.ClassInfo` is implemented by `bloat.file.ClassFile` (see section 1.2.1). Access to a class is further refined by `bloat.editor.ClassEditor` (see section 3.2.2).

A `bloat.reflect.ClassInfoLoader` provides one method, `loadClass`, that loads a class of a given name. `bloat.reflect.FieldInfo` allows a method's name and type (represented as indices into the constant pool) as well as its modifiers to be accessed and modified. `FieldInfo` is implemented by `bloat.file.Field` (see section 1.2.3). `bloat.reflect.MethodInfo` grants access to information about a method such as its declaring class, its name and type, and the types of any exceptions it may throw (all represented as indices into the constant pool). `MethodInfo` is implemented by `bloat.file.Method` (see section 1.2.4).

Modifiers (a.k.a. access flags)

The `bloat.reflect.Modifiers` interface contains a number of constant (`public static final`) `shorts` that are used as masks to determine whether a class, method, or field has certain attributes (such as being public, private, final, static, etc.).

1.1.2 The Constant Pool

Each Java class has a number of constants associated with it. These constants may be the initial values of variables, numbers that are frequently used in the program, or the name of the class itself. These constants are grouped together into the class's *constant pool*. The BLOAT reflection mechanism models constants in the constant pool with `bloat.reflect.Constant`. Each `Constant` consists of a tag and a value. All values are a `java.lang.Object` and tags fall into one of several categories:

`UTF8` Represents constant string values in a special compact format. BLOAT represents its value as a `String`. You can read all about it in [LY96].

`CLASS` Represents a class or interface. Its value is an `Integer` representing the index into the constant pool of the name (a UTF8) of the class or interface.

`FIELD_REF/METHOD_REF/INTERFACE_METHOD_REF` Represents a field or method. Its value is an array of two `ints`. The first is the index into the constant pool specifying the class/interface in which the field/method is declared. The second is an index into the constant pool indicating the name and type of the field/method.

`STRING` Represents a constant `java.lang.String`. Its value is an `Integer` that is the index into the constant pool for its UTF8 string constant.

`INTEGER/FLOAT` Represents a four-byte (`int` or `float`) numeric constant. Its value is an `Integer` or `Float`.

`LONG/DOUBLE` Represents an eight-byte (`long` or `double`) numeric constant. Its value is an `Long` or `Double`.

`NAME_AND_TYPE` Represents a field or method without specifying which class or interface to which it belongs. Its value is an array of two `int`s. The first `int` is an index into the constant pool indexing a UTF8 string that specifies the name of the field or method. The second is an index into the constant pool indexing a UTF8 string representing the field's or method's descriptor.

1.1.3 Exception Handlers

A method's bytecode contains instructions for each exception handler. In the classfile, exception handler information is grouped together in exception table. Entries in that table are modeled in BLOAT with `bloat.reflect.Catch`. Instances of `Catch` are created by `bloat.file.Code` (see section 1.2.4).

An exception handler consists of the starting and ending indices into the code array indicating the code region in which the exception handler is active. This is equivalent to the exception handler's `try` block. A `Catch` object also stores the index into the code array of the start of the exception handler and the index into the constant pool of the type of the exception that is caught.

1.1.4 Debugging Information

The Java virtual machine specification gives guidelines to the format of two kinds of debugging information. The first kind is found in the the line number table attribute (see section 1.2.4) of the code array. It is modeled by `bloat.reflect.LineNumberDebugInfo` and consists of a line number in a Java source file and an index into the code array specifying the first instruction corresponding to that line.

Another kind of debugging information discussed in the virtual machine specification is local debugging information. This information allows a debugger to obtain the value of a given local variable during program execution and is modeled by `bloat.reflect.LocalDebugInfo`. A `LocalDebugInfo` object consists of an index into the code array at which the variable must have a value and length (number of instruction) during which the variable will have a value. Two indices into the constant pool, corresponding to the name of the local variable and its type descriptor, are also maintained. Finally, the index of variable itself (i.e. which number local variable it is) is stored in `LocalDebugInfo`. Instances of class `bloat.file.LocalVariableTable` (see section 1.2.4) contain an array of `LocalDebugInfo`.

1.2 Loading Classes from Files

The classes in `bloat.file` implement the interfaces of the reflection mechanism. Specifically `bloat.file` provides classes that read a classfile from a file on disk and model it.

`bloat.file.ClassFileLoader` does the work of loading a class and creating a `ClassInfo` (see section 1.1.1) object representing the class. If the class is not found, a `ClassNotFoundException` is throw. A `ClassFileLoader` searches its class path (see method `setClassPath`) for classes to load. The class path defaults to the JVM's class path (the `java.class.path` property). Any class files that are written to disk ("committed") are placed in the output directory (see method `setOutputDir`).

The method `loadClass` in class `ClassFileLoader` searches for a class file of a given name. A class may be specified with its full package name (e.g. `java.lang.String`) or by its class file (e.g. `myclass/Test.class`). With the help of private methods `loadClassFromFile` and `loadClassFromStream` it creates a `java.io.File` object representing the file and creates a new `ClassInfo` object from the file's stream.

`ClassFileLoader` maintains a cache of the `ClassFile`s that it has most recently loaded. Note that `ClassFile` implements `ClassInfo` (see section 1.2.1). If the class file is not found in the cache, the class path is searched. The class path may contain directories, Jar files, or Zip files. If the class is not found along the class path, then the current directory is searched as a last resort.

1.2.1 Modeling a Classfile

A classfile is read from disk and is modeled using a `bloat.file.ClassFile`. The contents of a `ClassFile` are read from a `java.ioDataInputStream`. Each `ClassFile` knows the `ClassFileLoader` that loaded it. A `ClassFile` implements the `reflect.ClassInfo` interface (see section 1.1.1). A classfile is read from disk and its contents are modeled using objects of various classes as discussed below. If while during the reading something goes wrong, a `bloat.reflect.ClassFormatException` is thrown.

For the most part, a `ClassFile` object models the class file as it is represented on disk. The details of the classfile format can be found in [LY96]. A classfile's "header" consists of its magic number (`0xCAFEBABE`) and its major and minor number. The constants in a classfile's constant pool are read and are modeled as an array of instances of `bloat.reflect.Constant`. A classfile's modifiers (also known as access flags) are an unsigned short whose bits determine whether or not the class is public, final, an interface, etc. Information such as a class's superclass and the interfaces it implements are represented by indices into the constant pool. A class's fields (i.e. a class's data members, see section 1.2.3) are represented by an array of `file.Field`. A class's methods (see section 1.2.4) are represented by an array of `bloat.file.Method`. A class's attributes (see section 1.2.2) give information about the class and are modeled by an array of `bloat.file.Attribute`. Attributes are used to represent miscellaneous information such as the name of the source file from which a classfile was compiled.

A classfile modeled by a `ClassFile` object can be written to disk by invoking the `commit` method. It looks to its `ClassFileLoader` to get the `File` to which it is written.

1.2.2 Modeling Attributes

Attributes are general description mechanisms used in a Java class file. Classes, fields, methods, and bytecode all have attributes. BLOAT models attributes with `bloat.file.Attribute`. Each attribute consists of the name of the attribute (represented as an index into the constant pool) and the length of the attribute (not including the space to store the name and the length). `Attribute` is extended to represent code (see section 1.2.4), exceptions (see section 1.2.4), a constant value (see section 1.2.3), as well as debugging information.

Generic Attributes

The Java virtual machine specification allows Java implementors to use arbitrary attributes for their own purposes (e.g. additional debugging information). BLOAT uses `bloat.file.GenericAttribute` to model these attributes. `GenericAttribute` is a subclass of `Attribute` and therefore has a name (index into the constant pool) and a length. It also contains an array of `byte`s that holds the generic attribute's raw data.

1.2.3 Modeling Fields

A classfile represents its data members by a field. The `bloat.file.Field` class models fields in a Java classfile. Each `Field` is read from a `java.io.DataInputStream` by a `ClassFile` object (see section 1.2.1). A field must know about the `ClassFile` to which it belongs so that it may access its constant pool, etc.

Each field has a bit vector whose bits determine the field modifiers (public, private, static, final, etc.). The field's name and descriptor are represented by indices into its class's constant pool (see section 1.1.2). A field may have attributes associated with it (see section 1.2.2). A common attribute of a field is its constant value (see section 1.2.3).

The Constant Value Attribute

A constant value attribute is represented by `bloat.file.ConstantValue` which is a subclass of `Attribute`. A `ConstantValue` is read from a `java.io.DataInputStream` and has a name (index into the constant pool) and a length associated with it. It also has an index into the constant pool that represents the constant value itself.

1.2.4 Modeling Methods

The Method Itself

Methods in a classfile are modeled with instances of `bloat.file.Method`. A `ClassFile` reads method information from a `java.io.DataInputStream`. A field has an unsigned short whose bits represent its modifiers (access flags). Modifiers determine whether or not a method is public, private, static, final, synchronized, etc. A method's name and its value are both represented as indices into the constant pool. A method's value is a method descriptor representing the types of the method's parameters and its return type. A method's attributes are modeled as an array of `Attribute` (see section 1.2.2). Two special attributes that methods have are code (modeled by `Code`) and exceptions (modeled by `Exceptions`).

The Code Attribute

A code attribute contains a method's instructions (bytecodes) and other auxiliary information. A code attribute is modeled in BLOAT by `bloat.file.Code`, a subclass of `Attribute`. Like all attributes, a code attribute has a name (index into the constant pool) and a length. `Code` also contains a the maximum number of words (`maxStack`) that can be on the operand stack during the method's execution and the number of local variables (`maxLocals`) the method has including its parameters. The actual code itself is modeled as an array of `bytes`. Exception handlers in the method are modeled by an array of `Catch` instances (see section 1.1.3). Code may also have attributes. Two interesting attributes are the line number table and the local variable table.

The Line Number Table Attribute

The line number table attribute is an optional attribute that may be used by Java debuggers to determine which portion of the code array corresponds to which line number in the original source file. BLOAT models it by `bloat.file.LineNumberTable`. In addition to having a name and length, it essentially consists of an array of `LineNumberDebugInfo` (see section 1.1.4).

The Local Variable Table

The local variable table is an optional attribute that may be used by Java debuggers to determine the value of a given local variable during program execution. BLOAT represents this table by `bloat.file.LocalVariableTable`. Like all attributes, `LocalVariableTable` has a name and a length. It also contains an array of `LocalDebugInfo` (see section 1.1.4).

The Exceptions Attribute

A method's exceptions attribute indicates which checked exceptions a method may throw. It is modeled by `bloat.file.Exceptions`. An `Exceptions` consists of a name (index into the constant pool) and a length. Additionally, it consists of an array of `int`'s that holds the indices into the constant pool that represent information about the types of exceptions that may be thrown.

1.3 Summary

The classes in `bloat.reflect` and `bloat.file` are used to model Java classes at the lowest level. `bloat.reflect` provides an abstract interface through which class file data may be accessed and modified. It also models several essential portions of a Java class such as its constant pool. `bloat.`

`file` implements the reflection interface to work with classes that reside in a file. It faithfully represents classes according to [LY96]. At this stage, methods are still represented as raw bytes, constants in the constant pool are referred to by their indices, and exceptions are modeled as offsets into the code array.

Chapter 2

Utility Classes

BLOAT uses a number of utility classes to help it model and optimize Java classfiles. Many of these classes come from the JDK1.2 version of the `java.util` package. Others are unique to BLOAT.

2.1 The `java.util` package

Starting in JDK1.2 ("Java 2"), the `java.util` package came with a number of utility classes that gave greater power and flexibility over classes such as `java.util.Hashtable` and `java.util.Enumerator`. The new utility classes are used extensively in BLOAT[1].

2.1.1 Utility Interfaces

Collections

The `Collection` interface represents a group of `Objects`. It has methods to add and remove `Objects`, search it for a given `Object`, to return an array of the `Objects` in the `Collection`, and to get an `Iterator` (see section 2.1.1) over the `Collection`.

`Collection` is subclassed by `List` and `Set`. A `List` is an ordered collection. Each element in the list has an integer index. In addition to obtaining an `Iterator` over a `List`, one may also obtain a `ListIterator` (see section

[1] In fact, Nate once admitted to me that he might have gone a little overboard with the util classes. Unfortunately, in the Spring of 1998 when he was coding BLOAT, JDK1.2 was not finalized. As a result, he used the beta source code of the classes. When the final version was released, some changes were made to the API and I had fun fixing those bugs. Not really.

2.1.1). A `Set` is a collection that contains no duplicate elements. If one attempts to add a duplicate `Object` to a `Set`, a `ClassCastException` is thrown.

Iterators

JDK1.1 provided `java.util.Enumeration` which allowed you to traipse through a bunch of `Objects`. JDK1.2 improves on this concept with `java.util.Iterator` interface. A thorough description of the iterator pattern is given in [GHJV95]. An `Iterator` iterates over the elements of a `Collection` (see section 2.1.1) and allows you to remove an element from the underlying `Collection` once the `next` method has been called.

`java.util.ListIterator` provides more flexibility. A `ListIterator` has indexed elements, can traverse a `List` in either direction and can insert, remove, or modify elements in the `List`. Note that when an element is added to a `ListIterator`, it is added so that a call to `next` would be unaffected and a call to `previous` would return the added value.

Comparing Objects

The `java.util.Comparator` interface has two methods, `compare` and `equals`. A `Comparator` is used to impose a total ordering on a `Collection`. Objects that implement `Comparable` may also be compared to other objects. BLOAT doesn't use these guys too much, however the concept is used in classes such as `bloat.trans.NodeComparator` (see section 8.3.2).

Maps

`java.util.Map` is an interface for classes that map one `Object`, the *key*, to another `Object`, the *value*. A `Map` may not contain duplicate keys. A `Map` may be viewed as a set of keys, a collection of values, or a set of key-value mappings. A `Map` basically behaves like a hash table.

2.1.2 Implementations

JDK1.2 implements the above interfaces in a number of different ways. Different implementations have different characteristics (e.g. a hash table versus a red-and-black trees), but have the same interface.

Several abstract classes (`AbstractCollection`, `AbstractList`, `AbstractMap`, and `AbstractSet`) are supplied to make the job of implementing the `java.util` interfaces easier. An `AbstractSequentialList` is an abstract class of `List` implementations that are sequential in nature (e.g. a linked list as opposed to an array).

`ArrayList` implements the `List` interface and allows you to change its size. It is kind of like a `Vector`. `LinkedList` implements the `List` interface and provide methods so that it may behave like a stack or queue.

`HashMap` implements the `Map` interface. It is roughly equivalent to a hash table, and does not guarantee the order of its mapping. `TreeMap` implements the `Map` interface using a red-and-black tree. Its keys are sorted in ascending order.

It is left as an exercise to the reader to figure out what `HashSet` and `TreeSet` are. I knew you could do it.

Other Interesting Classes

`java.util.Arrays` provides a number of static methods for dealing with (e.g. searching and sorting) arrays. `java.util.Collections` provides static methods for dealing with (e.g. search, copy, sort) `Collections`.

2.2 BLOAT Utility Classes

In addition to the classes that come with JDK1.2, there are several utility classes that were written especially for BLOAT.

2.2.1 Representing a Directed Graph

`bloat.util.Graph` represents a directed graph of `GraphNode`s. A `GraphNode` is added to a `Graph`, then an edge between that node and another is added. Each node in the graph has a unique key associated with it. For instance, if the `Graph` represents a control flow graph (see section 5.4), each `GraphNode` would have a basic block associated with it. When a `GraphNode` is added to a `Graph`, the graph is considered modified. Before any information about the graph can be gleaned, such as its pre- and post-order traversals, must be recalculated. To facilitate this, `Graph` maintaines a modification count of the nodes and edges in the graph.

A `GraphNode` contains a set of successors and predecessor as well as its index in a pre-order and a post-order traversal of the `Graph` in which it resides. All of this information is calculated by a `GraphNode`'s `Graph`.

A `Graph` may have a number of roots and reverse roots. Each `Graph` maintains a pre-order and post-order traversal of itself. This is used to determine the indices of its nodes. Through `Graph`, one can obtain information about its nodes such as their indices in a pre-order or post-order traversal of the graph and two nodes ancestor/descendent relationship.

2.2.2 More Collection Classes

`bloat.util.IdentityComparator` has one method, `compare`, that compares two `Objects` using the `System.identityHashCode` method, which returns the hash code that would be returned by `Object.hashCode()` regardless if a class overrides `hashCode`. I don't think it is ever used.

`bloat.util.ImmutableIterator` is an `Iterator` whose contents cannot be changed. Its `remove` method has no effect.

`bloat.util.ResizableArrayList` is a subclass of `ArrayList` that differs only in the fact that empty space is padded with `null` values. This way, the `size` method will return the length of the array not just the number of elements in it.

`bloat.util.UnionFind` is used represent disjoint sets of integers. There are two common operations on disjoint sets that we want to be executed efficiently. We want to be able to obtain the set in which a given integer resides ("find") and we also want to combine the contents of two disjoint sets ("union").

`UnionFind` represents a disjoint set as a tree of `Nodes`[2]. Each `Node` knows its parent, child, and integer value. A `Node`'s parent is the root of the tree in which it resides. Each `Node` also has a "rank" associated with it that approximates the size of the logarithm of the a `Node`'s subtree (i.e. the height of the subtree). The rank is used when combining (unioning) two sets. When two sets are unioned, the root with the smaller rank is made to point to the root with the larger rank.

`UnionFind` has methods to find the set in which an integer resides, to determine whether or not two integers are in the same set, and to compute the union of two sets. `UnionFind` is used to determine the type (i.e. `HEADER`, `NON_HEADER`, etc.) of basic blocks in method `setBlockTypes` of class `bloat.cfg.FlowGraph` (see section 5.6.6).

[2]`Node` is a class that is local to `UnionFind` and should not be confused with `bloat.tree.Node`.

Chapter 3

Editing Java Classfiles

Once Java classes have been read from a file on disk, or wherever else they may reside, they are edited using the classes in the `bloat.editor` package.

3.1 Editing Many Classes

`bloat.editor.Editor` is a central repository for all information regarding editing classes. It maintains a `Collection` (see section 2.1.1) of the names of classes it knows about. It also knows about a `bloat.reflect.ClassInfoLoader` (see section 1.1.1) that it uses to load classes.

An `Editor` maintains a number of caches of various editing objects. These caches are implemented as mappings between a reflection object (e.g. `bloat.reflect.ClassInfo`) and its corresponding editing object (e.g. `ClassEditor`). Each editing object also has a *reference count* associated with it. Each time an editing object is accessed via one of the "edit" methods (e.g. `editField`), its reference count is incremented. The "release" methods decrement the reference count and if it equals zero, the editing object is removed from the cache. When editing is complete, an editing object is *committed.* Committed editing objects are written back to where they came from and are removed from their cache. The `hierarchy` method returns a `bloat.tbaa.ClassHierachy` object representing the class hierarchy of all classes in the `Editor` (see section 9.1).

3.2 Editing Pieces of a Class

3.2.1 Modeling Parts of the Classfile

BLOAT's reflection mechanism models the contents of a Java classfile. However, it only goes so far. Recall that instructs are modeled as arrays of `byte` (see section 1.2.4) and that constants in a class's constant pool are just tags and values (see section 1.1.2). The editing mechanism further refines the representation of a Java classfile.

The Constant Pool

A class's constant pool is modeled by `bloat.editor.ConstantPool`. A `ConstantPool` is created from an array of `bloat.reflect.Constants` (see section 1.1.2).

`ConstantPool` maintains several pieces of information about the constants in the constant pool. A mapping between constants and their indices is maintained for the ease of adding new constants to the constant pool. It maintains an list (`bloat.util.ResizableArrayList`) of the constants in their `Constant` form. `Constants` are not resolved until they are needed. Constants are resolved to an instance of `Type`, `String`, `MemberRef`, or `NameAndType`.

Type Descriptors

A (type) descriptor is a string that represents the type of a field or method. They have a funky format that is described in detail in [LY96]. Basically, they encode the type of a field or method using a symbolic notation. For example, a field `int x[]` has the type descriptor

`[I`.

The method `String f(int a, boolean b, Object c)` has the type descriptor

`(IZLjava/lang/Object;)Ljava/lang/String;`.

BLOAT models a type descriptor using `bloat.editor.Type`. A `Type` can be created from a `String`, a `char` representing a primary type, or an `int` representing a primary type. The latter two are used in `bloat.codegen.CodeGenerator` (see section 7.3).

`Type` has a number of constants that are used to identify and build type descriptors. It also has methods to determine whether or not the `Type` it represents an object, array, boolean, etc., the number of slots on the stack it occupies (`stackHeight`), and methods that are specialized for type descriptors of arrays and methods.

Name And Type Information

The type of a method or field are represented in BLOAT by a `Type`. The type descriptor is grouped together with the name of the field or method to form a `bloat.editor.NameAndType`. A `NameAndType` simply consists of a `String` representing the name and a `Type` representing the type.

Field and Method Information With Class

A field or method may be represented along with the class in which it is declared. `bloat.editor.MemberRef` embodies this representation by associating a `NameAndType` of a method or field with the `Type` of the class in which it is declared.

3.2.2 Editing Class Information

`bloat.editor.ClassEditor` gives finer-grain access to a class than `bloat.reflect.ClassInfo` does (see section 1.1.1). A `ClassEditor` is based on a `ClassInfo` object and knows the `Editor` that "owns" it (see section 3.1).

A `ClassEditor` knows its type, the type of its superclass, and the types of the interfaces it implements. All of these types are represented by objects of `Type`. It also has a `ConstantPool`. Through a `ClassEditor` one can easily determine the class's access flags (see section 1.1.1).

3.2.3 Editing Field Information

A class's field is edited with a `bloat.editor.FieldEditor`. A `FieldEditor` is created from a `ClassEditor` and a `bloat.reflect.FieldInfo` (see section 1.1.1). Each `FieldEditor` knows its name, type, and constant value (if appropriate). It obtains this data from constants in its class's constant pool. A `FieldEditor` has methods to access and change its modifiers (e.g. public, private, final, static, etc. see section 1.1.1)

3.3 Modeling Methods

Java methods have a lot of stuff. They've got names, and code, and exceptions, and parameters, and all sorts of nastiness. And we've got to model it all!

3.3.1 Java Virtual Machine Instructions

Dealing With Opcodes

There are a lot of opcodes in the Java Virtual Machine. To keep track of them all, the `bloat.editor.Opcode` interface defines constants to name them all. The `opc_`x constants represent the numerical values of the opcodes. Opcodes that are similar (e.g. all opcodes that load integers) are grouped together with constants of the form `opcx_`x. The actual mapping is stored in `opcXMap`. An array of `String`s, `opcNames`, gives the names of all the opcodes. Finally an array of `byte`s, `opcSize`, gives the size of each opcode.

Local Variables

In Java, method variables and parameters are represented by local variables. BLOAT models local variables with `bloat.editor.LocalVariable`s. Each `LocalVariable` has a name (`String`), a `Type` descriptor, and a number associated with it (`index`). Only local variables with debugging information have a name and type. All local variables have an index.

Labels

`bloat.editor.Label` is used to label an JVM instruction. Each `Label` consists of an integer index indicating its offset into the code array and a `boolean` that determines whether or not it starts a basic block (see section 5.2).

Instructions

BLOAT models Java virtual machine instructions with the `bloat.editor.Instruction` class that implements the `Opcode` interface. Each `Instruction` consists of an integer opcode and an optional `Object` operand. The operand may be an `Integer`, `Float` or one of the special operand types (described below) used to express multiple operands.

An `Instruction` can be created from an opcode and an operand, or it may be read from an array of `bytes`. This form of the `Instruction` constructor also requires the targets and lookups of any instructions that change control flow, (e.g. ifs, return, tableswitch), any local variables the instruction may reference, and the constant pool. In BLOAT this information is compiled in the `munchCode` method of `MethodEditor` (see section 3.3.2).

The `Instruction` class has several methods to determine what kind of instruction (conditional jump, store, etc.) it is. It also has several utility methods for dealing with `byte` data (e.g. turning four `byte`s into an `int`). It also has a `visit` method that allows the `Instruction` to be visited by an `InstructionVisitor` (see section 3.3.1).

Representing Switches

The Java virtual machine instruction tableswitch and lookupswitch are unique in that they have a variable-length operand. In order to accommodate this, the `bloat.editor.Switch` is used to represent their operands. Each `Switch` consists of a mapping between targets (an array of `Label`s) and values (an array of `int`s), as well as a default target. See [LY96] for more details.

Instructions that Increment

Like the switching instructions, the integer increment instruction (iinc) has more than one operand: a local variable to increment and the amount by which to increment. This information is encapsulated by the `bloat.editor.IncOperand` class. Instances of `IncOperand` contain a `LocalVariable` object and an integer specifying the increment.

Creating Multidimensional Arrays

The multianewarray instruction creates a new multidimenional array. As such, it requires a type descriptor for the type of the array and the number of dimensions in the array. This information is modeled by the `bloat.editor.MultiArrayOperand` class. Each instance of `MultiArrayOperand` has a `Type` representing the type of the multidimensional array and an integer representing the number of dimensions.

Visiting Instructions

There are over 200 different instructions for the Java virtual machine. Some of the operations that BLOAT performs (such as simulating a program execution) require knowledge about the behavior of individual instruction. One way of representing the differences between instruction is to have a separate class for each instruction. However, that would mean create over 200 classes for instructions that, for the most part, look and act the same. Instead we use a visitor.

A visitor[1] is another design pattern [GHJV95]. An `bloat.editor.InstructionVisitor` is used to perform operations on a per-instruction basis. `InstructionVisitor` has a method, `visit_`*opcode*, for every instruction that takes an `Instruction` parameter. The visitor pattern simulates double dispatching with the `visit` method. For instance, the `Instruction` class has a `visit` method that takes an `InstructionVisitor` as an argument. The `visit` method switches on the opcode of the `Instruction` and calls the appropriate `visit_`*opcode* method of the `InstructionVisitor`.

Some of the benefits of using the visitor pattern are that functionality is added to classes without the classes being modified and that this functionality is centralized in one class (with lots of methods), the visitor. Visitors are used in several places in BLOAT.

Modeling try-catch Blocks

Exceptions are very important to BLOAT. At this level a try-catch block is modeled by `bloat.editor.TryCatch`. It consists of three `Label`s that label the first instruction in the try block, the last instruction in the try block, and the first instruction in the exception handler. It also contains the `Type` of the exception that is thrown. All of this information is gleaned through a `bloat.reflect.Catch` object (see section 1.1.3).

3.3.2 Editing a Method

A method is edited using a `bloat.editor.MethodEditor` that is constructed from a `ClassEditor` and a `MethodInfo`. A `MethodEditor` knows all sorts of stuff about a method such as its name, type (descriptor), its code (represented as `Label`s and `Instruction`s), parameters (represented as `LocalVariables`), its try-catch blocks, line number information, as well as the maximum height of its stack, the maximum number of local variables it has, and the last label in its code.

A `MethodEditor` obtains its code as an array of `byte`s from its `MethodInfo`. If the `PRESERVE_DEBUG` flag is set, a `MethodEditor` will preserve debugging information by creating `LocalVariable`s with name and type information and will maintain a mapping between `Label`s and line number information.

[1] Dr. Palsberg loves visitors.

The MethodEditor uses a private helper method called munchCode to work with the raw bytecodes. munchCode examines an opcode and extracts information about the opcode that is needed for creating an Instruction object. The targets of branch instructions are determined. The targets and lookup tables are compiled for switch instructions.

The MethodInfo is consulted and Labels are created for beginning and ending instructions for try blocks, as well as the beginning instruction of the exception handler. An instance of TryCatch is created to hold this information.

Finally, Instructions are created and added along with their Labels to a linked list. A Label is also added to the end of the list to signify the start of the next basic block of code.

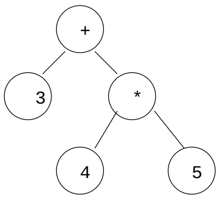
3.4 Summary

Classes in the bloat.editor package further refine the representation of a Java class. The Editor class serves as a central repository for objects which can edit classes, fields, and methods. These objects offer access to type and modifier information. Individual instructions (bytecodes) in Java methods are identified and constructs representing labels (targets of branches) and try-catch block information are maintained.

Chapter 4

Expression Trees

In BLOAT each basic block has an *expression tree* associated with it. An expression tree represents the nested nature of code. For example, `3 + 4 * 5` is represented as:



4.1 A Node In an Expression Tree

DeadCodeEliminationDEAD DeadCodeEliminationLIVE

BLOAT represents a node in the expression tree with class `bloat.tree.Node`. Each `Node` has a parent `Node`, a value number that is used when eliminating redundant expressions, and an integer key that is used by some analyses to indicate whether a node is `LIVE`, `DEAD`, etc. `Node` provides various methods to access these values.

All nodes in the expression tree are subclasses of `Node`. There are two kinds of expression tree nodes: expressions and statements. Expression have a value, statements do not.

There are also methods of `Node` that "clean up" a node. Basically, when a `Node` is cleaned up, it and all of its children are removed from the tree by setting its parent node to `null` and then recursively cleaning its children.

A `Node` may also be replaced by another `Node`. Expressions cannot be replaced by Statements and vice versa. An expression may only be replaced by an expression with the same type descriptor. A `ReplaceVisitor` (see section 4.2) is used to do the actual replacing.

4.2 Visiting the Nodes In an Expression Tree

The visitor pattern allows operations to be performed on objects, but does not require the objects to be aware of the specifics of the operation. Visitors work especially well with a data structure like a tree whose nodes are heterogeneous and well-defined.

The abstract class `bloat.tree.TreeVisitor` provides an interface for visiting an expression tree. It has a `visit`x method for each kind (subclass) of node, `bloat.cfg.Block`, and `bloat.cfg.FlowGraph`. Most of these methods provide a default implementation that delegates the work to other methods. For example `visitConstantExpr` calls `visitExpr`, etc.

There are two concrete subclasses of `TreeVisitor`. The first is `bloat.tree.ReplaceVisitor`. A `ReplaceVisitor` traverses an expression tree and replaces all occurrences of one expression with another. The second is `bloat.tree.PrintVisitor`. A `PrintVisitor` writes a textual representation of an expression tree to a `java.io.PrintWriter`. A detailed description of `PrintVisitor` is deferred until section 5.6.14.

4.3 Simulating the Operand Stack

The Java virtual machine is a stack machine. Many of the JVM's instructions operate on and obtain their operands from a stack. In order to understand the meaning of JVM instructions, the stack behavior must be simulated. The class `bloat.tree.OperandStack` simulates the behavior of of JVM's operand stack. It contains an `java.util.ArrayList` of expressions (`Expr`, see section 4.5) that is the stack.

An `Expr` is pushed onto the stack and the height is adjusted accordingly (see `Type.stackHeight()`, section 3.2.1). Expressions can be popped off the stack in several different ways. The popped expression can be compared against an expected expression `Type`. If the popped expression type does not match the expected type, an exception is thrown. There is also support for popping wide and (explicitly) non-wide expressions off the stack. Additionally, there are methods for peeking into the stack, replacing an expression at a given depth in the stack, and obtaining the height and the size[1] of the

[1] Because of wide expressions, the height and the size (the number of expressions in the stack) of the stack may not be the same.

stack. All of these operations are used when simulating JVM instruction execution when building an expression tree (see section 4.8).

4.4 Node Types

There are many different kinds of Nodes that may populate an expression tree. These nodes are represented by subclasses of `Node`. The `bloat.tree.Tree` class is a subclass of `Node` that represents an expression tree. A `Tree` instance is constructed from the code in a basic block (an instance of `bloat.cfg.Block`, see section 5.2) and an `OperandStack` representing the state of the JVM's stack when the block begins execution. A `Tree` consists of a list of statements (`Stmt`, see section 4.6) and an `OperandStack` whose initial value is a copy of the preceding `OperandStack`.

Statements can be added and removed from a `Tree` in several different ways (e.g. add a statement before another statement, remove the last non-`LabelStmt` statement, etc.). Instructions may also be added to the tree (basic block). Adding instructions is covered in detail in section 4.8.1.

When dealing with the `dup` instructions (see section 4.8.2) it is necessary to "save" the contents of the stack. This is done in the `saveStack` method. If the `USE_STACK` flag is set, then the expressions on the stack are saved to a "stack variable" (see `StackExpr`, section 4.5.5). Else, a new local variable (see `LocalExpr`, section 4.5.5) is created and the stack element is stored into it using a `StoreExpr` (see section 4.5.1). `saveStack` is also called when new statements or instructions are added to the tree (see section 4.8.1).

The `Tree` class is used to construct expression trees. To accomplish, `Tree` implements the `bloat.editor.InstructionVisitor` interface. The details of how the expression tree is built will be covered in section 4.8.

4.5 Expressions

An expression is a node in the expression tree that has a value associated with it. Expressions are modeled with the abstract `bloat.tree.Expr` class. Each `Expr` has a `bloat.editor.Type` associated with it that represent the type of the expression.

In addition to having methods that return the `Type` and defining expression of the expression, `Expr` has methods that determine whether or not it defines a variable (by default, an expression is not), return the statement (`Stmt`, see section 4.6) in which the expression resides (by examining its parent nodes), clone an `Expr` object, and compare two `Exprs`.

4.5.1 Basic Expressions

`bloat.tree.ArithExpr` represents a binary arithmetic operation. It consists of left and right `Expr` operands, and an operator represented by a `char` constant. Legal operators are `ADD`, `SUB`, `DIV`, `MUL`, `REM`, `AND`, `IOR`, `XOR`, `CMP`, `CMPL`, and `CMPG`.

`bloat.tree.ArrayLengthExpr` represents the arraylength instruction that gets the length of an array. It has one operand, a reference to an array (represented as a `Expr`).

`bloat.tree.CastExpr` represents casting an object to a type. It consists of an `Expr` representing the object to be cast, and a `Type` to which the object is to be cast. Note that this type is also the type of the expression.

CatchTHROWABLE

`bloat.tree.CatchExpr` represents catching an exception. It has an instance of `Type` that represents the type of the exception that is thrown. A `CatchExpr`'s expression type is `Catch.THROWABLE`.

`bloat.tree.ConstantExpr` represents a constant value such as an integer, double, or string. It consists of an `Object` representing the constant value.

`bloat.tree.NegExpr` represents the arithmetic negation of an expression. It has an instance of `Expr` that represents the expression that is being negated.

`bloat.tree.NewArrayExpr` represents the newarray instruction. It consists of an `Expr` representing the size of the array being created and the `Type` of the array.

`bloat.tree.NewExpr` represents the new instruction. It knows the `Type` of the object to create.

`bloat.tree.NewMultiArrayExpr` represents the multianewarry instruction for creating a new multidimensional array. It has an array of `Expr` representing the dimensions of the array and the `Type` of the elements in the array.

TypeADDRESS

`bloat.tree.ReturnAddressExpr` represents a return address and has type `Type.ADDRESS`.

`bloat.tree.ShiftExpr` represents a bit shift operation. It consists of an integer constant representing the direction to shift (`LEFT`, `RIGHT`, `UNSIGNED_RIGHT`), an `Expr` representing the expression to shift, and an `Expr` representing the number of bits by which to shift.

`bloat.tree.StoreExpr` represents a store of an expression into a memory location. It consists of a `MemExpr` (see section 4.5.5) into which a `Expr` is to be stored. `StoreExpr` implements the `Assign` interface (see section 4.7) because it involves an assignment.

4.5.2 Expressions For Calling Methods

The abstract class `bloat.tree.CallExpr` represents invoking a method. As one might expect, it consists of an array of `Expr` representing the parameters to the method and an `bloat.editor.MemberRef` (see section 3.2.1) object representing the method.

CallMethodExprVIRTUAL CallMethodExprNONVIRTUAL CallMethodExprINTERFACE

Calls to an instance method are modeled with `bloat.tree.CallMethodExpr`. `CallMethodExpr` augments `CallExpr` with an integer representing what "kind" of method is being called (`VIRTUAL`, `NONVIRTUAL`, or `INTERFACE`) and an `Expr` representing the receiver object on which the method is invoked.

Calls to class methods (the invokestatic instruction) are modeled with `bloat.tree.CallStaticExpr`. `CallStaticExpr` is simple: it just contains an array of parameters (`Expr`) and a `MethodRef`.

4.5.3 Expressions That Check Things

Several Java instructions result in things that need to be checked. The abstract class `bloat.tree.CheckExpr` models such instructions. A `CheckExpr` contains an express (`Expr`) that is checked. There are three subclasses of `CheckExpr`.

UCExprPOINTER UCExprSCALAR

Class `bloat.tree.RCExpr` represents the *residency check* opcode (rc) that is present in the PJama virtual machine. It just has an `Expr` to check. A `bloat.tree.UCExpr` represents the *update check* opcode (uc) that is present in the PJama virtual machine. In addition to an `Expr` to check, `UCExpr` also has an integer kind (either `POINTER` or `SCALAR`).

For some instructions, such as divides, it is important to know when an operand is zero. This concept is modeled by the `bloat.tree.ZeroCheckExpr` class. It consists of an `Expr` to be checked.

4.5.4 Boolean Expressions

Class `bloat.tree.CondExpr` is an abstract class representing a conditional (i.e. yields a true or false value) expression. It is subclassed by `bloat.tree.InstanceOfExpr`. `InstanceOfExpr` represents the instanceof instruction and has an `Expr` to check against a certain `Type`.

4.5.5 Expressions That Define Local Variables

For some of the optimizations that BLOAT performs, it is important to know when local variables are defined. Expressions that define (i.e. assign to) variables are modeled with `bloat.tree.DefExpr`. `DefExpr` has a `java.util.Collection` of places in which the variable that is being defined is used. Each `DefExpr` also has a unique *version number* associated with it. This is the "SSA" number (see chapter 6) of the variable being defined.

`DefExpr` has an abstract subclass `bloat.tree.MemExpr` that represents instructions that access a memory location.

Referencing the Heap

Abstract class `bloat.tree.MemRefExpr`, a subclass of `MemExpr`, represents a group of expressions that reference a memory location (i.e. in the heap). It is subclassed by three concrete classes.

Class `bloat.tree.ArrayRefExpr`, a subclass of `MemRefExpr`, represents an expression that references an element in an array. It consists of expressions (`Expr`) representing an index into the array and the array itself, as well as the `Type` of elements in the array.

An expression that accesses a field of an object is modeled by the `bloat.tree.FieldExpr`, a subclass of `MemRefExpr`. It consists of an `Expr` representing the object being accessed and a `MemberRef` (see section 3.2.1) representing the field being accessed.

Accesses to a class's static fields are represented by `bloat.tree.StaticFieldExpr`, a subclass of `MemRefExpr`. A `MemRefExpr` consists of a `MemberRef` representing the static field that is accessed.

Referencing a Local Variable

Class `bloat.tree.VarExpr` is an abstract subclass of `DefExpr` that represents an expression that accesses local (or stack) variables. Each `VarExpr` has an integer index associated with it. `VarExpr` has two subclasses.

TypeADDRESS

`VarExpr` is subclassed by `bloat.tree.LocalExpr` to represent an expression that accesses (uses or defines) one of a method's local variables. It contains a boolean that determines whether or not the local variable was allocated on the stack. Note that `LocalExpr` implements `LeafExpr` (see section 4.7) and consequentially has no children nodes. `LocalExpr` has one interesting method, `isReturnAddress`, that determine whether or not the local variable being accessed contains a return address (`Type.ADDRESS`).

Class `bloat.tree.StackExpr` represents an expression that is stored on the JVM stack. Its index is the stack item that is being referenced. Index 0 corresponds to the bottom of the stack. Recall that some data types occupy more than one stack slot. So, a `StackExpr` with an index of 3 is not necessarily in the third stack slot from the bottom. This is important to keep in mind when working with the `dup` and instructions for persistence.

In order for a Java class to verify, the height of the operand stack must be the same and contain the same types for all paths to a given point in the program. Therefore no statement containing a stack variable (i.e. `StackExpr`) may be inserted, removed, or relocated in the program. This property hinders some optimizations, but is necessary.

4.6 Statements

Statements are nodes in expression trees that have no value associated with them. They just perform some action the result of which is not important. Statements are modeled with the abstract class `bloat.tree.Stmt`. A `Stmt` is essentially the same as a `Node`. `Stmt` has a number of concrete subclasses.

`bloat.tree.AddressStoreStmt` represents store a `bloat.cfg.Subroutine`'s (see section 5.3) address to a local variable using the `astore` opcode. Consequently, an `AddressStoreStmt` instance consists of a `Subroutine` representing the subroutine whose address is being stored. Because an address cannot be "reloaded", it has no value and therefore must be differentiated from `LocalExpr` (see section 4.5.5), a store to a variable that has a value.

`bloat.tree.ExprStmt` represents an expression whose value is not used (i.e. the expression is not nested). It consists of an `Expr`.

`bloat.tree.InitStmt` represents the initialization of some number of local variables (usually a method's parameters). It consists of an array of `LocalExpr`. Since values are assigned to, `InitStmt` implements the `Assign` interface and consequently has a `defs()` method that returns the array of `LocalExpr`.

`bloat.tree.LabelStmt` is a placeholder for a label (target of a jump) in an expression tree. It consists of a `Label`.

`bloat.tree.MonitorStmt` represents the monitorenter and monitorexit opcodes. A `MonitorStmt` has a kind (`ENTER` or `EXIT`) and an `Expr` representing the the `Object` whose monitor is being entered or exited.

`bloat.tree.SCStmt` represents a swizzle check (aswizzle opcode in the PJama virtual machine) on an element in an array. It consists of two `Exprs` representing the index of the element in the array to be swizzled and the array itself.

`bloat.tree.SRStmt` represents a range swizzle (aswizzleRange opcode in the PJama virtual machine) over a range of elements in an array. It consists of three `Exprs`: one for the lower value of the range, one for the upper value of the range, and one for the array itself.

`bloat.tree.StackManipStmt` represents opcodes that change the ordering of elements on the stack (e.g. dup and swap). It consists of an two arrays of `StackExpr` (section 4.5.5). One array represents the stack before the instruction is executed. The other, after the instruction is executed. It also has an integer type that determines what instruction (`SWAP`, `DUP`, `DUP_X1`, `DUP_X2`, `DUP2`, `DUP2_X1`, or `DUP2_X2`) the `StackManipStmt` represents. Because it defines values on the stack, `StackManipStmt` implements the `Assign` interface. Consequently, it has a method, `defs`, that returns the array of `StackExpr` representing the stack **after** the instruction has been executed.

4.6.1 Statements That Change Control Flow

The Java virtual machine has a handful of instructions that change a program's control flow. These instructions are modeled with classes that subclass the abstract `bloat.tree.JumpStmt` class. `JumpStmt` has a method, `catchTargets()`, that returns a `java.util.Collection` of `Block`s that begin exception handlers (the "catch targets") of any exceptions that can be thrown in the basic block that is terminated by the `JumpStmt`. `JumpStmt` has several subclasses.

`bloat.tree.GotoStmt` represents a jump to another basic block. It has a `Block` that represents the target of the jump.

`bloat.tree.JsrStmt` represents the jsr opcode. jsr jumps to a subroutine. Subroutines are used to implement the `finally` clause of a try-catch block. A `JsrStmt` consists of the `bloat.cfg.Subroutine` that is called and the `Block` of code that follows the jump.

`bloat.tree.RetStmt` represents the ret opcode that returns from a subroutine. It consists of the `Subroutine` from which control is returning.

`bloat.tree.ReturnExprStmt` models the areturn opcode which returns an `Object` from a method. `ReturnExprStmt` has an `Expr` that represents the `Object` being returned.

`bloat.tree.ReturnStmt` represents the return opcode which simplify returns from a method. It has no special data.

`bloat.tree.SwitchStmt` represents one of the switch instructions of the Java virtual machine. A `SwitchStmt` consists of an integer, index, (represented by an `Expr`) on which the switch is to be performed. An array of integers represents the interesting values of index. An array of `Block`, targets, represents the blocks corresponding to the interesting values. Finally, a default `Block` is provided, if the index is not interesting.

`bloat.tree.ThrowStmt` represents the athrow opcode which throws an exception. An instance of `ThrowStmt` consists of an `Expr` representing the exception object that is thrown.

If Statements

"If" statements are represented by the abstract `bloat.tree.IfStmt` class, a subclass of `Stmt`. Each `IfStmt` consists of two `Block`s representing the true and false targets, and an integer that specifies the kind of comparison operation (`EQ`, `NE`, `GT`, `GE`, `LT`, or `LE`) represented by the statement. In addition to having methods that access the true and false blocks, `IfStmt` has a method, `negate`, that negates it meaning.

The comparison of two expressions is represented by `bloat.tree.IfCmpStmt`, a concrete subclass of `IfStmt`. In addition to an operator kind, a true `Block` and a false `Block`, an `IfCmpStmt` has a left and right `Expr` whose values are compared.

When an expression's value is compared against zero, an instance of `bloat.tree.IfZeroStmt` is used. An instance of `IfZeroStmt` consists of a comparison constant, a true `Block`, a false `Block`, and an expression (`Expr`) whose value is compared with zero.

4.6.2 ϕ-Statements

When a control flow graph is transformed into *static single assignment* (SSA) form, special nodes called ϕ-statements are placed into the graph. ϕ-statements are placed at merge points in the program and represent a merge of certain variable information. ϕ-statements combine information from various paths in the control flow graph into a new piece of information.

BLOAT represents ϕ-statements with the abstract `bloat.tree.PhiStmt` class. Each `PhiStmt` has a `VarExpr` (see section 4.5.5) that represents the target of the ϕ-statement. Since a ϕ-statement assigns a value its target, it implements the `Assign` interface. `PhiStmt`'s `defs()` method returns its target `VarExp`.

There are two concrete subclasses of `PhiStmt`. `bloat.tree.PhiJoinStmt` represents a ϕ-statement inserted into a basic block to merge some number of variables. `PhiJoinStmt` has a `VarExpr` target, a `Block` in which it resides, and a `Map` of operands (which themselves are `Expr`s, usually `VarExpr`s) to the `Block`s in which they are assigned. Don't worry, this will all be explained later. It has methods to access the `PhiJoinStmt`'s operands.

Exception handling complicates the work of a ϕ-statement. The class `bloat.tree.PhiCatchStmt` is used to represent merging variables inside a catch block. In addition to a `LocalExpr` target, `PhiCatchStmt` has a list of operands (`LocalExpr`). `PhiCatchStmt` has methods that work with its operands.

4.7 Interfaces Used In the Expression Tree

Classes that implement the `bloat.tree.Assign` interface perform an assignment. An assignment implies that a definition of a variable occurs. Knowing where variables are defined is important for some of the optimizations that

BLOAT performs. The `Assign` interface has one method, `defs()`, which returns an array of `DefExprs` (see section 4.5.5).

`Assign` is implemented by `InitStmt` (defines local variables), `PhiStmt` (defines a ϕ-variable), `StackManipStmt` (defines a slot on the stack), and `StoreExpr` (defines a memory location).

The only place `Assign` is used is in the `isDef()` method of `DefExpr`. A test is made to determine if a `DefExpr` is nested inside a node that implements `Assign`.

The `bloat.tree.LeafExpr` interface has no methods. It simply denotes a class that does not have any children nodes. It is implemented by `ConstantExpr` and `LocalExpr`.

4.8 Constructing the Expression Tree

The class `bloat.cfg.FlowGraph` has a private method, `buildTreeForBlock`, that begins the process of building and expression tree for a basic block of code. After creating a new `Tree`, it iterates over the code in the method (`bloat.editor.Instructions` and `bloat.editor.Labels`). `buildTreeForBlock` then figures out things like which `Block` follows a jump instruction, or which `Subroutine` a `ret` instruction lies in, and then adds the instructions to the expression tree. The details of `buildTreeForBlock` are discussed in section 5.5.4.

4.8.1 Adding Instructions to the Tree

`Tree` has three public methods (all named `addInstruction`) that are used for adding instructions to an expression tree. One method is used for `jsr` and jump instructions and includes a `Block` parameter that specifies the `Block` that follows the jump. Another method is for the `ret` and `astore` instructions. This method has a `Subroutine` parameter that specifies a `Subroutine` in which the instruction may reside. All `ret` instructions reside in subroutines. An `astore` may store a subroutine's return address into a local variable. The third `addInstruction` method is for all other instructions. All three methods call the private `addInst` to do the bulk of the work of adding an instruction to the expression tree.

The one-parameter `addInst` method attempts to do some optimization of `dup` instructions[2]. Then it calls the two-parameter `addInst` that may save all of the expressions on the operand stack to stack variables or local

[2] I'm not too sure about this one, folks.

variables (`saveStack`). If that succeeds, the instruction is visited by the `InstructionVisitor`, the `Tree` that creates the nodes in the expression tree for the instruction.

4.8.2 Visiting Instructions and Building the Tree

Recall that `Tree` implements the `InstructionVisitor` interface. We will now focus on the details of the `InstructionVisitor` and how it builds the tree. Each `visit_`*opcode* method creates one or more expression tree nodes and in the process pushes and pops information off of the `Tree`'s operand stack (`OperandStack`, see section 4.3). Recall that each `visit_`*opcode* operates on an object of `Instruction` (see section 3.3.1).

Loading Data

Pushing Constants Instructions that push a constant from the constant pool (ldc and ldc_w) result in the creation of an instance of `ConstantExpr` whose value is the operand of the instruction and whose type is the type of the value. The `ConstantExpr` is pushed onto the operand stack.

Loading From a Local Variable Instructions that load data from a local variable (iload, lload, fload, dload, and aload) result in the creation of an instance of `LocalExpr` whose index is the index of the local variable from which the data is loaded. The type of the `LocalExpr` depends on the instruction. The `LocalExpr` is pushed onto the operand stack.

Loading From an Array Instructions that push an element of an array onto the stack (iaload, laload, faload, daload, aaload, baload, caload, and saload) result in the creation of an `ArrayRefExpr`. The array and index of the `ArrayRefExpr` are popped off the operand stack. The `ArrayRefExpr` is pushed onto the operand stack.

Storing Data

When storing data a `StoreExpr` is generated from a `MemExpr`. Sometimes we want the `StoreExpr` to be pushed onto the operand stack and sometimes we want it to be wrapped in an `ExprStmt` and treated as a statement and added to the `Tree`'s statement list. This decision is made in the private `addStore` method. If the opcode preceding a store is one of the `dup` instructions, the generated `StoreExpr` is pushed onto the operand stack. In all other cases the `StoreExpr` is represented as a statement.

Storing Basic Types Into Local Variables Instructions that store basic types into local variables (istore, lstore, fstore, and dstore) create a new instance of LocalExpr. The LocalExpr's index is gleaned from the operand to the instruction and its type is obtained by popping the operand stack. addStore is called.

Storing a Reference Into Local Variables The astore instruction stores an object reference to a local variable. If the reference is an Object, it is handled in the same manner as the basic types: A LocalExpr is created with the operand of the instruction and with the type of the object on the top of the operand stack. Then addStore is called.

However, the reference may also be a return address. This will occur when the Tree represents a block in a subroutine. An instance of AddressStoreStmt is created with the Subroutine in which the block resides and is added to the statement list.

Storing Into Arrays Instructions that store into arrays (iastore, lastore, fastore, dastore, aastore, bastore, castore, and sastore) result in the creation of an ArrayRefExpr. The ArrayRefExpr's array (Expr), index, and new value are popped from the operand stack. addStore is called.

Working With the JVM's Stack

Instructions that pop elements off of the JVM's stack (pop and pop2) cause Exprs to be popped off the operand stack and instances of ExprStmt to be created from the Exprs. The ExprStmt is added to the statement list.

TreeUSE_STACK

The Java virtual machine has several instructions for duplicating elements on the operand stack. These are the dup instructions. BLOAT can deal with the dup instructions in one of two ways. It can model the dup behavior with StackExprs and StackManipStmts (section 4.5.5 and section 4.6), or it can just forget about messing with the stack and used local variables. The USE_STACK flag determines which method is used.

If the USE_STACK flag is set, then a StackManipStmt is created to represent the changes that the dup instruction makes to the stack. Let's consider what happens when a dup_x1 instruction is encountered. The top element of the stack is duplicated and is placed two words below the top of the stack. The dup_x1 instruction can be represented as:

```
0 1 -> 1 0 1
```

Opcode	Transformation	Description
dup	`0 -> 0 0`	Duplicate top element of stack
dup_x1	`0 1 -> 1 0 1`	Duplicate top element of stack and put two down
dup_x2	`0 1 2 -> 2 0 1 2`	Duplicate top element of stack and put three down
	`0-1 2 -> 2 0-1 2`	
dup2	`0 1 -> 0 1 0 1`	Duplicate top two elements of stack
	`0-1 -> 0-1 0-1`	
dup2_x1	`0 1 2 -> 1 2 0 1 2`	Duplicate top two elements of stack and put three down
	`0 1-2 -> 1-2 0 1-2`	
dup2_x2	`0 1 2 3 -> 2 3 0 1 2 3`	Duplicate top two stack elements and put four down
	`0 1 2-3 -> 2-3 0 1 2-3`	
	`0-1 2 3 -> 2 3 0-1 2 3`	
	`0-1 2-3 -> 2-3 0-1 2-3`	
swap	`0 1 -> 1 0`	Swap the top two elements of the stack

Table 4.1: The Java Virtual Machines `dup` Instructions

The top two elements, `s1` and `s0`, are popped off the operand stack and placed into an array. Both `s1` and `s0` should be instances of `StackExpr`. An integer array is used to represent the transformation between the old stack (before the `dup_x1` instruction) and the new stack. For instance, the integer array for `dup_x1` is {`1, 0, 1`}. Using these two arrays, the private `manip` method, adjusts the elements of the `Tree`'s operand stack to reflect the execution of the instruction and adds a `StackManipStmt` to the statement list that represents the transformation. As Table 4.1 shows, care must be taken when dealing with wide data on the stack.

If the `USE_STACK` flag is not set, a `StackManipStmt` is not used. Instead, a local variable (`LocalExpr`) is used to represent the element of the stack. This causes more local variables to be used, but reduces the complexity of the expression tree.

For instance, when the `dup_x1` instruction is encountered the top two elements of the stack, `s1` and `s0` are popped. Two new local variables (`LocalExprs`), `t0` and `t1`, are created to represent `s1` and `s0`. Unless `s1` and `s0` happen to be equal to `t0` and `t1`[3], their values are stored using method

[3]This means that the top two elements of the stack were `LocalExprs` representing `s1`

addStore (see section 4.8.2). Finally, clones of t0 and t1 are pushed onto the stack in the appropriate order. For dup_x1 a clone of t1 is pushed, followed by a clone of t0 and a clone of t1. Again, things are complicated by wide stack elements.

Arithmetic Operations

Addition, Subtraction, and Multiplication Opcodes for handling addition, subtraction, and multiplication ($x$add, $x$sub, $x$mul) are handled similarly. The left and right operand expressions (Expr) are popped off the stack and an ArithExpr (see section 4.5.1) is created to represent the operation.

Division and Remainder The handling of the division and remainder opcodes ($x$div and $x$rem) involves popping the left and right operand expressions off the stack. Unless a float or double division operation (fdiv or ddiv) is being handled, a ZeroCheckExpr (see section section 4.5.3) is created for the right operand expression. The ZeroCheckExpr is used as the right operand for the ArithExpr which is then pushed onto the stack.

Negation The negation instructions ($x$neg) are represented by NegExprs. The operand of the NegExpr is popped from the stack.

Bit Shifting The bit shift instructions (ishl, lshl, ishr, lshr, iushr, and lushr) are represented by a ShiftExpr (see section 4.5.1) whose operands are popped off the stack.

Boolean Operations The boolean operation instructions (iand, land, ior, lor, ixor, and lxor) are modeled by ArithExprs whose operands are popped from the stack.

Incrementing The iinc has no corresponding Expr class. Recall that the iinc instruction has two operands, a local variable to increment and the amount by which to increment the local variable. Also recall that this information was encapsulated in the bloat.editor.IncOperand class (see section 3.3.1).

A LocalExpr is created from the IncOperand's local variable and a ConstantExpr is used to represent the amount by which to increment the local variable. An ArithExpr is created to perform the increment

---

and s0.

(or decrement, if the amount is negative). A `StoreExpr` is created to store the result of the `ArithExpr` into the `LocalExpr`. Finally, the `StoreExpr` is wrapped in an `ExprStmt` and is added to the statement list.

Comparing `floats` and `doubles` The fcmpl, fcmpg, dcmpl, and dcmpg instructions compare two `floats` (or `doubles`). If the left operand is greater than the right operand, an integer 1 is pushed on the stack. If the left operand is equal to the right operand, an integer 0 is pushed on the stack. If the left operand is less than the right operand, an integer -1 is pushed on the stack. The $x$cmpl instructions differ from the $x$cmpg instructions in the manner in which they handle NaN (not a number). The left and right operands are popped off the stack and an `ArithExpr` is constructed.

Changing Control Flow

Comparing With Zero The "if" instructions (ifxx, ifnull, and ifnonnull) compare a value with zero. The expression to test is popped off the stack. The true block is obtained from the `Tree`'s control flow graph and the if instruction's operand. The false block is the block following the block for which the `Tree` is constructed. From this information an `IfZeroStmt` (see section 4.6.1) is created an pushed onto the operand stack.

Comparing Two Expressions The "compare" instructions (if_$x$cmpop) compare two expressions and branch depending on the expressions's equality. The two expression are popped from the operand stack. The true block is obtained from the `Tree`'s control flow graph using the operand of the compare instruction. The false block is the block following the block that is modeled by the `Tree`. All of this information is used to create an `IfCmpStmt` which is added to the statement list.

Unconditional Jump The unconditional jump instruction, goto, is modeled with a `GotoStmt`. The destination block is obtained from the `Tree`'s control flow graph and the operand to the instruction. A `GotoStmt` is created with the destination block and added to the statement list.

Jump to a Subroutine The jsr instruction jumps to a JVM subroutine. The operand to a jsr is the "address" of the first instruction of the subroutine. The address of the instruction following the jsr is pushed onto the stack. When a jsr instruction is encountered, the `bloat.cfg.Subroutine` (see section 5.3) that is the target of the jump is obtained from the instruction's operand and the `Tree`'s control flow graph. A `JsrStmt` (see section 4.6.1) is created from the `Subroutine` and the block following the jsr instruction[4]. The `JsrStmt` is added to the statement list. Finally a `ReturnAddressExpr` (see section 4.5.1) is pushed onto the stack.

Return From a Subroutine The ret instruction returns from a subroutine. ret instructions are only encountered inside of subroutines. A `RetStmt` is created from the `bloat.cfg.Subroutine` in which the instruction resides. The `RetStmt` is added to the statement list.

Switch Instruction The tableswitch instruction jumps to a instruction associated with a index into a jump table. The lookupswitch looks up a key in a jump table and branches to the instruction associated with the key. Together these instructions are modeled with a `SwitchStmt`.

The index (or key) is popped from the operand stack. The operand to the instruction is an instance of `bloat.editor.Switch` (see section 3.3.1). The targets, as well as the default target, of the switch are obtained from the `Switch` and the `Tree`'s control flow graph. This information along with the integer values of the keys is used to construct a `SwitchStmt` that is added to the statement list.

Returning an Expression The $x$return instructions represent returning a value from a method. The value is popped off the operand stack (as an `Expr`) and is used to create a `ReturnExprStmt` which is added to the statement list.

Accessing Parts of a Class

Fetching a Field The getfield instruction fetches a field from an object. The operand to a getfield is an instance of `bloat.editor.MemberRef`. The object whose field is being fetched is popped off the stack. A `ZeroCheckExpr` (see section 4.5.3) is created to ensure that the object is non-null. The `ZeroCheckExpr` and the `MemberRef` are used to create a new `FieldExpr` (see section 4.5.5) which is then pushed onto the stack.

[4] Remember that any jump terminates a basic block. Therefore the instruction following the jsr is always in another `Block`.

Setting a Field The putfield instruction sets the value of an object's field. The field itself is represented as a `bloat.editor.MemberRef` that is the operand to the instruction. The object whose field will be set and the value to which it will be set (both `Exprs`) are popped off the operand stack. A `ZeroCheckExpr` is created to ensure that the object is non-null. Using this information a `FieldExpr` is created to reference the field. Finally, the value is stored into the field (`FieldExpr`) with method `addStore` (see section 4.8.2).

TypeVOID

Invoking Methods The invokevirtual, invokespecial, invokestatic, and invokeinterface instructions invoke a method. The private `addCall` method is used to create the appropriate instance of `CallMethodExpr` or `CallStaticExpr` to represent the call. The operand to the invoke instruction is a `MemberRef`. From the `MemberRef` the types of the parameters and the return type of the method are obtained. The types of the parameters are used to ensure that the parameters are valid as they are popped off of the operand stack. If the instruction is invokestatic, then an instance of `CallStaticExpr` is created to represent the method call. Else, an instance of `CallMethodExpr` is created. If the method's return type is not `Type.VOID`, then the `Call`*xxxxxx*`Expr` is pushed onto the stack. Else, it is wrapped in a `ExprStmt` and is added to the statement list.

Instantiating Objects

Creating a New Instance The new instruction creates a new object. The operand of the new instruction is a `bloat.editor.Type` (see section 3.2.1) that represents the class of the object to be created. A `NewExpr` is created with the type of the new object.

Creating a New Array The newarray and anewarray are used to create new arrays of basic types and objects, respectively. The type of the elements in is obtained from the `MemberRef` operand of the instruction. The size of the array is popped off the top of the stack. This information is used to create a `NewArrayExpr` that is pushed onto the operand stack.

Creating a Multidimensional Array The multianewarray instruction is used to create a new multidimensional array. Information about the array to be created is obtained from the operand of the instruction, a `bloat.editor.MultiArrayOperand`. Recall that a `MultiArrayOperand` contains the number of dimensions in the array and the `Type` of the elements in the array. The lengths of the array in each dimension are popped off of the operand stack. This information is used to create a `MultiArrayExpr` that is pushed on the operand stack.

Persistent Store Instructions

Updating Data In a Persistent Store The aupdate and supdate are used to update, respectfully, pointer and scalar values in a persistent store. A `UCExpr` is used to represent these checks. `UCExpr` is a little different from other expressions in that does not change the operand stack. The object on which the update check is being performed is obtained by "peeking" into the stack at a certain depth. The operand to the update instruction is the depth of the stack at which the object (`Expr`) resides. A `UCExpr` is created and replaces the object's `Expr` in the stack.

Swizzling an Element of an Array The aswizzle instruction is used to swizzle an element of an array. The array and the index into the array are popped from the operand stack. This information is used to create a `SCStmt` that is added to the statement list.

Swizzling a Range of Array Elements The aswizzleRange instruction is used to swizzle a range of elements in an array. The starting and ending indices of the range, as well as the array itself are popped off the stack. A `SRStmt` is created and is added to the statement list.

Other Instructions

Obtaining the Length of an Array The arraylength instruction gets the length of an array. The reference to an array (`Expr`) is popped off the stack and is used to make an `ArrayLengthExpr` which is popped onto the stack.

Throwing an Exception The athrow instruction throws an exception. The exception object to throw (`Expr`) is popped off the operand stack. It is used to create a `ThrowStmt` that is added to the statement list.

Casting The casting instructions ($x2y$) are modeled by a `CastExpr` whose operand is popped off the stack and whose type is the type to which the operand is cast.

Checking Casting The checkcast instruction checks if an object is of a given type. The object (`Expr`) is popped off the stack. The type to check against is obtained from the operand to the instruction. This information is used to create a `CastExpr` that is pushed onto the operand stack.

Determining Type The instanceof instruction determines if an object is of a given type. The type to check against is obtained from the operand to the instruction. The object (`Expr`) is popped off the operand stack. This information is used to create an `InstanceOfExpr` that is pushed onto the operand stack.

Entering and Exiting a Monitor The monitorenter and monitorexit are used to enter and leave an object's monitor. The object is popped from the stack and a `MonitorStmt` is created and added to the statement list.

4.9 Summary

BLOAT models Java instructions using expression trees. An expression tree consists of non-valued statements and expression that have a value. BLOAT expression trees are populated with nodes that represent operations such as arithmetic, method invocation, stack manipulation, and exception throwing. Expression trees are constructed by examining each instruction and simulating the Java Virtual Machine's operand stack.

Chapter 5

Control Flow Graphs

5.1 Background

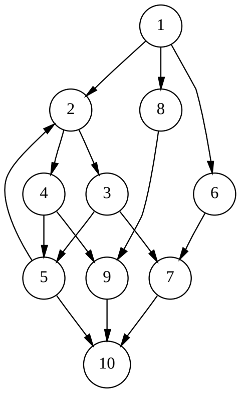
A *basic block* contains a sequence of instructions in which there is no change of flow control. That is, the first instruction in the block has a label associated with it (i.e. it is the target of a jump) and the last instruction in the block is a jump to another block. Basic blocks have the property that the flow of control can only enter at their first instruction and can only exit at their last instruction.

A *control flow graph* is a directed graph in which the nodes of the graph are basic blocks. In the control flow graph there is a directed edge from a block x to block y if the target of x's last instruction is the first instruction in y. The graph has two additional nodes, the entry block and the exit block. There is an edge from the entry block to any node from which the program (in BLOAT's case, a method) can be entered. Similarly, there is an edge from every block from which the program can be exited to the exit block.

A basic block x *dominates* another block y in a control flow graph, if all paths from the entry node to y pass through x. A block x *strictly dominates* a block y if x and y are not the same block. A block's *immediate dominator* is it closest strict dominator. This dominance relationship results in a *dominator tree*[1]. The root of dominator tree is the entry node (which has no immediate dominator). The parent of a node in the dominator tree is its immediate dominator.

Conversely, x *postdominates* y if all paths from the **exit** node to x pass through y in the *reverse* control flow graph. This leads to a *postdomina-*

[1]Note that a tree results because a node can have at most **one** immediate dominator.



Node 2 dominates nodes 2, 3, 4, and 5.
Its dominance frontier is nodes 2, 7, 9, and 10.

Figure 5.1: Dominance Frontier

tor tree in which a node has a postdominator parent and postdominator children.

The *dominance frontier* of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w. Basically, nodes in the dominance frontier have one parent that **is** dominated by x and at least one parent that **is not** dominated by x. An example of a dominance frontier is giving in figure 5.1. Similarly, there is a *postdominance frontier*.

A *loop* is a strongly connected component of a control flow graph. The *loop header* is the block in a loop that dominates all other blocks in the loop. A loop is *reducible* if its only entry point is at the loop header.

A *trace* of a control flow graph is an ordering of its blocks with the following two properties. The first is that blocks that end with a conditional jump are followed in the trace by the block that is executed when the condition is false. The second is that blocks ending with an unconditional jump are followed, where possible, by the block that is the target of the jump. Bytecode will typically be in trace form.

5.2 Basic Blocks

BLOAT represents basic blocks with `bloat.cfg.Block`. Since a `Block` represents a node in a graph (a control flow graph), it is a subclass of `bloat.util.GraphNode` (see section 2.2.1). Each `Block` knows the `Label` (see section 3.3.1) that begins it the control flow graph (`bloat.cfg.FlowGraph`, see section 5.4) in which it is a node. The instructions in a basic block are represented by an instance of `bloat.tree.Tree` (see section 4.4), an expression tree.

Each `Block` knows both its parent and children in the dominator tree, parent and children in the postdominator tree, and its dominance and post-dominance frontier. The dominator and dominance frontier information is computed using the `DominatorTree` and `DominanceFrontier` classes (see section 5.6.1 and section 5.6.2).

BlockNON_HEADER BlockREDUCIBLE BlockIRREDUCIBLE

There are three "types" of basic blocks: `NON_HEADER`, `REDUCIBLE`, and `IRREDUCIBLE`. A `NON_HEADER` block is a block that is not the header block of a loop. A `REDUCIBLE` block is the header of a loop that can be reduced and an `IRREDUCIBLE` block is the header of a loop that cannot be reduced.

`Block` has a number of methods that grant access to its expression tree, control flow graph, type, header, and parent and children in various trees.

5.3 Exceptions

Exceptions are a pain. In the classfile, exceptions are represented by the exception table, a table consisting of a range of instructions over which the exception may be thrown, the instruction that begins the exception handler, and the type of the exception caught. At the reflection level, BLOAT models exceptions with `bloat.reflect.Catch` (see section 1.1.3). In the editing level, exceptions are modeled with `bloat.editor.TryCatch` (see section 3.3.1).

Now, exceptions are modeled with `bloat.cfg.Handler` and `bloat.cfg.Subroutine` classes. `Handler` consists of a `Set` of protected `Blocks` (the "try" blocks), a catch `Block`, and the `Type` of exception that is caught by the catch block.

Recall that `finally` blocks are implemented using Java Virtual Machine *subroutines* [LY96]. The jsr ("jump to subroutine") instruction is used to enter a subroutine. The jsr pushes the address of the instruction following it onto the JVM stack. This instruction is where control will return once the subroutine has completed. The first instruction of the subroutine is an astore that stores the return address into a local variable. The subroutine then goes about its merry way executing whatever code is in the `finally` block. When it is done, the subroutine executes a ret instruction whose operand is the local variable in which the return address is stored.

BLOAT models a JVM subroutine with the `bloat.cfg.Subroutine` class. A `Subroutine` knows the `FlowGraph` (i.e. method, see section 5.4) in which it resides, its (the subroutine's) entry and exit `Block`s, and the `bloat.editor.LocalVariable` (see section 3.3.1) in which its return address is stored. The local variable is set when an `astore` instruction in a subroutine is visited by `Tree` (an `InstructionVisitor` see section 4.8.1 and section 4.8.2).

Additionally, each `Subroutine` has a list of `Block` pairs that represent the block in which the subroutine is called (ends in a `jsr`) and the corresponding block that is executed upon return from the subroutine (begins at the return address). These `Block` pairs are referred to as the "paths" and are constructed by the `buildBlocks` method of `FlowGraph` (section 5.5.1). `Subroutine` has methods to add and remove paths.

5.4 Modeling Control Flow Graphs

BLOAT models a control flow graph with `bloat.cfg.FlowGraph`, a class that extends `bloat.util.Graph` (see section 2.2.1) and represents a Java method. Each control flow graph is associated with a method via a `bloat.editor.MethodEditor` (see section 3.3.2). The nodes of the control flow graph, basic blocks, are instances of `Block`. Each `FlowGraph` has three special blocks. The *source block* is the control flow graph's entry block. Control enters the `FlowGraph` through the source block. The *sink block* is the control flow graph's exit block. Control exits the `FlowGraph` through the sink block. The *init block* contains code that handles the initialization of method parameters, etc.

In addition to the three special blocks and the method's `MethodEditor`, some other information about the control flow graph is maintained. A list of all the `Block`s in the control flow graph, called the "trace" is maintained. A `Graph` called the "loop tree" represents any loops occurring in the control flow graph and their nesting (see `buildLoopTree` in section 5.6.7).

A `FlowGraph` maintains some information pertaining to the method it models. Most of this information is related to the exceptions that are handled in the method. `FlowGraph` maintains a mapping between a subroutine's entry `Block` and its `Subroutine`, a list of all of the `Block`s that begin exception handlers, and mapping between the first `Block` of an exception handler and its `Handler` object (see section 5.3).

5.5 Constructing the Control Flow Graph

5.5.1 Building Basic Blocks

The private `buildBlocks` method of `FlowGraph` creates basic blocks from the code of the method that the control flow graph models. It first obtains a list of the instructions (`bloat.editor.Instructions` and `bloat.editor.Labels`, see section 3.3.1 and section 3.3.1) from the `MethodEditor`. It examines every `Label` and if it starts a basic block, a new `Block` is created with that `Label` and added to the `FlowGraph` via a call to `newBlock`. This new block is added to the `FlowGraph`'s trace.

The method's code is again examined from the beginning. Several things may occur when a `Label` that starts a basic block is encountered. First, a mapping between `Label`s that begin basic blocks and their offset in the code (`Integer`) is maintained. If the last `Instruction` that was encountered was a jsr, the `Subroutine` corresponding to the operand of the jsr (a `Label`) is obtained. A "path" (see section 5.3) is added to the `Subroutine` from the `Block` that contains the jsr to the `Block` that starts with the `Label` being examined (i.e. the block to which the subroutine will return).

When a jsr `Instruction` is encountered and the `Subroutine` target of the jsr has not yet been encountered, a new `Subroutine` is created. By examining the operand of the jsr instruction, the `Block` that begins the `Subroutine` is obtained. A mapping between this `Block` and its `Subroutine` is maintained.

5.5.2 Dealing With Exception Handlers

Before the expression trees for the basic blocks are constructed, the `buildTrees` method performs some processing of try-catch blocks. Each of the `MethodEditor`'s `bloat.editor.TryCatch`es (see section 3.3.1) is examined. Two `Block`s are created for each `TryCatch`. The first `Block`, the "catch block", is the target of the exception handler. It saves the exception on the JVM stack. Recall that the athrow instruction pushes the exception object back onto the stack. We need to model this behavior. This block is also created so that the handler target cannot possibly be a loop header.

The second `Block`, the "catch body", contains the code that handles the exception. A mapping from the catch block to the catch body is maintained. The catch block's position in the code is the same as the catch body's. An edge in the control flow is added from the catch block to the catch body. The catch block is added to the trace of the control flow graph just before the catch body.

An expression `bloat.tree.Tree` is created for the catch block. The `Tree` consists of a `StoreExpr` (see section 4.5.1) that stores a `CatchExpr` (see section 4.5.1) into a `StackExpr` (see section 4.5.5) followed by a `GotoStmt` (see section 4.6.1) that jumps to the catch body.

A new `Handler` (see section 5.3) is created for the `Type` of the `TryCatch`. A mapping between the catch block and its `Handler` is maintained. Then, every `Block` in the control flow graph is examined. If the block's offset in the code lies between the start and the end of the `TryCatch`, then the block is a protected block and we add it to the `Handler`'s list of protected blocks.

Edges are added from the control flow graph's source block to its init block, its source block to its sink block, and its init block to the first block of code. Then the private `buildSpecialTrees` method is called to construct the expression trees for these "special" blocks (i.e. sink, source, and init). New `bloat.tree.Trees` are created for the special blocks. If there is code in the method being modeled by the `FlowGraph`[2], then in `initLocals` method of the init block's `Tree` is called. Recall that `initLocals` method initializes a method's parameters (represented as local variables) by adding a bunch of `InitStmts` to a `Tree`. The local variables for the method modeled by the `FlowGraph` are obtained by calling `FlowGraph`'s private `methodParams` method which constructs an `ArrayList` of `LocalExpr` from the `MethodEditor`. A `goto Instruction` that jumps to the first block in the method (control flow graph) is added to the init block's expression tree. Finally, `addHandlerEdges` is called for the init block.

Adding Edges to Exception Handlers

Recall that if some instruction in a basic block throws an exception, flow control will be transferred to the exception handler. Thus, there must be an edge in the control flow graph from the block that may throw an exception to the first block of the exception handler. `FlowGraph`'s private `addHandlerEdges` method adds these edges. First, it iterates over all of the `Handlers` that the `FlowGraph` knows about. If the block (that many throw an exception) in question or any of its immediate successors lies inside the protected region of the `Handler`, then we need to process it. The "catch block" (first block in the exception handler) for the `Handler` is obtained. This block is added to the list of "catch targets" of the `JumpStmt` that terminates the block in question. An edge in the `FlowGraph` is added between

[2]Note that if there is no code in the method, `buildSpecialTrees` would have been called long ago and none of this malarky with the exception handlers would have been necessary.

the block that may throw an exception and the catch block. If the expression tree for the "catch body" associated with the catch block has not yet been created, do so[3] by calling `buildTreeForBlock` (see section 5.5.4). Finally, `addHandlerEdges` is called recursively for the catch block in case there are exceptions handled within exception handlers.

5.5.3 A Quick Regroup

Okay, what have we done so far? We've added `Block`s to the `FlowGraph` for every `Label` in the code that begins a basic block. We've created `Subroutine`s to represent the subroutines in the method being modeled by the `FlowGraph`. We've created "catch block" and "catch body" `Block`s and expression trees for each exception handler. We've also dealt with the sink, source, and init blocks, adding edges and creating expression trees where necessary. Finally, we've added edges from blocks that may throw exceptions to their exception handlers.

5.5.4 Building Expression Trees

Expression `Tree`s are created by the `buildTreeForBlock` method. `buildTreeForBlock` generates expression trees for a given `Block` and all `Block`s reachable from that `Block` that do not already exist. It has already been used to generate expression trees for the init block and the exception handler blocks. The last thing the `buildTrees` method does is invoke `buildTreeForBlock` on the first block in the method with an initial stack corresponding to the operand stack of the init block.

If an expression tree does not already exist for the `Block`, `buildTreeForBlock` creates a new `Tree` for the `Block` using the current contents of the operand stack. It then iterates over the `Block`'s code (`Instruction`s and `Label`s) obtained from the `MethodEditor`. An initial pass is made over the code. If a jsr or a conditional jump `Instruction` is encountered, the code is searched for the target of the jump. This is the "next block".

Another pass over the code is made. Instructions are handled as follows.

astore The instruction is added to the expression tree using the `addInstruction` method of `Tree` (see section 4.8.1) making note of the current `Subroutine` that we are in[4].

[3]Note that the initial `OperandStack` of the exception handler contains an object of `Type.THROWABLE` representing the exception object that was pushed on the stack by the athrow instruction.

[4]Recall that the astore may store the return address of the `Subroutine`.

ret Make note of the fact that the `Block` is the exit block for the current `Subroutine`. The instruction is added to the `Tree` via a call to `addInstruction`. Edges in the control flow graph are added from the exit block of the `Subroutine` to the block that is executed following the ret from the `Subroutine`. These blocks are determined using the `Subroutine`'s "paths" (see section 5.3).

throw or return instruction An edge in the control flow graph is added from the block to the sink block after the instruction is added to the tree.

jsr The instruction is added to the tree noting the next block. `buildTreeForBlock` is then recursively called to build and expression tree for the target of the jsr, a `Subroutine`. An edge in the `FlowGraph` is added from the block containing the jsr to the beginning of the `Subroutine`. If the `Subroutine`'s exit block is known, code is generated for the next block using the operand stack of the `Subroutine`'s exit block. An edge from the `Subroutine`'s exit block to the next block is added in the `FlowGraph`.

Conditional Jump The instruction is added to the tree noting the next block. An edge is added from the block to the target of the jump (the "true" block). An expression tree is generated for the target block via a recursive call to `buildTreeForBlock`. An edge is also added between the block and the next block (the "false block" because the blocks are in trace order). An expression tree is also generated for the next block.

Switch The instruction is added to the expression tree. The `bloat.editor.Switch` object corresponding to the instruction's operand is obtained. Through the `Switch` the targets of the switch statement are obtained. An edge is added from the block containing the goto to each target block. An expression tree is generated for each target block.

When a `Label` that starts a block is encountered, a new goto instruction is added to the tree. An edge is added from the block to the block starting with the `Label` and an expression tree is generated for the block starting with the `Label`. After all of the `Instructions` and `Labels` have been processed, `addHandlerEdges` (see section 5.5.2) is called to add edges from blocks that may throw exceptions to the appropriate exception handlers.

Once the `Blocks` and expression `Trees` have been built, the `removeUnreachable` method of `Graph` (see section 2.2.1) is called to remove `Blocks` in the `FlowGraph` that are not reachable from a pre-order traversal. When blocks are removed from the control flow graph, the `Label`s that start those blocks no longer label anything that is valid. The `saveLabels` method saves these `Label`s by adding them as `LabelStmt`s to the init block and marking them as no longer starting a block.

5.6 Initializing the Control Flow Graph

After the nodes of a control flow graph have been built, the graph must be initialized. The initialization process involves computing the dominance relationships among the nodes, building the loop tree, splitting reducible and irreducible loops, peeling loops, removing critical edges, and inserting stores after conditionals and before protected regions. `FlowGraph`'s `initialize` method performs these tasks and relegates most of the work to other methods and classes.

5.6.1 Building the Dominator Tree

Recall that a control flow graph's dominator tree is the tree rooted at the entry node where the parent of a node is its immediate dominator. The `bloat.cfg.DominatorTree` class has one public method, `buildTree`, that does the work of constructing the dominator tree for a control flow graph. The Purdum-Moore [PM72] algorithm is used.

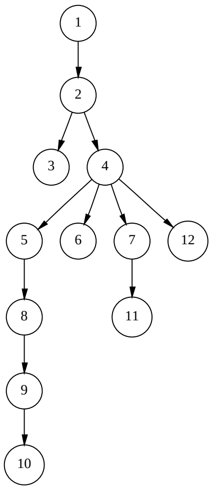
First, the private `insertEdgesToSink` method is called to create a mapping between the sink node(s) of the control flow graph and its immediate predecessors. In the case of finding postdominators, the mapping is between the sink node's predecessors and the sink node[5].

The dominance relationship among the nodes in the control flow graph is conceptually represented by a two-dimensional bit matrix. If node x dominates node y, then bit (x, y) will be set in the matrix. Initially, all of the bits in the matrix are set, except for the row corresponding to the root node. The root node's row has one bit set, the bit corresponding to the root (i.e. the root dominates itself).

Then every block in the graph is examined in order and their dominators (not **immediate** dominators, that will be done later) are computed.

$$Dominators[n] = n \cup (\bigcap_{p \in pred[n]} Dominators[p])$$

[5] I'm not too sure why this is necessary. The behavior of this algorithm appears to imply that the sink node is not connected to the rest of the graph. However, as far as I can tell, it is.



Block	Is dominated by
1	
2	1
4	1 2
5	1 2 4
8	1 2 4 5

Figure 5.2: A Dominator Tree

This dominator information is then used to compute each node's immediate dominator. The immediate dominator of a block x is computed by removing all blocks from x's dominator set that themselves dominator one of x's dominators[6]. Let's go through an example using the dominator tree in figure 5.2.

Let's say we want to find the immediate dominator of 8. We start with its set of dominators, $\{1\ 2\ 4\ 5\}$. We examine each block in this set and remove its dominators. So, we remove block 1's dominators (none). We remove block 2's dominators (1) leaving $\{2\ 4\ 5\}$. We remove block 4's dominators leaving $\{4\ 5\}$. Finally, we remove block 5's dominators leaving $\{5\}$ which is block 8's immediate dominator.

After ensuring that a block has only one immediate dominator, `buildTree` determines each block's immediate dominator and notifies the block using `bloat.cfg.Block`'s `setDomParent` method (see section 5.2).

5.6.2 Computing the Dominance Frontier

`bloat.cfg.DominanceFrontier` calculates the dominance and postdominance frontiers of the nodes in a `FlowGraph`. The public static method

[6] Is there a nicer way to say this?

`buildFrontier` is called to calculate the frontiers. However, the actual work is performed by the private `calcFrontier` method.

A `Block` n's dominance frontier is the union of two sets. The first set consists of the blocks in the dominance frontier of the nodes that n dominates that themselves are not dominated by n's immediate dominator. This set is calculated by iterating over the blocks that n dominates and recursively determining their dominance frontiers. If n is not the immediate dominator (i.e. parent in the dominator tree) of a block x in one of these dominance frontiers, then x is in the dominance frontier of n.

The second set consists of the successors of n (in the control flow graph) that are not strictly dominated by n. `calcFrontier` maintains an array of `Block`s that represents the a block's dominance frontier. The array is indexed by the per-order index of its blocks. Presumably, an array is used so that no block is added to the dominance frontier twice.

Iterated Dominance Frontier

The *iterated dominance frontier* for a set of nodes in a control flow graph is the union of the dominance frontiers of all the nodes in the set. It is used to determine the nodes into which ϕ-nodes should be inserted during conversion of a control flow graph into static single assignment form (see section 6.1.1). The iterated dominance frontier for a given set of `Block`s is calculated by `FlowGraph`'s `iteratedDomFrontier` method.

5.6.3 Preparing for ϕ-statement Insertion

Eventually, we'll be inserting SSA ϕ-statements into the control flow graph. In order to ensure that the ϕ-statements are inserted correctly, we have to examine some blocks. We must make sure that no block is more than one of: a catch block (first block in an exception handler that saves the exception), the entry block of a subroutine, or the target of a subroutine return. If a block has two or more of these properties, more than one SSA ϕ-statement could be placed in the block.

Luckily, catch blocks and are mutually exclusive with subroutine return targets and subroutine entry blocks, we only need to ensure that a block is not an entry block of a subroutine and a return target.

`FlowGraph`'s `splitPhiBlocks` method looks at the entry blocks of all of the control flow graph's subroutines' (`entrySub` in figure 5.3). If an entry block, `splitBlock`, is also the target of a subroutine (`returnSub`) return, then it needs to be "split". Two new `Block`s are created: the `newEntry` and

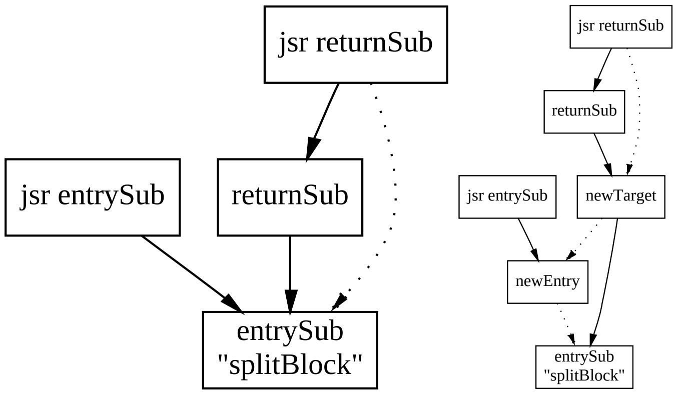


Figure 5.3: Splitting ϕ-blocks

`newTarget` blocks. Both of these blocks jump to `splitBlock`. The edges in the control flow graph are adjusted so that all blocks that end in a `jsr` to `entrySub` now point to `newEntry` and that all return targets of `returnSub` jump to `newTarget`. This process is illustrated in figure 5.3. Dotted arrows represent the trace order of blocks.

Replacing Blocks

`splitPhiBlocks` is an example of a place where one `Block` in the control flow graph needs to be replaced with another. This process is facilitated by the `bloat.cfg.ReplaceTarget` class. `ReplaceTarget` is a `bloat.tree.TreeVisitor` (see section 4.2) that replaces a target block of a jump or `ret` with another block. The targets of `JumpStmts`, the entry blocks of the targets of `JsrStmts`, the destinations of `RetStmts`, the targets of `GotoStmts` and `SwitchStmts`, and the true and false targets of `IfStmts` are replaced.

5.6.4 Splitting Irreducible Loops

The loop optimizations that BLOAT performs work on *reducible* loops. Recall that a loop is reducible if it has a single entry. The loop *header* is the block that dominates all blocks in the loop. An *irreducible* loop has no one block entry that dominates all the blocks in the loop. The block chosen as the entry block of an irreducible loop depends on the path taken by a depth-first ordering of the control flow graph.

Paul Havlak [Hav97] gives an algorithm that maximizes the number of **reducible** loops in a graph by splitting blocks that could be both the header of reducible and irreducible loops. A *back edge* is an edge in the control flow graph whose source is a successor of its destination. A back edge defines a loop for which its destination is the header. A *reducible backedge* has a destination that dominates the source. Havlak's algorithm guarantees that every reducible backedge goes to the header of a reducible loop. This property maximizes the number of reducible loops in the control flow graph and is performed by adding empty blocks such that no reducible backedge shares a destination with an irreducible backedge.

Havlak's algorithm is implemented in the private `splitIrreducible-Loops` method of `FlowGraph`. It iterates over all of the blocks in the control flow graph. If a block dominates one of its predecessors, then it is a reducible back edge. All other incoming edges (irreducible backedges) are marked to be split. The actual work of splitting an edge is done with the private `splitEdge` method.

`splitEdge` first ensures that no edge involving in the source or sink blocks can be split. A new `Block` is created and is placed before the destination of the edge. The expression tree for the new block is just a `goto` to the destination block. A `ReplaceTarget` (see section 5.6.3) is used to adjust edges, etc. if the destination block is the target of a `JumpStmt`, etc. Edges are added from the source block to the new block and from the new block to the destination block. The old edge from the source block to the destination block is removed. Later optimization may move code from the destination block into the new block. So, if the destination block is a protected block, then the new block must also be a protected block. Thus, `Handler` (see section 5.3) objects, et. al. must be adjusted accordingly.

5.6.5 Splitting Reducible Loops

`splitReducibleLoops` ensures that each loop has a unique header block, by splitting loop headers such that no reducible backedge shares a destination with another reducible backedge. It iterates over all blocks in the control flow graph and notes the reducible backedges.

For each block that is the destination of a reducible backedge, its predecessor with the lowest pre-order (depth first) index, `min`, is located. The edge from `min` to the block in question, `header`, is split. All other reducible backedges incident on header are adjusted to point to the new block. This process is repeated on the new block until the new block is the target of only one reducible backedge.

5.6.6 Determining the Types of Blocks

The private `setBlockTypes` method of `FlowGraph` iterates over every `Block` in the control flow graph. It uses an algorithm presented in [Hav97] to determine whether a block is a `NON_HEADER`, `REDUCIBLE`, or `IRREDUCIBLE`. Initially, each block's loop header is set to the source block (except for the source block whose header is null) and a list of back edges and non-back edges is assembled.

The blocks are again iterated over in reverse pre-order so that the innermost loops are visited first. A `bloat.util.UnionFind` (see section 2.2.2) is used to store the indices that represent blocks in the various loops. For each loop header, the back edges are followed to construct the body of the loop. If one of the blocks in the loop body is not a descendent of the loop's header, then there is another entry path into the loop, and the loop (and thus its header) is irreducible. The blocks in the loop are merged (unioned) into the header's set in the `UnionFind`. To prevent further agony at the hand of exceptions, all loops that contain `jsr` or catch blocks are labeled as irreducible.

5.6.7 Building the Loop Tree

A *loop tree* represents the nesting hierarchy of loops in a control flow graph. Each node in the loop tree is an instance of `LoopNode` (that extends `bloat.util.GraphNode`, see section 2.2.1), a private class in `FlowGraph` that contains a header `Block`, the depth and the level of the loop, and the `Blocks` that comprise the loop. Each node is the loop tree is associated with its header block.

The root of the loop tree is the source block of the control flow graph, itself a header block. The blocks in the control flow graph are iterated over. Each block is added to the loop tree of its header block. If the block itself is a header block, a new loop tree node is created for it. An edge in the loop tree from the outer loop node to the inner loop node is created.

Once the loop tree has been constructed, the depth and level of each node is calculated. The root node of the loop tree has depth 0. The leaf nodes of the loop tree have level 0. Depth and level are calculated by a pre-order and post-order traversal of the loop tree, respectively.

5.6.8 Peeling Loops

We would like to move loop invariant code out of a loop. However, we can only evaluate an expression that has side effects in the context in which it occurs. For example, if an expression may thrown an exception, we must guarantee that all preceding expressions that may thrown exceptions are evaluated first.

Loop peeling copies the first iteration of a loop, causing it to be executed before the remaining iterations. Loop peeling also results in *loop inversion* whereby a loop's condition is placed at the end of the loop (i.e. converts a "while" loop into a "do-while" loop preceded by a condition). Note that neither loop peeling nor loop inversion can be performed on irreducible loops. Because loop peeling can result in a significant increase in code size, it is only performed on the innermost loops (i.e. with level 0). Additionally, only loops that contain code that has side effects and can be hoisted are peeled.

BLOAT performs loop peeling in the private `peelLoops` method of `FlowGraph`. The class variable `PEEL_LOOPS_LEVEL` determines the maximum (loop nesting) level at which loops can be peeled. Recall that the innermost loops have level 0. There are two class constants, `PEEL_NO_LOOPS` and `PEEL_ALL_LOOPS` that have obvious meaning.

`peelLoops` first makes a list of all blocks in the control flow graph in which an exception could occur and can be hoisted. Exceptions can occur in `CastExpr`s, `ArithExpr`s, `ArrayLengthExpr`, and `FieldExpr` when their operands are `LeafExpr`s.

The nodes in control flow graph's loop tree (see section 5.6.7) are visited in post-order (i.e. innermost loops are visited first). A list of loops to be peeled is assembled. Irreducible loops as well as the outermost loop cannot be peeled. The loops that are candidates for peeling are examined. If a block in the loop contains an expression that can be hoisted (and the peeling level has not been exceeded), then the loop can be peeled. If a loop cannot be peeled, it may still be able to be inverted. As long as loop's header has an edge to a block that is not in the loop, then it can be inverted.

A list of blocks that may exit the loop (i.e. the blocks that may thrown an exception and the blocks that have a successor that lies outside the loop) is assembled. By examining the predecessors of the blocks in this list, we determine the blocks in the loop that need to be copied. Copies of blocks are made with the private `copyBlock` method. `copyBlock` simply creates a new `Block` with an expression tree that has the same initial stack as the original block. A clone of all the statements in the block (except for any `LabelStmt`s, see section 4.6) is added to the new block.

The copy of the loop is added to the trace of blocks of the control flow graph after the "latest" (i.e. has the highest pre-order index) predecessor of the loop header. Edges are added between the blocks in the copied loop to duplicate the behavior of the original loop. Finally, edges entering the loop are adjusted to enter the peeled loop instead.

5.6.9 Removing Critical Edges

A *critical edge* is an edge from a block with more than one successor to a block with more than one predecessor. Critical edges can hinder code motion and should be removed. Splitting critical edges creates a block in which code can be placed during partial redundancy elimination and when translating the control flow graph back from static single assignment form. Critical edges often occur from a block inside a protected region to a block in an exception handler. These edges cannot be split without creating a new exception handler. So, they are not split and are given special treatment during PRE and SSA destruction.

`FlowGraph`'s private `removeCriticalEdges` method constructs a list a critical edges in the control flow graph. Edges whose destination blocks are inside subroutines or exception handlers, or edges whose destination is the sink block are ignored. All other edges whose destination has more than one predecessor and whose source has more than one successor are added to the list of critical edges. `splitEdge` (section 5.6.4) is called to insert a block between the source and destination blocks of the critical edge. Splitting the edge, in turn, removes the critical edges form the control flow graph.

5.6.10 Inserting Stores after conditional statements

Some conditional statements allow us to make certain assertions about expressions. Consider the following code.

```
if(a+b == c+d)
  X
else
  Y
```

Knowing that `a+b` indeed equals `c+d` can help when performing constant and copy propagation. We can add a store (assignment statement) that can be used in constant and copy propagation after the conditional to represent the equality.

```
if((e = a+b) == (f = c+d))
  e = f
  X
else
  Y
```

This transformation is only performed when the compared expressions are non-leaf and are not reference types. Consider the following example that involves reference types.

```
class A {};
class B extends A { void foo(); }

A a = someA();   // Returns an instance of A
B b = someB();   // Returns an instance of B

if(a == b) {
  b.foo();
}
```

If we were to insert an assignment after the `if`, the type information would be incorrect.

```
if(a == b) {
  b = a;           // b now has type A, not B
  b.foo();
}
```

`FlowGraph`'s private `insertConditionalStores` method does the work of adding the assignment statements to the conditionals. It examines the last statement in every block in the control flow graph. Recall that conditional statements end blocks because they cause a change of control flow.

If the last statement in a block is an `IfCmpStmt`, then the following occurs. If the true and false targets of the if statement are the same, then do nothing. This should not occur because critical edges were removed. If the comparison being made is an equality (`IfStmt.EQ`), then any assignment statement will be placed in the "true" target. Conversely, if the comparison being made is an inequality (`IfStmt.NE`), then any assignment statement will be placed in the "false" target. If any other comparison is being made, the conditional is ignored.

The conditional (equality or inequality) has a left and a right expression. If either expression is **not** a leaf expression (see `LeafExpr`, see section 4.7), the it is replaced with a `StoreExpr` (see section 4.5.1) that stores the expression into a new local variable (e.g. replace `a+b` with `e = (a+b)`). An assignment to the local variable is prepended to the expression tree of the (true or false) target block (e.g. add `e = f` to the target block).

The process for handling the case when the last statement in a block is an `IfZeroStmt` is similar. However, we only need to be concerned with the left expression because we know that the right expression is 0 or `null`. If the left expression is not a reference type and is non-leaf, it is replaced by an assignment to a new local variable. The left expression (now a local variable reference) is then examined. If it is an integer, then it will be assigned 0, else it will be assigned `null`. The assignment is prepended to the target block.

If the last statement in a block is a `SwitchStmt`, certain assertions about the integer index variable may be made in the "target blocks". For instance:

```
switch(index) {
  case 0:
    index = 0;
    ...
    break;

  case 4:
   index = 4;
   ...
   break;
}
```

Note that the assertions cannot be made when a target corresponds to multiple index values.

```
switch(index) {
  case 0:
    index = 0;
    ...
    break;

  case 1:
  case 2:
    index = 1;
    index = 2;    // WRONG!!
    ...
    break;
}
```

The targets of a `SwitchStmt` that are not used for multiple index values have a assignment to index prepended to them. The process is similar to that for `IfCmpStmt` and `IfZeroStmt`.

5.6.11 Inserting Stores Before Protected Regions

To facilitate code generation of `PhiCatchStmts`, statements that copy local variables are inserted before jumps to protected blocks. This ensures that locals used by the jump statement are not redefined. `FlowGraph`'s private `insertProtectedRegionStores` method compiles a list of blocks whose last statement is a jump to a protected block.

`insertProtStores` is called to do the work of inserting the copy statements to the blocks. It maintains an array of the defining expressions (`LocalExprs`) that define local variables. Each expression (`Expr`) in the jump statement is saved to a stack variable (`StackExpr`). For each block that ends in a jump to a protected block, a statement that makes a copy of each local variable in use is inserted before the jump. This process starts with the source block and is repeated for all of the blocks that are dominated by the block in question.

5.6.12 Verifying the Correctness of the Control Flow Graph

The vast majority of what BLOAT does involves changing the control flow graph. A control flow graph is verified to ensure that it is still consistent and correct after a transformation. `VerifyCFG`, a subclass of `bloat.tree.TreeVisitor`, traverses a `FlowGraph` and performs various checks on its nodes. While checking a control flow graph, `VerifyCFG` keeps track of the `Block` in which it expects expression tree nodes to reside, the expected parent block of expression tree node being checked, all of the expressions in the control flow graph that are uses of a variable, and all of the nodes in the expression tree that have been visited. Because value numbers may not have been assigned yet, verifying them is optional.

Verifying a `FlowGraph` involves examining its basic blocks and expression trees. It is checked to make sure that all uses of variables defined in the control flow graph reside within the `FlowGraph`.

When a `Block` is checked, it is verified that it is indeed in the control flow graph. If the block begins an exception handler, then it is ensured that all of the protected blocks have edges to the handler block. It is also verified that each of the block's successors has a corresponding predecessor and vice versa.

Statements that involve a change in control flow all have targets that must be verified. A list of targets for each `RetStmts`, `JsrStmts`, `SwitchStmts`, `IfStmts`, and `GotoStmts` is compiled. The private `verifyTargets` method is called to ensure that the number of targets equals the block's number of successors, that the targets all reside in the control flow graph, and that every target is a successor of the block.

When a `StoreExpr` is verified, if desired, its value number is checked to make sure that it is not -1. Its block and parent `Node` are compared against the expected values. If the `StoreExpr`'s type is `VOID`, then it is verified that it is not nested in any other expression (i.e. its parent node is an `ExprStmt`).

The children of of `Nodes` are verified to make sure that they are correct. If desired, the value numbers of `Exprs` are checked to make sure that they are not -1.

`VarExpr` are verified to ensure that they either define a local variable, are defined by another expression, or are the child of a `PhiStmt` (a ϕ-operand).

5.6.13 Committing Changes to the Control Flow Graph

Once a control flow graph has been modified by various optimizations, its changes are committed back to its `MethodEditor` using the `commit` method. First, new bytecode for the method modeled by the control flow graph is generated by a `bloat.codegen.CodeGenerator` (see section section 7.3). Second, information about the various exceptions in the program is generated. From each `Handler` (see section 5.3) object associated with the `FlowGraph`, a `bloat.editor.TryCatch` (see section 3.3.1) is generated. Recall that a `TryCatch` consists of the label of the first and blocks in the protected region, the label of the first block of the exception handler, and the `Type` of the exception being caught. The `TryCatchs` are added to the `MethodEditor`.

5.6.14 Looking at Control Flow Graphs

Now that we've all learned more about control flow graphs than we've ever wanted to know, we can start working with them. To make working with control flow graphs tolerable[7], BLOAT has several mechanisms for displaying control flow graphs.

[7]Tony once told me that Nate used to have dreams about control flow graphs. I call these nightmares.

Printing Expression Trees

The bloat.tree.PrintVisitor (see section 4.2) class is a TreeVisitor that generates a textual representation of the nodes in an expression tree to a java.io.PrintWriter. The following is an alphabetical summary of text generated by PrintVisitor. Note that if the expression tree node terminates a block (e.g. IfZeroStmt, GotoStmt, and RetStmt) caught by is printed followed by a list of the first blocks in the handlers for any exceptions that may be thrown in the block terminated by the statement.

AddressStoreStmt Prints La ("load address") followed by the integer index of the subroutine's return address.

ArithExpr Prints the left-hand Expr followed by the operator (+ - * /, etc.) and the right-hand expression. Note that <=> is compare, <l=> is compare less-than, and <g=> is compare greater-than.

ArrayLengthExpr Prints the array Expr followed by .length.

ArrayRefExpr Prints the array Expr followed by the index Expr surrounded by brackets.

Block Prints the block's label, its type, and the label of its header block if it is in a loop. It is also noted if the block is the source, sink, or init block of its control flow graph. If the block begins an exception handler, the type of exception that it catches and a list of its protected blocks is also given. Its contents (children) are then printed.

CallMethodExpr Prints the receiver Expr, the name of the method, and the parameter Exprs.

CallStaticExpr Prints the Type of the class on which the method is invoked, the name of the methods, and the parameter Exprs.

CastExpr Prints the Type to which to cast followed by the Expr to be cast.

CatchExpr Prints Catch followed by the type that is caught.

ConstantExpr If the constant is a String its first 50 characters are printed. Non-printable whitespace is ignored. If the constant is a Float, then the value is printed followed by an F. If the constant is a Long, then the value is printed followed by a L.

Expr By default, prints EXPR.

`ExprStmt` Prints `eval` followed by the expression in the `ExprStmt`.

`FieldExpr` Prints the object `Expr` followed by a `.` and the name of the field.

`FlowGraph` Prints the source block, followed by all of the control flow blocks in trace order, followed by the sink block.

`GotoStmt` Prints `goto` followed by the target `Label`.

`IfZeroStmt` Prints `if0` followed by the type of comparison. If the statement compares against a reference type, `null` is printed, else `0` is printed. The right-hand expression in the comparison is printed followed by the `then` and `else` targets.

`InitStmt` Prints `INIT` followed by the `LocalExpr`s that are initialized.

`InstanceOfExpr` Prints `instanceof` followed by the `Type` of the check.

`JsrStmt` Prints `jsr` followed by the entry block of the subroutine, `ret to`, then the block two which control is returned.

`LabelStmt` Prints the `Label` as `label_`*index*.

`LocalExpr` If the variable is allocated on the stack, a `T` is printed, else a `L` is printed. The `Type` of the variable followed by its index (local variable number or offset into stack). If the `LocalExpr` is defined by a `DefExpr`, its (SSA) version number is printed. Otherwise `undef` is printed. For instance, if local variable 1 contains a reference and has version 6, it will be represented by `Lr1_6`.

`MonitorStmt` Prints either `enter` or `exit` and the prints the object whose monitor is being entered or exited.

`NegExpr` Prints a `-` followed by the `Expr` that is negated.

`NewArrayExpr` Prints `new`, the `Type` of the array to be allocated, followed by the size of the array surrounded by braces.

`NewExpr` Prints `new` followed by the `Type` of object to be created.

`NewMultiArrayExpr` Prints `new`, the `Type` of the array to be allocated, followed by the dimensions surrounded by brackets.

`PhiCatchStmt` Prints the target `VarExpr`, an `:=`, and a list of the operands to the `PhiCatchStmt`.

`PhiJoinStmt` Prints the target `VarExpr`, an `:=`, and a list of the operands with the blocks from which they came.

`RCExpr` Prints `rc` followed by the `Expr` being checked.

`RetStmt` Prints `ret from` followed by the entry `Block` of the subroutine from which it is returning.

`ReturnAddressExpr` Prints `returnAddress`

`ReturnExprStmt` Prints `return` followed any expression that may be returned.

`SCStmt` Prints `aswizzle` followed by the array `Expr` and the index `Expr`.

`ShiftExpr` Prints the `Expr` to be shifted, `<<` for left shift, `>>` for right shift, or `>>>` for an unsigned right shift, followed by the `Expr` specifying the number of bits.

`SRStmt` Prints `aswrange array:` followed by the array `Expr` and the starting and ending `Expr`s.

`StackExpr` Prints `S`, by the `Type` of the stack expression, followed by the offset into the stack. If the stack variable has a known definition (`DefExpr`), its version number is printed, else `undef` is printed.

`StackManipStmt` Prints the `StackExpr`s that are targets, a `:=`, the kind of `StackManipStmt` (e.g. `dup_x1`) and then the `StackExpr`s that are the source.

`StaticFieldExpr` Prints the name of the class in which the field resides followed by the name of the field.

`Stmt` By default, prints `STMT`.

`StoreExpr` Prints the target `MemExpr`, an `:=`, and the `Expr`.

`SwitchStmt` Prints `switch`, the index `Expr`, `caught by`, and then the pairs of values and targets. The default target is printed last.

`ThrowStmt` Prints `throw` followed by the `Expr` being thrown and the first block of the exception handler that catches it.

`UCExpr` If the update check checks a pointer, then `aupdate` is printed, else `supdate` is printed. The expression being checked is printed.

`ZeroCheckExpr` If a reference type is being checked, then `notNull` is printed, else `notZero`. The `Expr` to be checked is printed.

Viewing the Control Flow Graph

The `print` method of `FlowGraph` prints a textual representation of the control flow graph to a `java.io.PrintStream` by using a `PrintVisitor`. The `printGraph` method creates a graphical representation of the control flow graph using the `dot` software available from

`http://www.research.att.com/sw/tools/graphviz/`

`dot` is used to draw graphs and can generate output in several formats including Postscript. `printGraph` uses a `PrintVisitor` to generate the nodes of a `dot` graph. Solid edges in the graph represent normal control flow edges. Dotted edges represents edges whose destination is the first block of an exception handler. `printGraph` works well with small methods (under 50 lines), but tends to get unmanageable with larger control flow graphs.

5.7 Control Flow Graph Examples

Now that we've seen how BLOAT models and constructs control flow graphs, let's look a several example of Java methods and their control flow graphs.

5.7.1 A Simple Example

To begin with let's start with a straightforward Java method that demonstrates an if-statement and some basic arithmetic operators. The source code and compiled (unoptimized) bytecode are given in Figure 5.4. It's control flow graph is given in Figure 5.5.

Let's examine each block (node) in the control flow graph. The first block is labeled `label_15`. This is the *source block*. The *sink block* is labeled `label_17`. Note that there is an edge from the source block to the sink block to represent that the method may not be executed. The block labeled `label_16` is the *init block*. This block contains an `InitStmt` that initializes local variable 0 (`Lr0`), the `this` pointer (recall that `L` stands for "local variable", `r` stands for "reference", and `i` stands for "integer"), and local variable 1 (`Li1`), the first parameter. Note that boolean values are represented by integers.

The method's code begins in the block labeled `label_0`. The first statement in the block assigns `0` to the second local variable (representing `x` in

```
public int f(boolean b) {
  int x = 0;
  if(b)
    return(x + 1);
  else
    return(x + 2);
}
```

```
public (Z)I f
  label_0
  ldc 0
  istore Local$2
  iload Local$1
  ifeq label_10
  label_6
  iload Local$2
  ldc 1
  iadd
  ireturn
  label_10
  iload Local$2
  ldc 2
  iadd
  ireturn
  label_14
```

Figure 5.4: Example 1: An if statement and basic arithmetic

the original program). The second statement is an if statement that will either branch to `label_10` or `label_6`. The blocks labeled `label_10` and `label_6` are relatively straighforward. They are both terminated by return statements and block have edges to the sink block.

5.7.2 Stack Variables

Next we consider a control flow graph that works with stack variables. Recall that stack variables (`StackExprs`) arise from `dup` instructions (see section 4.5.5). As an added bonus, we get to see objects being created and methods being invoked. Now how much would you pay?

The source code and compiled bytecode for the method in question is given in Figure 5.6. It's control flow graph is given in Figure 5.7. As we can see from the source, a `dup` instruction is used to make a copy of the `Integer` object on top of the stack. The first copy is used as an operand to the invokespecial instruction that initializes the object (i.e. incokes its constructor). The second copy is used as the reciever of the `floatValue` method (the invokevirtual instruction).

Examining the control flow graph, we see that the source block has `label_14`, the sink block has `label_16`, and the init block has label `label_15`. The block labeled `label_0` is interesting. The first statement creates a new

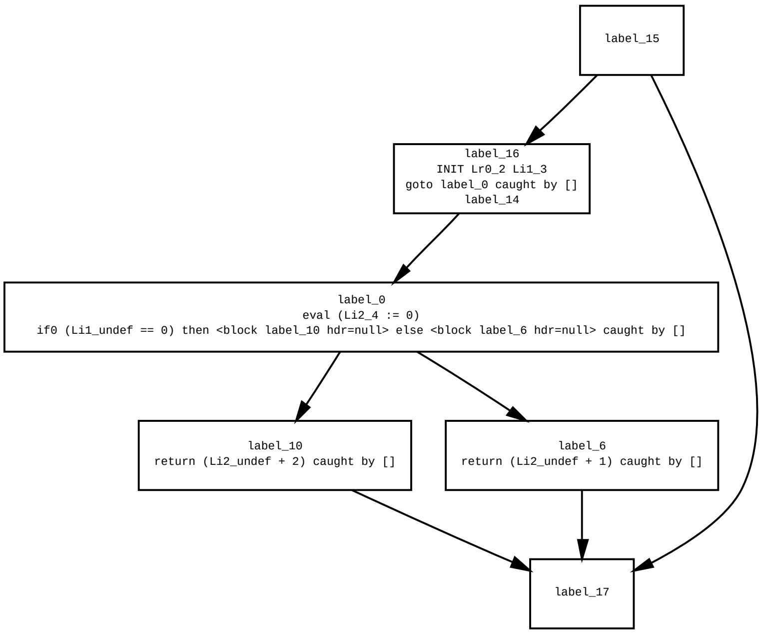


Figure 5.5: CFG for Example 1

`Integer` and assigns it to the slot on top of the stack, `Sr0`. Recall that `S` stands for "stack variable" and `r` stands for "reference". Also recall that a stack variable with index 0 is at the bottom of the stack. As the stack grows, the indices increase. The next statement represents the `dup` instruction. The top two slots on the stack (`Sr0` and `Sr1`) contain what used to be on top of the stack (`Sr0`). The next statement calls the `Integer` constructor on the object on top of the stack (`Sr1`). The statement after that invokes the `floatValue` method on the object on top of the stack (`Sr0`) and assigns its result to the second local variable (`Lf2`).

5.7.3 Exceptions

Next, we look at a method that contains an exception handler. Its source code is given in Figure 5.8. Its control flow graph is given in Figure 5.9. There are two interesting things to notice. First of all, the branch statement that terminate the block labeled `label_0` has a "`caught by`" clause associated with them. Any exceptions that occur in this block will transfer control to the block labeled `label_37`, the "catch block". This block pushes the exception object onto the stack. Edges in the control flow graph

```
public void g(int i, float x) {
  x = (new Integer(i)).floatValue();
}

public (IF)V g
  label_0
  new Ljava/lang/Integer;
  dup
  iload Local$1
  invokespecial <Method java/lang/Integer.<init> (I)V>
  invokevirtual <Method java/lang/Integer.floatValue ()F>
  fstore Local$2
  return
  label_13
```

Figure 5.6: An example using `dup` and stack variables

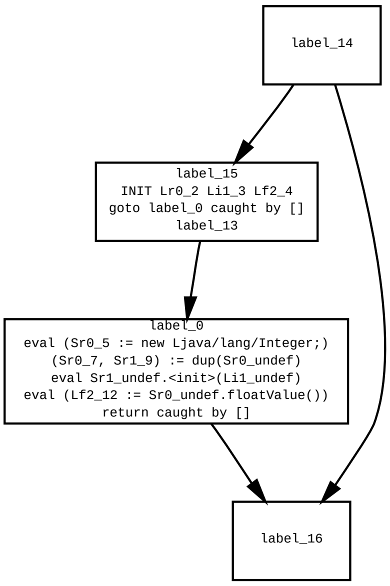


Figure 5.7: Control Flow Graph for Figure 5.6

that are taken when an exception occurs are dotted. There is also some interesting stuff that goes on in the exception handler (the "catch body", block `label_9`). Recall that when an exception occurs the exception object is pushed onto the stack. Since the exception handler makes use of the exception object, the object is popped off of the stack and placed in local variable 2 (`Lr2`). Recall that the `+` string operator in the Java language is just syntactic sugar for `StringBuffer`'s `append` method.

5.7.4 A Finally Clause

A JVM subroutine is used to implement the `finally` clause of an exception handler. An example method containing a `finally` clause is given in Figure 5.10. It's control flow graph is rather large and is shown in Figure 5.11. Before we discuss the `finally` clause, note that this method references a field. The block labeled `label_0` contains an assignment to field `i`. The object whose field is being assigned to (which in this case is the `this` pointer stored in local variable 0, `Lr0`) is wrapped inside a `ZeroCheckExpr` (the `notNull`.

Now, let's consider the exception. The method call in the block labeled `label_0` may throw an exception. The "catch block" for the exception is labeled `label_48`. The "catch body" is labeled `label_12`. (I'm not too sure what the purpose of the blocks labeled `label_49` and `label_26` are. They appear to be catching some exception that isn't thrown. This may be a bug.) Both the exceptional and the non-exceptional flows bottom out in the block labeled `label_20` that contains a jsr that jumps to a subroutine that begins with the block labeled `label_32`. The subroutine's return address is store in local variable 2, `La2`.

5.8 Summary

BLOAT performs its optimizations on a method's control flow graph. A control flow graph is a directed graph consisting of basic blocks of instructions. Special provisions must be made to accommodate exceptions and subroutines. Each basic block begins with a label that is the target of a branch and ends with a branch instruction. The instrucions in a block are modeled by an expression tree. Properties of the control flow graph such as its dominator tree, loop tree, and a block's dominance frontier can be calculated. A couple of transformations such as loop peeling, loop splitting, and removal of critical edges are performed on the control flow graph to enable certain optimizations.

```
public void h() {
  try {
    int i = Integer.parseInt("123");
  } catch(NumberFormatException ex) {
    System.out.println("NFE: " + ex);
  }
}

public ()V h
  label_0
  ldc "123"
  invokestatic <Method java/lang/Integer.parseInt (Ljava/lang/String;)I>
  istore Local$1
  label_6
  goto label_32
  label_9
  astore Local$2
  getstatic <Field java/lang/System.out Ljava/io/PrintStream;>
  new Ljava/lang/StringBuffer;
  dup
  ldc "NFE: "
  invokespecial <Method java/lang/StringBuffer.<init> (Ljava/lang/String;)V>
  aload Local$2
  invokevirtual <Method java/lang/StringBuffer.append
                              (Ljava/lang/Object;)Ljava/lang/StringBuffer;>
  invokevirtual <Method java/lang/StringBuffer.toString ()Ljava/lang/String;>
  invokevirtual <Method java/io/PrintStream.println (Ljava/lang/String;)V>
  label_32
  return
  label_33
```

Figure 5.8: An Example Containing an Exception Handler

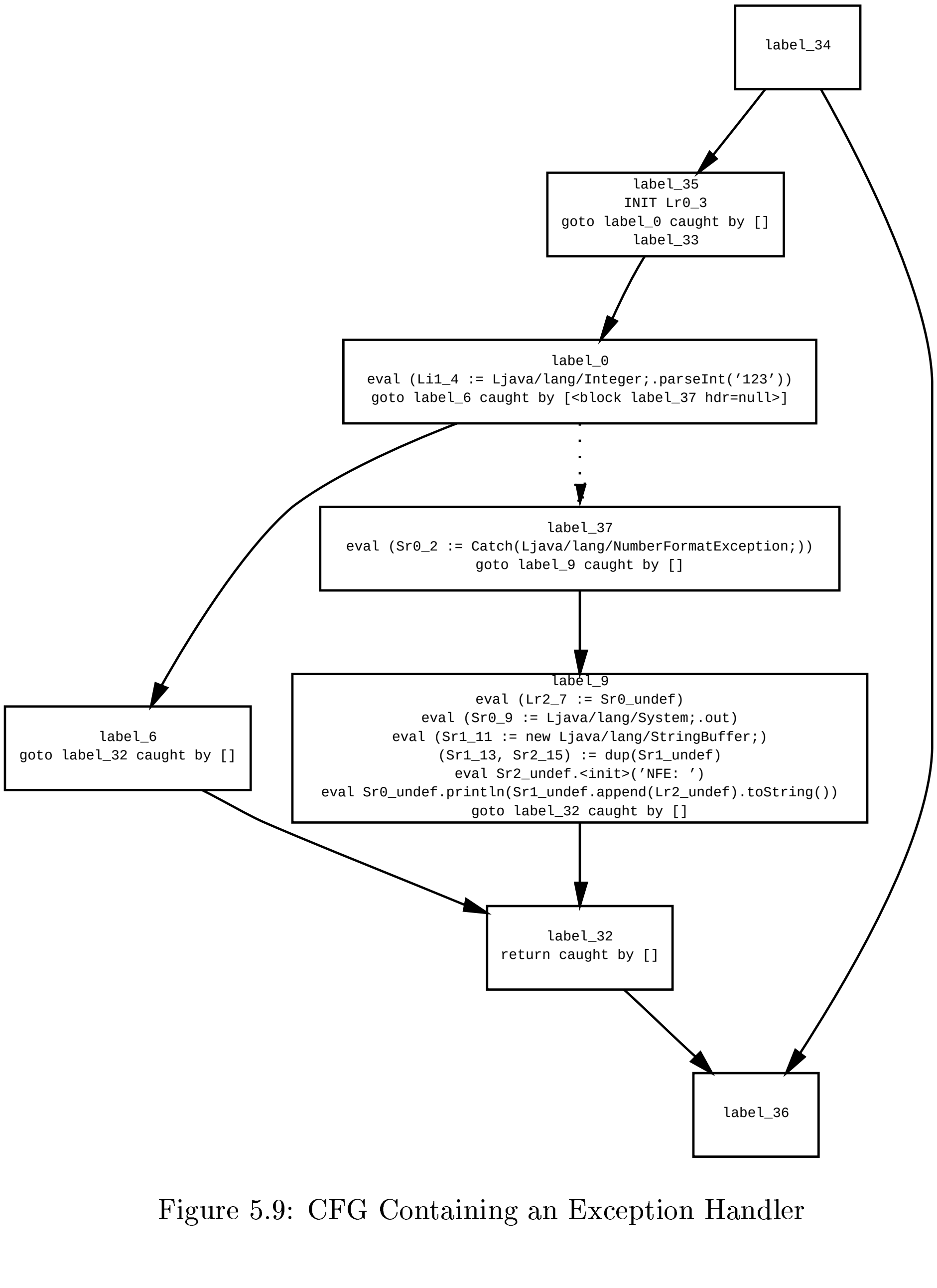


Figure 5.9: CFG Containing an Exception Handler

```
int i;
public void i() {
  try {
    i = Integer.parseInt("123");
  } catch(NumberFormatException ex) {
    System.exit(1);
  } finally {
    System.out.println("Done");
  }
}

public ()V i
  label_0
  aload Local$0
  ldc "123"
  invokestatic <Method java/lang/Integer.parseInt (Ljava/lang/String;)I>
  putfield <Field Finally.i I>
  label_9
  goto label_20
  label_12
  pop
  ldc 1
  invokestatic <Method java/lang/System.exit (I)V>
  goto label_20
  label_20
  jsr label_32
  label_23
  goto label_43
  label_26
  astore Local$1
  jsr label_32
  label_30
  aload Local$1
  athrow
  label_32
  astore Local$2
  getstatic <Field java/lang/System.out Ljava/io/PrintStream;>
  ldc "Done"
  invokevirtual <Method java/io/PrintStream.println (Ljava/lang/String;)V>
  ret Local$2
  label_43
  return
  label_44
}
```

Figure 5.10: A Java Method Containing a `finally` Clause

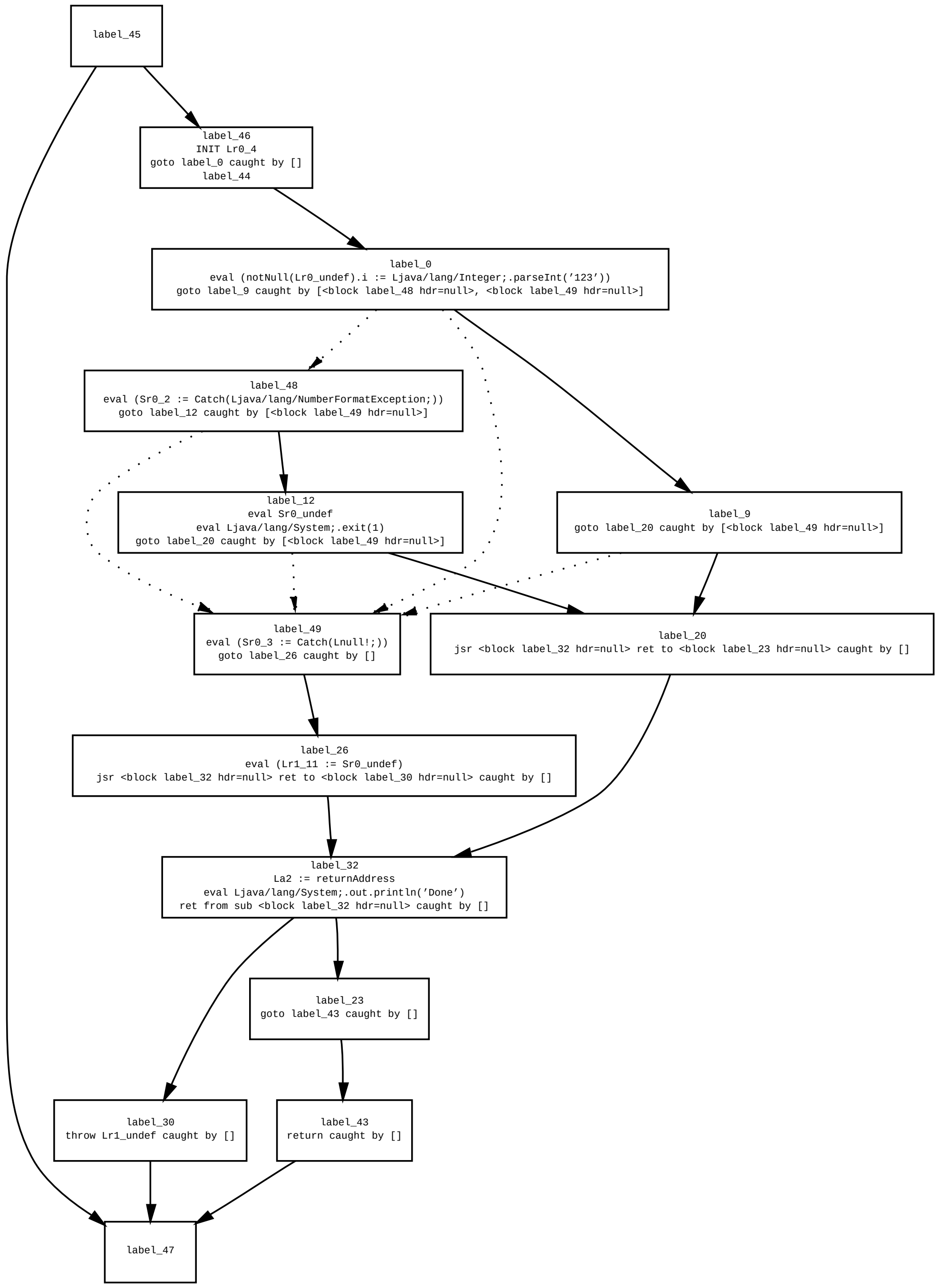


Figure 5.11: A CFG Containing a Subroutine Call

Chapter 6

Static Single Assignment Form

6.1 Background

Many optimizations need to know where variables are defined (assigned to) and where they are used. Such information is referred to as the *use-def* information. *Static Single Assignment Form* (SSA) provides a compact representation of a variable's use-def information. SSA form renames each occurrence of a variable such that each variable is only defined once (i.e. each variable has a *single* definition). When the flow of control merges (e.g. after an if-statement) SSA variables are merged using a ϕ-statement. A ϕ-statement is placed at the merge block, has operands corresponding to each incoming SSA variable, and defines another naming of an SSA variable. Figure 6.1 gives an example of SSA form.

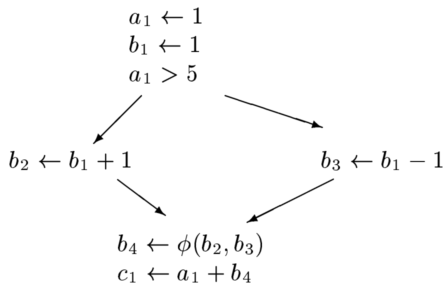
6.1.1 Placing ϕ-functions

Conceptually, ϕ-functions are placed at every merge block in the control flow graph. However, many of these ϕ-functions are unnecessary. Merge points are easily identified by using a block's iterated dominance frontier (see section 5.6.2). Recall that block z is in the dominance frontier of block x if x dominates some, but not all, of z's predecessors. The iterated dominance frontier is the union of the dominance frontiers of a set of blocks.

BLOAT uses the so-called *semi-pruned* SSA form [BCHS98]. Semi-pruned SSA form takes advantage of the fact that many variables are short-lived temporaries that exist within a single basic block. It calculates the set

$a \leftarrow 1$
$b \leftarrow 1$
if $(a > 5)$ **then**
  $b \leftarrow b + 1$
else
  $b \leftarrow b - 1$
$c \leftarrow a + b$

(a) Code



(b) Its CFG in SSA Form

Figure 6.1: An Example of SSA Form

non_locals $\leftarrow \emptyset$
for each block B **do**
  killed $\leftarrow \emptyset$
  for each instruction $v \leftarrow x$ *op* y in B **do**
    if $(x \notin$ *killed*$)$ **then**
      non_locals $\leftarrow$ *non_locals* $\cup \{x\}$
    if $(y \notin$ *killed*$)$ **then**
      non_locals $\leftarrow$ *non_locals* $\cup \{y\}$
    killed $\leftarrow$ *killed* $\cup \{v\}$

Figure 6.2: Algorithm for finding non-local variables

of variables that are live on entry to at least one basic block, the "non-local" variables, as shown in figure 6.2. Each basic block is visited once. When a variable is encountered that is not defined within the block (the "killed" set), the variable is added to the "non-local" list. So, ϕ-functions are only added for non-local variables in merge blocks. Semi-pruned SSA has the advantage of inserting a minimal number of ϕ-functions without having to perform expensive variable liveness analysis.

6.1.2 Naming Variables In SSA Form

After the ϕ-functions are inserted, the control flow graph is transformed so that each variable has a single definition and each variable use reflects this fact. The blocks in the control flow graph are visited in pre-order and the algorithm in figure 6.3 is applied to each block. As the algorithm proceeds, a global stack is maintained that keeps track of the current SSA number for each variable. Every time a variable is defined, a new SSA number is pushed onto the stack. When a use of a variable is encountered, the SSA number on the top of the stack is assigned to that variable.

6.1.3 Deconstructing SSA Form

Once all of the optimizations have been performed, the ϕ-functions must be removed from the control flow graph. ϕ-functions are replaced by a copy of each operand variable to the target variable in the predecessor block corresponding to the operand variable. To ensure that the copy is placed in the correct location, critical edges (see section 5.6.9) are removed from the control flow graph. Figure 6.4 demonstrates the need to remove critical edges.

6.1.4 Other ϕ-functions

Not surprisingly, special accommodations must be made for dealing with SSA variables in the presence of exceptions. Consider the following. A protected region defines a program variable several times. An exception handler makes use of that program variable. When converting into SSA form, which SSA variable does the exception handler use? The use could correspond to any one of SSA variables defined in the protected region. Standard SSA form dictates that edges in the control flow graph be added to the exception handler from both before and after each assignment to a local variable in the protected region.

To handle SSA variables inside protected regions, another type of ϕ-statement is used called the "ϕ-catch" statement, ϕ_c, is used to factor together all of the SSA variables in a protected region. ϕ_c-statements are inserted at the beginning of each basic block that begins an exception handler. The operands of the ϕ_c-statement are the SSA variables that occur (used or defined) within the protected region. When ϕ_c-statements are destructed a copy from the operand to the target is inserted just after the operand's definition[1]. It is possible that the operand's definition may be far away from the exception handler. As a result, the target could have an unnecessarily long live range. To alleviate this problem, copies ($a \to a$) of live variables entering the protected region are inserted. This new definition of a will cause the copy generated by the ϕ_c destruction to be inserted as close to the protected region as possible.

[1] Remember that each variable is defined once, so this is okay.

input:
  A CFG, G, after ϕ-nodes are placed
output:
  The SSA form of G

do
  for each variable v **do**
    $Stack(v) \leftarrow \emptyset$
    $Counter(v) \leftarrow 1$

  $renameBlock(entry)$
with
  procedure $renameBlock(block)$ **begin**
    for each variable v **do**
      $TopOfStack(v) \leftarrow top(Stack(v))$

    for each ϕ-node, $\langle v \rangle \leftarrow \phi(\ldots)$, in $block$ **do**
      $Version(\langle v \rangle) \leftarrow Counter(v)$
      push $Counter(v)$ onto $Stack(v)$
      $Counter(v) \leftarrow Counter(v) + 1$

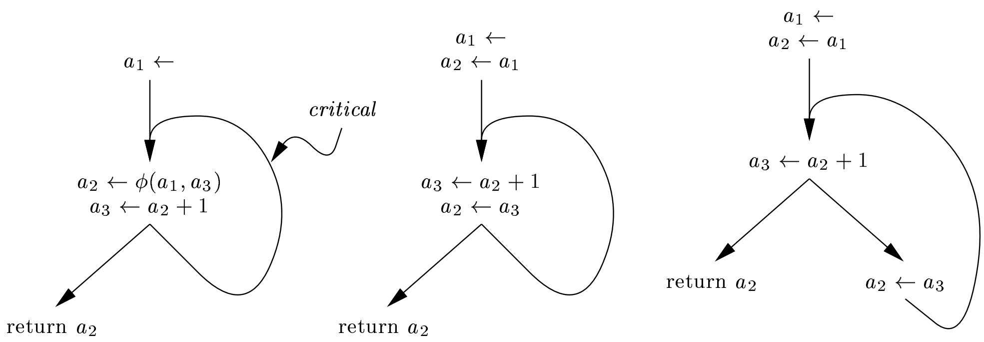
    for each instruction, $\langle v \rangle \leftarrow \langle x \rangle \otimes \langle y \rangle$, in $block$ **do**
      $Version(\langle x \rangle) \leftarrow top(Stack(x))$
      $Version(\langle y \rangle) \leftarrow top(Stack(y))$
      $Version(\langle v \rangle) \leftarrow Counter(v)$
      push $Counter(v)$ onto $Stack(v)$
      $Counter(v) \leftarrow Counter(v) + 1$

    for each $succ \in Succ(block)$ **do**
      for each ϕ-node, $\langle v \rangle \leftarrow \phi(\ldots)$, in $succ$ **do**
        $\langle v \rangle \leftarrow$ the $block$-operand of $\phi(\ldots)$
        $Version(\langle v \rangle) \leftarrow top(Stack(v))$

    for each $child \in DomChildren(block)$ **do**
      $renameBlock(child)$

    for each variable v **do**
      pop $Stack(v)$ until $top(Stack(v)) = TopOfStack(v)$

Figure 6.3: SSA Renaming (swiped from Nate's Thesis [Nys98])



(a) A program with a critical edge

(b) Incorrect ϕ replacement without splitting critical edges

(c) Correct ϕ replacement with splitting critical edges

Figure 6.4: Problems with Critical Edges (swiped from Nate's Thesis)

Subroutines also complicate the SSA representation. The Java Virtual Machine allows any local variable that is not referenced inside a subroutine to retain its type. As a result, two variables with incompatible types could be factored together in a ϕ-statement. To solve this problem, if a variable is not redefined in a subroutine, the SSA number for the variable is propagated back from the end of the subroutine to the block to which the subroutine returns, the "return site". This construct is represented by the "ϕ-return" statement, ϕ_r. Because critical edges were removed from the control flow graph, the return site has only one incoming edge, and thus the ϕ_r has only one operand. ϕ_r-statements are placed at the return site. During renaming, the operand of the ϕ_r-statement is given the SSA number that is on top of the variable's renaming stack, the uses of the SSA variables defined by the ϕ_r-statements are renamed to the ϕ_r-statement's operand, and the ϕ_r-statements are removed.

6.2 Constructing SSA Form

BLOAT converts a control flow graph into static single assignment form using the classes in the `bloat.ssa` package. The `transform` method of `bloat.ssa.SSA` begins the work of converting a `bloat.cfg.FlowGraph` into SSA form. The private `collectVars` method visits the `FlowGraph` and removes any existing `bloat.tree.PhiStmts`. It also maintains information about each variable encountered.

The class `bloat.ssa.SSAConstructionInfo` maintains information about a program (as opposed to and SSA) variable (`bloat.tree.VarExpr`) during SSA conversion. An instance of `SSAConstructionInfo` is created for every variable in the CFG. The `SSAConstructionInfo` maintains a clone of the `VarExpr` it represents (the "prototype"), a list of non-ϕ-statement occurrences of the variable (the "real" occurrences), a list of the real occurrences of a variable in a given block, the `PhiStmts` for that variable at each block (a variable can only be involved in one kind of `PhiStmt` at a given block), and a list of `bloat.cfg.Blocks` in which the variable is defined.

In addition to having methods that work with the information that it maintains, `SSAConstructionInfo` has several helper methods. `addPhi` adds a `PhiJoinStmt` for the variable represented by the `SSAConstructionInfo` to a given block to the control flow graph[2]. Similarly, the `addRetPhis` method adds a `bloat.ssa.PhiReturnStmt` to each block to which a `Subroutine` may return (see "paths" in section 5.3). The `addCatchPhi` method adds a `PhiCatchStmt` to a block if the variable represented by the `SSAConstructionInfo` is a local variable (`LocalExpr`).

6.2.1 Placing ϕ Statements

As mentioned above, the semi-pruned SSA form only places ϕ-statements for variables that occur in more than one basic block. `SSA`'s private `placePhiFunctions` method inserts `PhiStmts` into a `FlowGraph` for a given variable represented by an `SSAConstructionInfo`. Each real (non-ϕ) occurrence of the variable is examined. If the occurrence is a definition, then the variable is "killed" in the block in which the definition occurs. If a use of the variable is encountered in a block in which the variable is not killed, then the variable is "non-local" and ϕ-statements must be placed for it (see figure 6.2).

A `PhiCatchStmt` for the variable is added to every "catch block" (a block that begins an exception handler) in the control flow graph[3]. Similarly, a `PhiReturnStmt` for the variable is added at the "return blocks" of every subroutine in the program. Finally, a `PhiJoinStmt` (a regular ϕ-statement) is added to every block in the iterated dominance frontier of the blocks in which a definition of the variable occurs. Recall that once a `PhiStmt` is

[2]Well, it doesn't **actually** add the ϕ-statement to the CFG. It only marked as the `PhiStmt` at the block. It should also be noted that once a ϕ statement for a given variable is "inserted" into a block, no other ϕ statement for that variable is inserted. Thus, the order of insertion determines the precedence of the ϕ statements: `PhiReturnStmt`, `PhiCatchStmt`, then `PhiJoinStmt`.

[3]Remember that they're not **really** inserted. Most of them are useless.

"added" to a block, no other PhiStmt for the variable in question is added.

6.3 Renaming SSA Variables

The private search method of SSA performs the renaming of SSA variables. It is similar to the search algorithm given in [CFR$^+$91] and [BCHS98] except that the name stack is implicit in the way in which the method is invoked. search is called recursively for each variable (SSAConstructionInfo) in the program and the recursive call begins with the CFG's source block and an empty (null) stack.

If the top of the stack is a LocalExpr or if there is a PhiStmt for the variable in the block of interest, then the private addCatchPhiOperands is invoked with the variable, the block of interest, and the most recent definition (either the top of the stack or the target of the PhiStmt). addCatchPhiOperands determines whether or not the block is inside a protected region. If it is, then the variable becomes an operand to the PhiCatchStmt residing in the protected region's catch block.

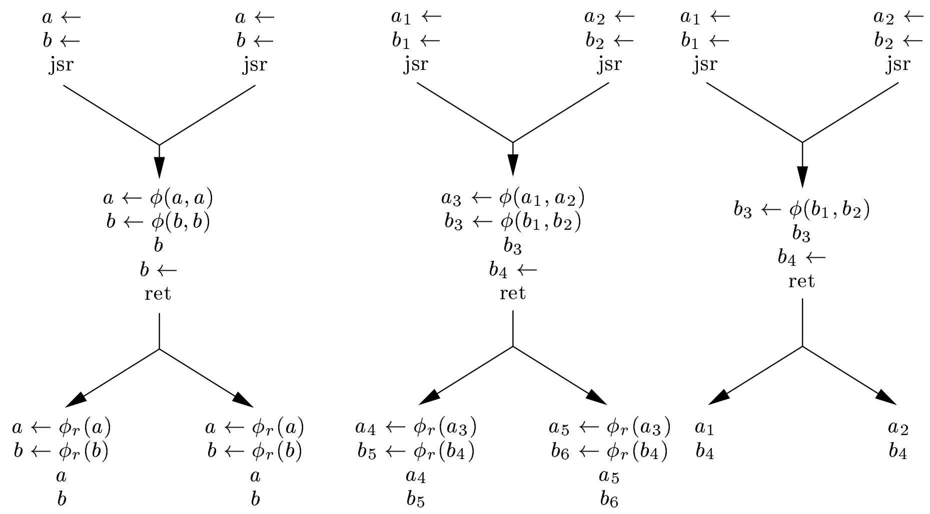
Back in search, if the block of interest is in a protected region and the variable in a stack variable (StackExpr), then the naming "stack" is cleared (i.e. set to null) because the runtime stack is popped down to 0 when an exception is caught.

Each real occurrence of the variable in the block of interest is examined. If the variable is defined in the block, then this definition becomes the top of the renaming stack. If the variable is used, we make sure that there is a valid definition of it (i.e. the top of the stack is not null) and we set its definition (using the setDef method) to be the variable on top of the stack.

Each of the block's successors in the control flow graph is visited. If the successor contains a PhiJoinStmt for the variable in question, then the operand corresponding to the block (recall that the operands to a PhiJoinStmt are represented by an SSA variable and the predecessor block from which it arrives to the merge) is assigned the SSA variable (setDef) on the top of the stack. If the successor contains a PhiReturnStmt, the SSA variable on top of the stack becomes the definition of the PhiReturnStmt's operand.

Finally, search is invoked recursively on each of the block's children in the control flow graph's dominator tree.

The deceptively-named rename method handles the naming (and subsequent removing) of PhiReturnStmts. Well, it invokes search first. Recall that the process of removing PhiReturnStmts entails replacing the uses of



(a) ϕ_r placement (b) After ϕ_r renaming (c) Final SSA form

Figure 6.5: ϕ-return (ϕ_r) Example

its target with either the SSA variable that is live at the end of the subroutine or the SSA variable that is live upon entry to the subroutine if the variable did not occur in the subroutine.

Each subroutine in the control flow graph is examined. If the entry block of the subroutine does not contain a `PhiJoinStmt` for the variable in question (the variable is live on only one path through the subroutine), then the subroutine is uninteresting and all uses of the target SSA variable will be replaced with the operand SSA variable. Additionally, if there is a `PhiJoinStmt` for the variable, but that variable is different from the operand of the `PhiReturnStmt` (the variable is redefined inside the subroutine like variable b in figure 6.5), then the subroutine is also uninteresting.

Otherwise, all uses of the target of the `PhiReturnStmt` are replaced with the SSA variable corresponding to the block in which the subroutine was called (like variable a in figure 6.5). This variable is obtained from the operands of the `PhiJoinStmt`. The `PhiReturnStmt` is removed from the control flow graph.

Finally, any remaining `PhiReturnStmts` are examined. These `PhiReturnStmts` correspond to the "uninteresting" cases mentioned above. The uses of the targets of these `PhiReturnStmt` are simply replaced with their operands.

6.3.1 Modifying the Blocks

The `insertCode` method actually adds the `PhiStmts` that were generated by the "insertion" process to the basic blocks. All of the blocks in the control flow graph are visited. If there is a `PhiStmt` for the variable of interest at a given node (recall that this information is maintained in the `SSAConstructionInfo` object), the it is added to the basic block with the `prependStmt` method of the block's expression `Tree`.

6.4 Examples of SSA Form

Now that we understand what SSA form is for, let's take a look at a couple of Java methods that demonstrate it. The first method is very straight forward. It merely contains an if-statement:

```
int f(boolean b) {
  int x;
  x = 1;
  if(b)
    x = 2;
  else
    x = 3;
  return(x);
}
```

It's control flow graph is shown in Figure 6.6. Note that block 11 contains

```
eval (Li1_7 := 0)
```

asserting the fact that the boolean variable `b` (stored in `Li1`) is false (see section 5.6.10). After it is transformed into SSA form, `PhiJoinStmts` are placed in the block 13:

```
<block label_13 hdr=label_16>
label_13
Li1_20 := Phi(label_6=Li1_1, label_11=Li1_7)
Li2_14 := Phi(label_6=Li2_6, label_11=Li2_4)
return Li2_14 caught by []
```

The definition of `Li1` in block 11 causes a ϕ-statement for `Li1` to be inserted in block 13 in addition to the ϕ-statement for `Li2` (the program

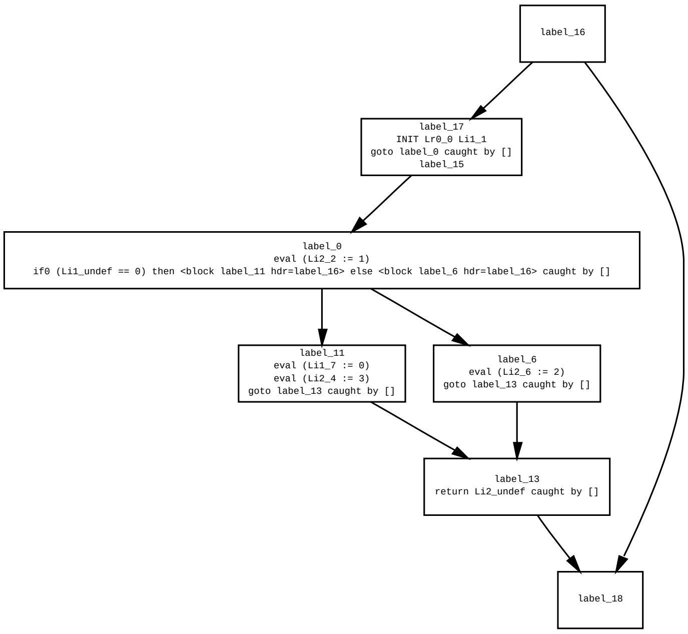


Figure 6.6: A Control Flow Graph with an if-statement

variable `x`). BLOAT also places ϕ-statements for `Li1` and `Li2` in the exit. Personally, I think this is a bug, but it's a benign one.

Converting to SSA form also assigns definitions to local variables. Notice how in block 13 in Figure 6.6 `Li2_undef` is returned. After the SSA transformation, it is established that `Li2` in the block 13 is defined by `Li2_14 := Phi(label_6=Li2_6, label_11=Li2_4)`.

The next example contains a loop:

```
int f() {
  int x;
  x = 1;
  while(x < 10)
    x = x + 1;
 return(x);
}
```

Its CFG is given in Figure 6.6. Note the effects of loop inversion (see Section 5.6.8): The loop's condition statement is duplicated in blocks 9 and 21. Notice also that the local variable `Li1` is undefined in blocks 5, 9, 15, and 21. Transformation to SSA form results in ϕ-statements being placed in blocks 5 and 15:

```
<block label_5 hdr=label_9>
label_5
Li1_15 := Phi(label_24=Li1_4, label_25=Li1_1)
eval (Li1_4 := (Li1_15 + 1))
goto label_9 caught by []

<block label_15 hdr=label_18>
label_15
Li1_12 := Phi(label_22=Li1_4, label_23=Li1_1)
return Li1_12 caught by []
```

Notice that the definitions of Li1 have been adjusted.

6.4.1 An Example ϕ_c-statement

Recall that special arrangements are made for variables that are used inside an exception handler (see Section 6.1.4). The following example contains an exception handler in which a variable is used.

```
int f(boolean b) {
  int x;
  x = 1;
  try {
    if(b)
      x = 2;
    else
      x = 3;
  } catch(Throwable ex) {
    System.out.println(x);
  }
  return(x);
}
```

The control flow graph for this program is given in Figure 6.8. An exception may be thrown in blocks 2, 6, or 11. The exception is caught in block 30. The exception handler code is contained in block 16. Since Li2 is defined (assigned to) inside the protected region (blocks 6 and 11), a ϕ_c-statement is needed in block 30.

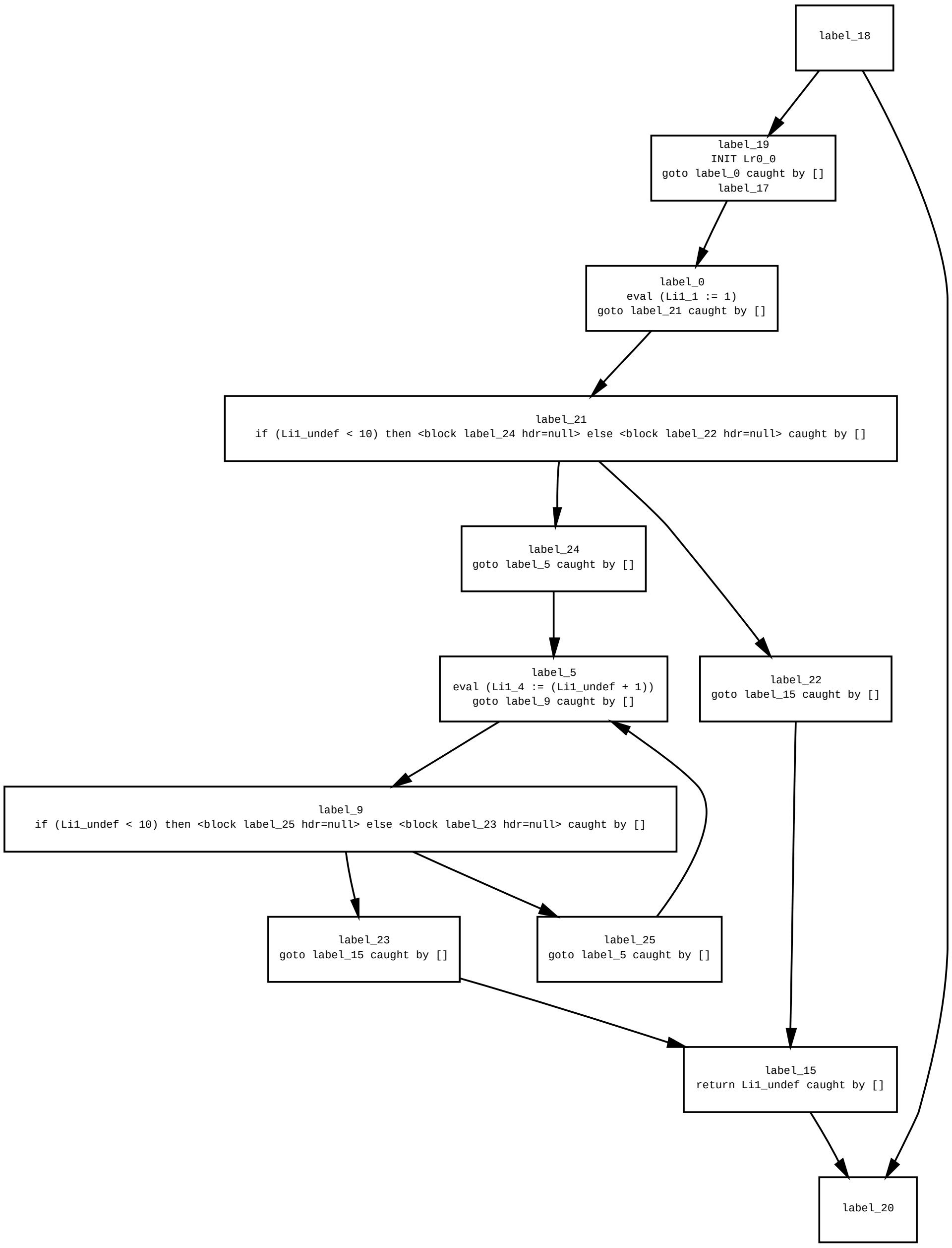


Figure 6.7: A Control Flow Graph Containing a Loop

```
<block label_30 hdr=label_27>
catches Ljava/lang/Throwable;
protects [<block label_6 hdr=label_27>, <block label_11 hdr=label_27>,
          <block label_2 hdr=label_27>]
label_30
Lr0_51 := Phi-Catch(Lr0_13)
Li1_34 := Phi-Catch(Li1_15, Li1_12)
Li2_23 := Phi-Catch(Li2_17, Li2_11, Li2_5)
eval (Sr0_0 := Catch(Ljava/lang/Throwable;))
goto label_16 caught by []
```

The ϕ_c-statement for `Lr0`, the `this` pointer, is not strictly necessary. Recall that in the exception handler, the exception object is placed on top of the stack (the `eval Sr0_undef` in block 16). Transformation to SSA form also fixes definitions of stack variables. After SSA transformation block 16 contains `eval Sr0_0`. The merge block 24 contains

```
<block label_24 hdr=label_27>
label_24
Lr0_52 := Phi(label_13=Lr0_13, label_16=Lr0_51, label_31=Lr0_13)
Sr0_44 := Phi(label_13=Sr0_undef, label_16=Sr0_0, label_31=Sr0_undef)
Li1_35 := Phi(label_13=Li1_12, label_16=Li1_34, label_31=Li1_15)
Li2_24 := Phi(label_13=Li2_5, label_16=Li2_23, label_31=Li2_11)
return Li2_24 caught by []
```

It looks like there is extra work being done. While it does increase the amount of storage necessary for the CFG and may slow down the optimizations, it does not hurt us.

6.5 Summary

Many of the optimizations that BLOAT performs rely on knowning a variable's definitions and uses. Static single assignment (SSA) form compactly represents the use-def information by renaming variable that each is defined only once. When two SSA namings of the same variable reach a merge point in the control flow graph, a ϕ-statement is inserted that represents the merging of the two variables. Traditional SSA form does not properly handle Java exceptions. Consequently, BLOAT introduces ϕ-catch and ϕ-return statements to merge variable information across exception handlers and subroutines.

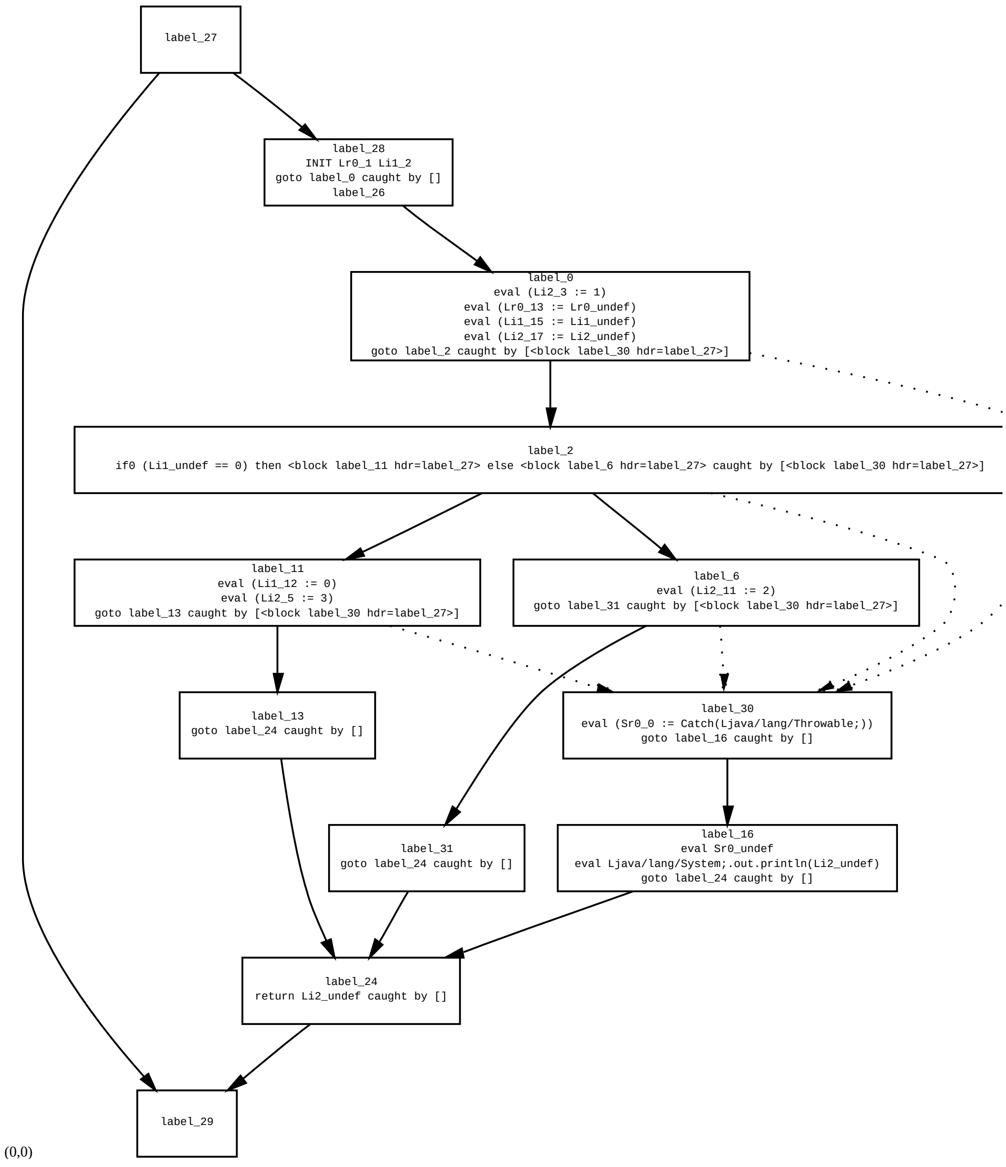


Figure 6.8: A Control Flow Graph with Exception Handling

Chapter 7

Code Generation

After performing analysis and optimization BLOAT converts the control flow graph back into a Java bytecode method. The classes in the `bloat.codegen` package perform a liveness analysis and subsequent "register allocation" on the local variables used in a method. A couple of simple optimizations on the control flow graph are performed before the code is generated.

7.1 Liveness Analysis

The code generation phase treats local variables as registers and attempts to allocate them efficiently. The `bloat.codegen.Liveness` class creates an interference graph for the local variables (`bloat.tree.LocalExpr` see section 4.5.5) used in a method. Each local variable has a node in the interference graph. An edge between two nodes indicates that the corresponding variables are simultaneously *live*. A variable is live at a given program point if it may be needed later in the program. That is, a variable v is live at a program point p if there is a path in the control flow graph from p to a use of v. The construction algorithm essentially traces backwards in the control flow graph from each use of a variable, v, to its definition. (Remember that the CFG is in SSA form, so each variable has only one definition.) Any other local variable, w, that is defined between v's definition and final use, interferes with v and causes an edge between v and w to be added to the graph.

Liveness analysis is performed on a `bloat.cfg.FlowGraph`. `Liveness`'s private `computeIntersections` method begins the work of constructing the interference graph. The interference graph is a `bloat.util.Graph` with nodes of type `IGNode`, a local class of `Liveness` that consists of a `LocalExpr` representing a variable that occurs in the method and a `List` of statements in which the variable is defined[1]. The basic blocks in the control flow graph are visited in trace order[2]. Each expression tree is visited twice in backward order using a `bloat.tree.TreeVisitor` to gather some information about variables that occur in the method. For each block, a mapping between the variables that occur in that block and in what order they occur is maintained. For each block we also keep a list of all of the variables (nodes in the interference graph) that are defined in that block. The first pass examines each `bloat.tree.PhiJoinStmt` and the second examines each `bloat.tree.LocalExpr` and `bloat.tree.PhiCatchStmt`.

Once all of the variable definitions in the program have been visited and the nodes in the interference graph have been created, analysis is performed to determine which nodes interfere with each other. First the *live out* variables are computed. A variable is live out for a given block, b, if the variable is used in a successor of b. The private method `liveOut` is used to determine at which blocks a variable is live out. This information is propagated from a variable's use to its definition.

Special care must be taken when computing the live range of a variable that is the target of a `PhiCatchStmt`. The target variable must be live throughout the entire catch block as well as after its re-definition by the `PhiCatchStmt`. However, we don't want the target to conflict (interfere) with any of the `PhiCatchStmt`'s operands. So, we make each target interfere with all of the variables that the operands interfere with. The analysis performed by `liveOut` ensures that ϕ-catch targets do not interfere with their operands. The interference edges between the ϕ-catch targets and the variables that interfere with their operands are added at the end.

`Liveness` has methods that work with the interfere graph. For instance, a variable can be removed from the interference graph, a list of variables (`LocalExprs`) that interfere with a given variable can be obtained, and it can be determine whether or not two variables interfere.

7.2 Register Allocation

Even though the Java Virtual Machine does not have registers in the traditional sense, efficient allocation of JVM local variables can be beneficial.

[1] I don't why a list is used instead of a set. Each variable should only be defined once because the CFG is in SSA form.

[2] Nate notes that the code generation of ϕs depends on going in trace order. I'm not too sure why. I guess trace order ensures that ϕ-operands will be encountered before ϕ-statements.

Variables that are accessed often (such as those that are accessed within deeply nested loops) are allocated to the first four local variables. Instructions such as iload1 may run faster than their two-byte counterparts (iload). Luckily, allocating local variables is not as complex as allocating registers. For instance, we do not have to worry about "spilling" and recomputing the interference graph.

The `bloat.codegen.RegisterAllocator` class examines the variables used in a method (`FlowGraph`) using its liveness analysis (`Liveness`). Based upon the interference graph constructed for the liveness analysis, a new interference graph containing the same interference relationships is constructed containing nodes with additional information such as whether or a node is a wide value (wide values require two local variables), the color (local variable) assigned to the node, and the weight of the node. A node's weight is a function of the loop depth (see section 5.6.7) of the blocks in which the variable it represents occurs:

$$weight(n) = \sum_{o \in occurrences(n)} (LOOP\_FACTOR)^{depth(o)}$$

If a variable is used as an operand to a `PhiJoinStmt`, the weights of each occurrence of the operand variable in the predecessors of the `PhiJoinStmt` are added to the total weight. There is a similar situation for a `PhiCatchStmt`.

We want to identify copy statements so that the variables involved in the copy may be coalesced and allocated to the same local variable. A list of copies between two nodes in the interference graph is maintained. A `PhiStmt` generates a copy between its target and each one of its parameters. A `StoreExpr` can generate a copy in one of two ways. If both the left and right sides of the assignment (store) are variables, then there is a copy between the two nodes in the interference graph corresponding to the two variables involved in the copy. However, the `StoreExpr` may also represent an iinc instruction. Such a `StoreExpr` must have an integer target and the left side of the assignment must be an `ArithExpr` consisting of a variable (`LocalExpr`) and an integer constant that can represented as a `short`. In the above situation there is a copy from the target variable to the variable in the `ArithExpr`.

Nodes that are related to each other via a copy and do not interfere with each other can be coalesced. Nodes are coalesced in an order based upon their weight. For each copy, $v \leftarrow w$, the union of the nodes that interfere with v and the nodes that interfere with w is taken. The following formula is used to determine the order in which copies are coalesced.

$$\frac{weight(w) + weight(v)}{size(union)}$$[3]

The process of coalescing w into v involves copying all of the edges (incoming and outgoing) of w into v and removing w from the interference graph. Two nodes can only be coalesced if they have the same width. All of the variable definitions of node w now apply to node v. Finally, if any other copy involves w, that copy is removed from further consideration.

The final step is to color (i.e. assign local variables to) the nodes in the interference graph. Nodes are assigned values in `InitStmts` are considered to be pre-colored. Nodes that were coalesced with pre-colored nodes are also considered to be pre-colored. The remaining uncolored nodes are sorted in decreasing order by weight. Each node in the interference graph is visited. All of its neighbors are examined to determine the available colors with which to color the node. The lowest available color is chosen. The number of colors used is kept track of. Wide variables must be assigned two consecutive colors. Once every node has been colored, its color number is assigned to the index of the local variable's `LocalExpr` and all of its uses.

7.3 Code Generation

The class `bloat.codegen.CodeGenerator` generates Java bytecodes from a control flow graph. However, before actual code generation begins several additional transformations are be made.

7.3.1 Auxiliary Methods

`CodeGenerator` has several helper methods for performing common functions. `createStore` creates a `StoreExpr` from a `Expr` source and a `VarExpr` destination. `createUndefinedStore` creates an initialization `StoreExpr` for a given `VarExpr` source. For instance, for an integer `VarExpr` it will create a `StoreExpr` that stores the constant 0 into the `VarExpr`.

7.3.2 Replacing ϕ Statements

`CodeGenerator`'s `replacePhis` method converts ϕ-statements into copy statements and removes them from the control flow graph. Two auxiliary methods, `replaceCatchPhis` and `replaceJoinPhis` do the bulk of the work.

[3] This formula seems to conflict with the one given on page 38 of Nate's thesis [Nys98].

`replaceCatchPhis` replaces `PhiCatchStmt`'s with copies from each of its operands to its target at the operand's definition point. Each block in the control flow graph that begins an exceptional handler is visited. When a `PhiCatchStmt` is encountered, the definition of each of its operands is noted. If the definition is nested inside a statement, a statement copying the operand to the `PhiCatchStmt`'s target is inserted after the defining statement. If the definition is nested inside an expression, the defining expression is replaced with a `StoreExpr` whose left-hand side is the target variable and whose right-hand side is the right-hand side of the defining expression. That is,

$$\text{operand} = \text{init}$$

becomes

$$\text{operand} = (\text{target} = \text{init})$$

`replaceJoinPhis` inserts a store of each `PhiJoinStmt`'s operand variable into the `PhiJoinStmt`'s target variable after the operand's final occurrence. This final occurrence resides in the block preceding the `PhiJoinStmt`. The control flow graph's blocks are visited in trace order by a `bloat.tree.TreeVisitor`. When a `PhiJoinStmt` is encountered, each of its operands is visited. If by some chance the target and operand were allocated to the same local variable, no copy is inserted. Recall that each block ends in a jump statement. This jump statement may contain an expression that uses local variables. The copy statement that is inserted for the operand of the `PhiJoinStmt` must not redefine any of the local variables used in the expression in the jump statement. So, the expression in the jump is copied to a stack variable (i.e. pushed on the stack), the copy of the ϕ-operand to the ϕ-target is inserted, and the stack variable is used in place of the expression in the jump statement.

Lastly, all of the `PhiStmt`'s are removed from the control flow graph.

7.3.3 Simplifying Control Flow

The various optimizations that BLOAT performs may render some basic block useless. For instance, a block may consist solely of a jump to another block. These blocks are useless and can be removed from the control flow graph. The method `simplifyControlFlow` removes such blocks.

`simplifyControlFlow` calls `removeEmptyBlocks` to remove blocks that are empty. A block is considered empty if it only contains `GotoStmts`, JsrStmts, RetStmts, and LabelStmts. Basically, an empty block contains labels and a jump. If an empty block ends with a GotoStmt, all it does is jump to a target block. The jump statements that terminate the predecessors of the empty block are modified to jump to the empty block's successor instead of the empty block.

There are two interesting cases when an empty block ends in a RetStmt. Obviously, the empty block is in a subroutine. If the predecessor to the empty block ends with a JsrStmt, then the entire subroutine is empty. The JsrStmt is replaced with a jump (GotoStmt) to the block following the JsrStmt (i.e. the block to where the subroutine would have returned). The catch targets of the GotoStmt are updated accordingly. All paths (see section 5.5.1) involving the JsrStmt are removed from the Subroutine.

In the case that the block preceding the RetStmt is a GotoStmt (that jumps to the empty block), the subroutine is still valid. The GotoStmt is replaced with a clone of the RetStmt. The catch targets of the RetStmt clone and the exit block of the Subroutine must be updated appropriately.

If the last statement in the empty block is a JsrStmt, each of the empty block's predecessors is visited. If the predecessor ends in a GotoStmt, the GotoStmt is replaced with a clone of the JstStmt. The control flow graph and Subroutine called by the JsrStmt are updated appropriately.

After removeEmptyBlocks has been called, simplifyControlFlow examines each Subroutine in the control flow graph. If there is only one path through the subroutine (i.e. the subroutine is only called once), then its corresponding jsr and ret instructions can be replaced with cheaper goto instructions. So, the subroutine's entry and exit blocks are examined. The JumpStmt (a RetStmt) that terminates the exit block is replaced with a GotoStmt that jumps to the block following the JsrStmt. The JsrStmt is replaced with a GotoStmt to the entry block of the subroutine. At this point the subroutine is no longer really a subroutine. So, it is removed from the list of subroutines that the FlowGraph maintains. Additionally, all of the AddressStoreStmts that store the return address of the subroutine are removed from the control flow graph.

7.3.4 Allocating Subroutine Return Addresses

During register allocation it may be possible that the local variable that stored a subroutine's return address gets allocated to another variable. Instead of worrying about how return address variables conflict with other variables, return address variables are given unused ("fresh") local variables. This is accomplished by the allocReturnAddresses method.

7.3.5 Generating Code

`CodeGenerator` implements the `bloat.tree.TreeVisitor` interface. Code for each kind of node in an expression tree is generated in the visitor methods implemented by `CodeGenerator`. `bloat.editor.Instructions` are added the `bloat.editor.MethodEditor` that represents the method for which code is being generated. It is assumed that the `MethodEditor`'s `clearCode` method has been called before code is generated. Code generation essentially involves breaking down nodes in the method's control flow graph into JVM instructions. The visitor visits most nodes in post-order and invokes the `addInstruction` method of the `MethodEditor`.

Postponing Instructions

Code generation performs one optimization. The insertion of instructions dealing with checks of persistent objects (`rc` and `uc`) is postponed until the last possible moment. Let's consider a message send `x.m(a.f)`. Both `x` and `a` must be checked for residency. We could generate code like this:

```
aload x
rc 0
aload a
rc 0
getfield <A.f>
invoke <X.m>
```

However, we would like to generate:

```
aload x
aload a
rc 0
getfield <A.f>
rc 1
invoke <X.m>
```

So, when we encounter an `RCExpr` we look at its parent in the expression tree. Depending on its parent's type, the generated `rc` instruction may be postponed. This is accomplished by maintaining a mapping between the parent node and the instruction that is postponed. The visitor method for the parent will call the `genPostponed` method to add the postponed instruction to the `MethodEditor` (remember that code is generated for a node's children first).

Visiting Nodes

The following is a list of various types of nodes in the expression tree and a description of the code that is generated when they are encountered. Along the way we maintain the current height of the stack. Unless otherwise specified, it is assumed that the traversal proceeds in post-order. That is, a node's children are visited and any postponed instructions are generated before the code for the node itself is generated.

`FlowGraph` Before any code is generated a couple of checks are performed on the control flow graph. First, each `Label` in a block is visited and it is ensured that only one label is designated as starting the block. Then the code for each block is generated. Lastly, each protected region in the method is examined and `bloat.editor.TryCatch` objects are created to represent the protected regions.

`ExprStmt` If the `ExprStmt`'s expression is not a `StoreExpr` and has a non-void type, then a pop instruction is generated. If the type of the expression is wide, then a pop2 is generated, else a pop is generated.

`GotoStmt` A goto instruction whose target is the label of the `GotoStmt`'s target is generated.

`IfCmpStmt` The true and false blocks are considered. If the next block in the trace is the false block, then the `genIfCmpStmt` helper method is called. If the next block in the trace is the true block, the `IfCmpStmt` is negated and `genIfCmpStmt` is called. If neither block is next, `genIfCmpStmt` is called and a goto the false block is generated.

`genIfCmpStmt` examines the type of the comparison being performed by the `IfCmpStmt` and generates the appropriate opcode: if_acmpeq, if_acmpne, if_icmpeq, if_icmpne, if_icmpgt, if_icmpge, if_icmplt, or if_icmple, with the label of the true block as an operand.

`IfZeroStmt` The process for generating code for an `IfZeroStmt` is similar to that of a `IfCmpStmt`, except that there is an equivalent `genIfZeroStmt` method and the generated opcodes are: ifnull, ifnonnull, ifeq, ifne, ifgt, ifge, iflt, or ifle.

`LabelStmt` The `addLabel` method of the `MethodEditor` is invoked with the `LabelStmt`'s label.

`MonitorStmt` Depending on the kind of the `MonitorStmt` either a monitorenter or monitorexit instruction is generated.

RCExpr If the RCExpr is nested inside an ArrayRefExpr, CallMethodExpr, or a FieldExpr then the generated rc instruction can be postponed. The index of the rc depends on the properties of the expression in which it is nested.

If the RCExpr is not nested inside one of the interesting expressions, its index is calculated as follows. If the operand of the RCExpr is a StackExpr, then the index is the current height of the stack minus one less than the index of the StackExpr. Otherwise, the index is zero because whatever the operand expression is, its value will be at the top of the stack.

UCExpr If the UCExpr is nested inside a FieldExpr, the generated aupdate or supdate instruction is postponed. Otherwise, the instruction is generated in a manner similar to RCExpr above.

RetStmt Generates a ret instruction with the local variable containing the return address of the subroutine as its operand.

ReturnExprStmt Depending on the type of the expression being returns it generates an areturn, ireturn, lreturn, freturn, or dreturn instruction.

ReturnStmt Generates a return instruction.

StoreExpr If the StoreExpr's parent in the expression tree is not an ExprStmt, then the StoreExpr has a value. If both the left-hand and right-hand sides of the StoreExpr are the same variable (LocalExpr with the same index) and the StoreExpr does not have a value, then the StoreExpr is uninteresting and no code is generated.

If the left-hand side is a variable and the right-hand side is of integral type, then there is the potential that the StoreExpr represents an increment. An iinc instruction is generated when it can be discerned that the right-hand side consists of an ArithExpr with one integer constant (ConstantExpr) operand that can be represented as a short and the same local variable (LocalExpr) as the left-hand side.

If the StoreExpr has a value, we can use the dup instructions depending on the type of the left-hand side of the StoreExpr. The generated instructions are summarized in Table 7.1.

AddressStoreStmt Generates an astore instruction with an argument of the return address of the Subroutine associated with the AddressStoreStmt.

`JsrStmt` Generates a jsr instruction. If the block that is executed after the `JsrStmt` does not come immediately after it, a goto instruction is generated that jumps to the appropriate block.

`SwitchStmt` Generates a switch instruction and constructs a `bloat.editor.Switch` object to represent the mapping between target values and their labels.

`StackManipStmt` Because all of the children of a `StackManipStmt` are stack variables, they are not visited. Depending on the kind of `StackManipStmt` one of the following instructions is generated: swap, dup, dup_x1, dup_x2, dup2, dup2_x1, or dup2_x2.

`ThrowStmt` Generates a athrow instruction.

`SCStmt` Generates a aswizzle instruction.

`SRStmt` Generates a aswrange instruction.

`ArithExpr` Generates one of the arithmetic instructions: iadd, ladd, fadd, dadd, iand, land, idiv, ldiv, fdiv, ddiv, imul, lmul, fmul, dmul, ior, lor, irem, lrem, frem, drem, isub, lsub, fsub, dsub, ixor, lxor, lcmp, fcmpl, dcmpl, fcmpg, or dcmpg.

`ArrayLengthExpr` Generates an arraylength instruction.

`ArrayRefExpr` If the `ArrayRefExpr` defines a variable one of the store instructions is generated: aastore, bastore, castore, sastore, iastore, lastore, fastore, or dastore. If the `ArrayRefExpr` does not define a variable, the one of the load instructions is generated: aaload, baload, caload, saload, iaload, laload, faload, or daload.

`CallMethodExpr` Generates invokevirtual, invokespecial, or invokeinterface depending on the kind of the `CallMethodExpr`. The argument to the instruction is the `CallMethodExpr`'s method, an instance of `bloat.editor.MemberRef`.

`CastExpr` If the `CastExpr` has a reference type, then a checkcast instruction is generated. Otherwise, one of the $x2y$ instructions is generated.

`ConstantExpr` Generates an ldc instruction.

`FieldExpr` If the `FieldExpr` is a definition, then a putfield instruction is generated; else, a getfield instruction is generated. The instruction's operand is the name of the field reference by the `FieldExpr`.

Left-hand Side	Stack Before	Stack After	Instruction
`ArrayRefExpr`	array index rhs	rhs array index rhs	dup_x2 or dup2_x2
`FieldExpr`	object rhs	rhs object rhs	dup_x1 or dup2_x1
Other	rhs	rhs rhs	dup or dup2

Table 7.1: Generation of dup instructions from `StoreExpr`s

`InstanceOfExpr` Generates an instanceof instruction.

`LocalExpr` If the `LocalExpr` is a definition, then one of the store instructions (astore, istore, fstore, lstore, or dstore) is generated. Otherwise, one of the load instructions (aload, iload, lload, fload, or dload) is generated.

`NegExpr` Depending on the type of the `NegExpr` generates ineg, fneg, lneg, or dneg.

`NewArrayExpr` Generates a newarray instruction with an operand of the element type of the `NewArrayExpr`.

`NewExpr` Generates a new instruction with an operand of the object type (`bloat.editor.Type`) of the `NewExpr`.

`NewMultiArrayExpr` Generates a multianewarray instruction and a `bloat.editor.MultiArrayOperand` object to represent the operand.

`ReturnAddressExpr` Does nothing. Hrm.

`ShiftExpr` Generates a ishl, ishr, iushr, lshl, lshr, or lushr depending on the properties of the `ShiftExpr`.

`DefExpr` Nothing interesting.

`CatchExpr` Nothing interesting.

`StackExpr` Nothing interesting.

`StaticFieldExpr` If the `StaticFieldExpr` is a definition, a putstatic instruction is generated, else a getstatic instruction is generated.

`ZeroCheckExpr` Nothing interesting.

7.4 Summary

After optimizations have been performed on a method's control flow graph, new bytecodes for the method are generated. To make efficient use of Java Virtual Machine local variables, local variables in the method are treated as "registers" and are allocated to JVM local variables. SSA ϕ-statements in the control flow graph are converted into copies and empty blocks are removed. Finally, the expression trees in the control flow graph are visited and bytecodes are generated.

Part II

Optimizing Java Classes

Chapter 8

Program Transformations

The `bloat.trans` package contains a number of classes that perform optimizations on control flow graphs. These optimizations include dead code elimination, value folding, type inferencing, SSA-based partial redundancy elimination, expression propagation, persistent check elimination, and peephole optimizations. This chapter focuses on array initialization compaction, dead code elimination, value folding, expression propagation, persistent check elimination, and peephole optimizations. Chapter 9 covers typed-based alias analysis and chapter 10 describes SSA-based partial redundancy elimination.

8.1 Array Initialization Compaction

Some Java compilers generate straight-line code for initializing arrays. Basically there is a iconst, bipush, iastore, and a dup instruction generated for each element of the array (in this case, an array of integers). As a result, classes that have large, initialized arrays can have unnecessarily large classfiles. Array initialization compaction translates the initialization code into a loop that loads elements of an array from a string in the class's constant pool. Note that we only compact arrays of `byte`s, `short`s, `int`s, `char`s, and `boolean`s.

Array initializer compaction is performed by `bloat.trans.CompactArrayInitializer`. The work is divided between the `transform` method and the private `fillArray` method. `transform` takes a `bloat.editor.MethodEditor` (see section 3.3.2), extracts its code and scans it for an array initialization. If an array initialization is found, it calls `fillArray` to create a string in the constant pool, fill it with the data in the array, remove the old initialization code, and insert new code.

The following guidelines are used to determine what code constitutes an array initialization. The entire process can be thought of as a finite state machine.

1. When a load (constant) instruction is encountered, its operand is usually the size of the array.

2. When a "new array" instruction is encountered, its operand is used to determine the type of the array (see `bloat.editor.Type` section 3.2.1). Note that we only compact "integral" types: integers, shorts, characters, and booleans. At this point, the buffer to hold the data in the array is created.

3. A `dup` instruction is expected to occur next. If it does not, start over again.

4. The next load will load an index into the array. The data at the index will be the operand of the following load.

5. An array store instruction causes the value to stored into the buffer.

CompactArrayInitializerTHRESHOLD

This process repeats until an `astore`, `aastore`, `putstatic`, or `putfield` opcode is encountered, signifying the end of an array initialization. If a minimum number of elements have been read (`THRESHOLD`), the `fillArray` method is called to create the new UTF8 string containing the array, remove the old code, and insert new code.

`fillArray` begins by creating a character array to hold the data in `char` form[1]. A UTF8 string may be no longer than 64K bytes. Just to be safe, the compactor creates UTF8 strings of 32K bytes. Multiple strings may be necessary.

The old array initialization code is removed from the method. However, the `newarray` instruction remains so that the item on the top of the stack is the array being initialized.

Inserting new code to initialize the array differs depending on the data type of the array being initialized. If it is a `char` array, then `String.getChars()` method is invoked to copy the contents of the UTF8 string (represented as a `String` object) into an array of `char`.

[1]Remember that character and short data is 16-bits wide, byte and boolean data are 8-bits wide, and integer data is 32-bits wide and is stored in big-endian format.

Loading non-character data from the UTF8 string requires a little more work. First of all, the UTF8 string is copied into a local array of `char`. Then a loop is inserted to load each element of the array being initialized from the character array. Note that an `int` must be assembled from two `chars` and each `char` holds two `byte/boolean`.

8.2 Dead Code Elimination

Code that does not contribute to the final output of a program is considered to be "dead" code. BLOAT performs SSA-based dead code elimination as described in [CFR$^+$91]. There are three conditions under which a statement is considered to be live[2]:

1. The statement affects program output. For BLOAT's purposes this requirement extends beyond I/O or calling a routine that has side effects. Statements that may throw exceptions and synchronized statements are also considered to be live.

2. The statement is an assignment statement and its target is used in a live statement.

3. The statement is a conditional branch and one or more live statements are control dependent on the conditional branch's execution.

BLOAT implements dead code elimination with the `bloat.trans.DeadCodeElimination` class. The `transform` method does the work of marking nodes in the expression trees of a `FlowGraph`'s basic blocks as being dead or live. It uses a worklist mechanism similar to the algorithm presented in [CFR$^+$91]

Initially, all nodes in the expression tree are marked as `DEAD`. A `Node`'s `key` is used to keep track of its liveness. A number of kinds of statements and expressions are marked as being live (pre-live). Several of them may throw exceptions: `MonitorStmt`, `ZeroCheckExpr`, `CastExpr`, `ArithExpr` using division (`DivideByZeroException`), `ArrayRefExpr`, and `FieldExpr`. Several of them may change memory: `NewMultiArrayExpr`, `NewArrayExpr`, `SRStmt`, `SCStmt`, `RCExpr`, `UCExpr`, `NewExpr`, and a `StoreExpr` whose target is not a local variable (`LocalExpr`). All `InitStmts` are considered to be live so that formal parameters are correctly initialized during register coloring

[2]Note that this concept of liveness is slightly different than the liveness used in register allocation (see section section 7.1).

(see section 7.2). Statements that branch are pre-live: `JsrStmt`, `RetStmt`, `CallStaticExpr`, `CallMethodExpr`, `ThrowStmt`, `SwitchStmt`, `IfStmt`, `GotoStmt`, `ReturnStmt`, and `ReturnExprStmt`. Statements that change the stack are also pre-live: `StackExpr` that are not contained in `PhiStmts`, `CatchExpr` because the stack is cleared when an exception occurs, and `StackManipStmt`.

Each expression tree `Node` is marked as being live with the private `markLive` method. If a `StoreExpr` is being marked as live, its target and right-hand side expression are also marked as being live. Its target and RHS are also added to the worklist. Note that only `VarExprs` are added to the worklist. If the node being marked live resides within an `ExprStmt` (that is, its parent is an `ExprStmt`), then the `ExprStmt` is live. The node is then visited by a `TreeVisitor` that marks the node's children as being live. If the child is a `StoreExpr` whose target is a local variable, the target is not immediately made live. If the target is used again in a live expression, it will get marked as live then.

The `VarExprs` in the worklist are then examined. Each of the `DefExpr` that define the `VarExprs` is marked as being live. Once every node that is live has been marked as such, the removal can begin. First, dead stores are removed. If the left-hand side of a `StoreExpr` is dead and its right-hand side is live, then all occurrences of the `StoreExpr` are simply replaced with the right-hand side.

In some cases a live expression may be nested inside an expression that is dead. All occurrences of the live expression are replaced by a new stack variable (`StackExpr`). An `ExprStmt` evaluating the live expression is placed before the statement containing the live expression[3].

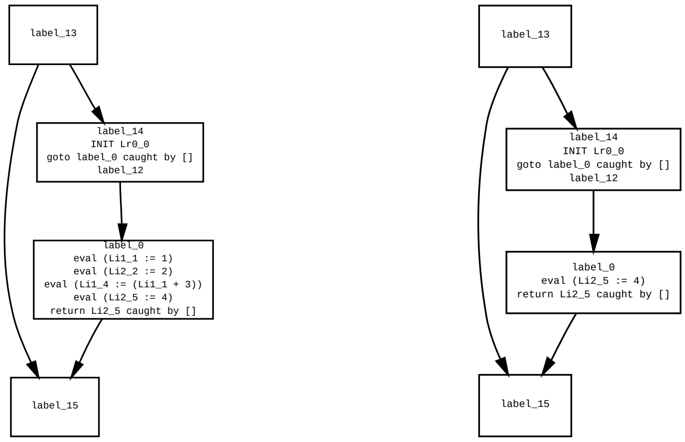
Finally, statements that are dead are removed from the tree. `LabelStmts` and `JumpStmts` are never removed.

8.2.1 An Example of Dead Code Elimination

The below code gives a very simple example of dead code elimination.

```
int f() {
  int x = 1, y = 2;
  x = x + 3;
  y = 4;
  return(y);
```

[3] Nate refers to this as "Pull out live expressions from their dead parents." Before he went into computer science, Nate used to write greeting cards. Eeep.



Before DCE After DCE

Figure 8.1: An Example of Dead Code Elimination

```
}
```

This method's control flow graphs both before and after dead code elimination are given in Figure 8.1. Notice that the code for calculating the dead program variable, `x` (store in `Li1`), is removed from block 0.

8.3 Value Numbering

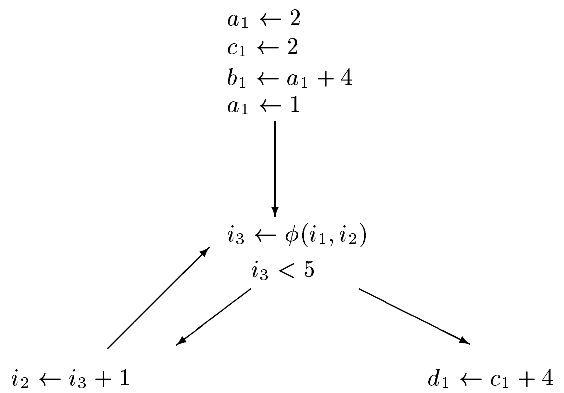
Value numbering associates a number with each expression in a program such that if two expressions have the same number, they have the same value. Therefore, if two expressions have the same value number, one of them may be eliminated. Traditional value numbering techniques used a hashing method to associate a number with each expression. The value numbering method that BLOAT uses (see [CS95] and [Sim96]) associates expressions based on the concept of *congruence* of variables. Two variables are congruent if their definitions have identical operators and congruent operands (equal constant values are always congruent). For instance, $a \to b + 3$ is congruent to $c \to 3 + d$ if b is congruent to d.

8.3.1 The SSA Graph

A control flow graph's *SSA graph* (also called the *value graph* ([Muc97])) represents how expressions in the CFG are related to each other. The SSA graph is a directed graph whose nodes are expressions (anything that has

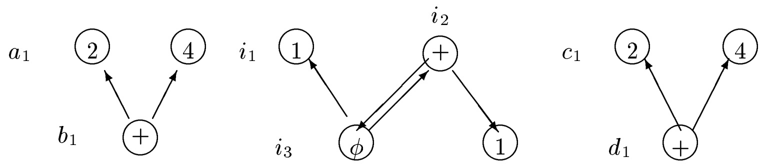
$a \leftarrow 2$
$c \leftarrow 2$
$b \leftarrow a + 4$
$i \leftarrow 1$
while $(i < 5)$ **do**
  $i \leftarrow i + 1$
$d \leftarrow c + 4$

(a) Code



(b) CFG in SSA Form

Figure 8.2: Code and CFG for Figure 8.3



(c) Its SSA Graph

Figure 8.3: The SSA Graph

a value). Edges in the graph represent which expressions are operands of another. Assignments are represented by labeling a node with the variable into which an expression is assigned. Let us consider figure 8.3. The assignment $a_1 \leftarrow 2$ creates a 2 node with the label a_1. The expression $a_1 + 4$ creates a + node with outgoing edges to the a_1 node and the 4 node. Notice that ϕ-statements are also represented in the SSA graph. In figure 8.3 the dependencies between the SSA variables for i result in a loop in the SSA graph. Notice also that variables b_1 and d_1 are congruent. Two nodes in the SSA graph are considered to be congruent if either they are the same node, they represent constants and their contents are equal, or they have the same operators and their operands are congruent.

BLOAT's implementation of value numbering works on the strongly connected components (SCCs) of the SSA graph [CS95]. The SSA graph represents how expressions in the control flow graph are "nested". If an SSA graph does not contain any cycles (SCCs), then a reverse post-order (visit a node's children right-to-left, before visiting the node) traversal of the SSA

while (there exists and unvisited node, n) **do**
  $DFS(n)$
procedure $DFS(node)$ **begin**
  $node.DFSnum \leftarrow nextDFSum++$
  $node.visited \leftarrow$ `TRUE`
  $node.low \leftarrow node.DFSnum$
  $PUSH(node)$
  for each operand o of $node$ **do**
    if $(\textbf{not}(o).visited)$ **then**
      $DFS(o)$
      $node.low \leftarrow MIN(node.low, o.low)$
    if $(o.DFSnum < node.DFSnum$ **and** $o \in stack)$ **then**
      $node.low \leftarrow MIN(o.DFSnum, node.low)$
  if $(node.low == node.DFSnum)$ **then**
    $SCC \leftarrow \emptyset$
    do
      $x \leftarrow POP()$
      $SCC \leftarrow SCC \cup \{x\}$
    while $(x \neq node)$
    $ProcessSCC(SCC)$

Figure 8.4: Tarjan's Algorithm for Finding SCCs

graph would be sufficient to value number. Cycles complicate things. So, each SCC is identified and treated as a single node. Thus, when we visit a node in a reverse post-order traversal, we know that all of the node's children (operands) have already been visited.

Tarjan's algorithm (figure 8.4) is used to find the SCCs in the flow graph. Once an SCC is found, its components are value numbered using the algorithm in figure 8.5. The SCC-based value numbering algorithm maintains two hashtables of value numbers. The *valid table* contains value numbers that are known to be correct. If an SCC contains only one component, then the valid table is used to compute the component's value number. The SCCs with multiple components are visited in reverse postorder **with respect to the CFG** using the *optimistic table*. Iterations over the components in the SCC effect their value numbers and refine the optimistic table. Once the optimistic table is refined, the components of the SCC are value numbered using the valid table. Along the way the components of the SCC (expressions) are simplified using algebraic identities and constant folding. ϕ-statements may be simplified if all of their operands are equal.

procedure $ProcessSCC(SCC)$ **begin**
  if (SCC has a single member n) **then**
    $Valnum(n, valid)$
  else
    do
      $changed \leftarrow$ `FALSE`
      for each $n \in SCC$ in reverse postorder **do**
        $Valnum(n, optimistic)$
    while ($changed$)
    for each $n \in SCC$ in reverse postorder **do**
      $Valnum(n, valid)$

Figure 8.5: SCC-Based Value Numbering

8.3.2 Implementation

SSA Graph

BLOAT's implementation of value numbering follows the algorithms described, but it's a little screwy. The SSA graph is implemented by `bloat.ssa.SSAGraph`. An `SSAGraph` visits the expression trees in a `FlowGraph` and determines which nodes are equivalent to each other. A `CheckExpr` and the expression it checks are equivalent. A `PhiStmt` and its target are equivalent. If a `VarExpr` does not define a variable, then the `VarExpr` and its definition are equivalent. `StackManipStmts` are visited such that the corresponding stack expressions before and after the manipulation are equivalent. Equivalent nodes are stored together in `Sets`.

The `children` method of `SSAGraph` constructs a list of a node's children in the SSA graph (the `Node`'s operands). If the `Node` is a `StoreExpr`, then its RHS is added to the list of children. If the LHS is a `VarExpr`, then it is equivalent to the `Node` and is therefore not a child. If the `Node` is a `PhiStmt`, the its operands are its children. Otherwise the `Node`'s "children" (`visitChildren` method) are used.

So far, things have been okay. The implementation takes a left turn towards Bizarro World in the `visitComponents` method. Okay, the algorithm in figure 8.4 is implemented in `visitComponents`. The SCC-based value numbering algorithm (figure 8.5) is implemented in `bloat.trans.ValueNumbering` which creates an anonymous implementation of the `bloat.ssa.ComponentVisitor` interface which contains one method, `visitComponent`, that works on a list of components[4]. Whatever!

`visitComponents` in `SSAGraph` computes the strongly connected components of the SSA graph and invokes the `ComponentVisitor` on each one. First each `Node` in the CFG is assigned a global (that is, with respect to all blocks) depth first search number. Then, each `Block` is visited in reverse post-order. If a `Node` has not already been visited, then it is assigned the next depth-first number and is pushed onto a stack. Each `Node` that it is equivalent to is also assigned that same depth-first number. Each child of the node in the SSA graph is then recursively visited.

Once the children have been visited, if the "low" depth-first number is equal to the current node's depth-first number, then we have a strongly connected component. So, we've visited all of the children and now we're back to where we started. Nodes are popped off of the stack until the current node is reached. The popped nodes constitute an SCC. The components in the SCC are sorted to ensure that they are still in reverse post-order. Finally, the `ComponentVisitor` is invoked on the strongly connected component.

If the node in question has already been visited (that is, it already has a depth-first number), then the edge between it and its parent node must form a loop. It is left as an exercise to the reader to try to figure out the correlation between Tarjan's algorithm and the current implementation. I ain't doing it.

Auxiliary Classes

Before we dive head first into value numbering, there are a couple of auxiliary classes that need to be discussed. While value numbering we will need to know which expressions could have side effects and what kinds of side effects those are. This is accomplished by `bloat.trans.SideEffectsChecker`. `SideEffectsChecker` implements the `TreeVisitor` interface. When an expression tree `Node` is visited by a `SideEffectsChecker`, it compiles a bit vector showing the kinds of side effects it may have.

A `CatchExpr`, `StackExpr`, or `StackManipStmt` effects the stack. A `ZeroCheckExpr`, `NewMultiArrayExpr`, `NewArrayExpr`, `CastExpr`, `ArrayLengthExpr`, `ArrayRefExpr`, `CallStaticExpr`, `CallMethodExpr`, or `MonitorStmt` may throw an exception. A `CallStaticExpr` or `CallMethodExpr` involve invoking a method. `MonitorStmts` causes thread synchronizations. A `NewMultiArrayExpr`, `NewArrayExpr`, or `NewExpr` causes memory to be allocated. `RCExprs` and `UCExprs` cause residency and update checks, respectively. A `StoreExpr`, a `LocalExpr`, `StackExpr`, `ArrayRefExpr`, `FieldExpr`,

[4] I believe Dr. Comer would refer to this solution as "elegant".

or `StaticFieldExpr` that defines a variable all involve a store to memory. An `ArrayRefExpr` and a non-final `FieldExpr` cause an alias. `FieldExprs` and `StaticFieldExpr` that are volatile also have side effects.

Along the way we will need to be able to differentiate between `Nodes` in expression trees. The `bloat.trans.NodeComparator` class helps us do this. Its `equals` method compares two `Nodes` for equality. It also has a `hashCode` method for determining a unique number for each kind of `Node`. `Nodes` that are composed of other objects (for example, an `IfCmpStmt` is made up of a comparison and two `Blocks` representing the targets) have hash codes that are based on the hash codes of their composites. Most kinds of `Nodes` are equivalent to each other (for example, every `RetStmt` always has the same hash code). However, method calls are never considered equal.

The `bloat.trans.ValueFolder` class is used to determine whether or not a given `Node` can be replaced with another `Node` (usually a `ConstantExpr`). It is a `TreeVisitor` that recognizes things like algebraic identities and redundant checks. The `ValueFolder` may or may not actually replace the `Node` in the CFG with its simplified version. For instance, during value numbering no `Nodes` are replaced, but during value folding (see section 8.4) they are. `ValueFolder` contains a mapping between value numbers and their constant (`ConstantExpr`) value, if any. It also keeps track of which value numbers correspond to the allocation of new objects (`NewExpr`, `NewArrayExpr`, `NewMultiArrayExpr`). The following summarizes the simplifications that can take place.

`LocalExpr` If the `LocalExpr` resides within a `InitStmt` and it is the first target in the `InitStmt`, then it represents the `this` pointer.

`PhiJoinStmt` A `PhiJoinStmt` may be eliminated if it is meaningless (all of its operands have the same value number) or it is redundant (its value is already computed by another `PhiJoinStmt`). Each operand is examined to make sure it has no side effects.

`StoreExpr` If the expression being stored resides within a `CheckExpr`, the `CheckExpr` is brought outside of the `StoreExpr` to facilitate copy propagation. If the `StoreExpr` stores into a local variable (`LocalExpr`), then determine whether or not the value number of the expression being stored has been mapped to a constant (`ConstantExpr`). If so, replace the expression with constant.

`NewMultiArrayExpr/NewArrayExpr/NewExpr` Keep track of the value numbers of expressions that create new objects.

`RCExpr` If the expression being checked is itself an `RCExpr`, then the outer one is redundant. If the expression being checked is the `this` pointer or if the expression results from an allocation expression, we know that the expression will always be resident and the `RCExpr` can be removed.

`ZeroCheckExpr` If the expression being checked is a `ZeroCheckExpr`, then the outer one is redundant and can be removed. The `this` pointer and objects that were created by allocation expressions will never be `null`, so the `ZeroCheckExpr` can be removed. If the expression being checked evaluates to a non-zero constant, then the `ZeroCheckExpr` can be removed.

`UCExpr` If the expression being checked is also an `UCExpr`, then it is redundant and can be removed.

`ArithExpr` By examining the `ArithExpr` operator and the operands we attempt to take advantage of several algebraic properties such as identity and associativity. Of course, if both operands evaluate to constants, we can calculate the result and remove the `ArithExpr`.

`CastExpr` In the special cases when the empty string, `''''`, is cast to a `String` or `null` is cast to some object type, the `CastExpr` can be eliminated.

`NegExpr` If the expression being negated evaluates to a constant, the `NegExpr` can be replaced with the negated `ConstExpr`.

`ShiftExpr` If the `ShiftExpr` shifts zero bits (expression being shifted doesn't change) or shifts zero itself (always zero), we can replace it with the appropriate expression.

`IfZeroStmt` If the expression being tested evaluates to a constant, the `IfZeroStmt` can be replaced with a `GotoStmt` that always jumps to the appropriate target.

`IfCmpStmt` If the expressions being compared evaluate to constants then we can replace the `IfCmpStmt` with a `GotoStmt`. If one of the expression being compared evaluates to zero, we can replace the `IfCmpStmt` with an `IfZeroStmt`.

`SwitchStmt` If the index of the `SwitchStmt` evaluates to a constant, then the `SwitchStmt` can be replaced with a `GotoStmt` to the appropriate target.

Numbering Expressions

SSC-based value numbering is implemented in the static `transform` method of `bloat.trans.ValueNumbering`. It implements the algorithm found in figure 8.5. After creating the `SSAGraph` for the `FlowGraph` it creates an anonymous implementation of `ComponentVisitor` to do the work of the algorithm. It has two hash tables, the optimistic table and the valid table. Both tables are global to the algorithm. It initializes the value number of each component of the SCC to -1. If the SCC has only one component, then the private `valnum` method is invoked for the component using the optimistic table.

BLOAT implements the valid and optimistic tables as `HashMap`s that map `Node`s to `Tuple`s. Each `Tuple` consists of a `Node` and an integer hash code based on the hash code of the `Node` (see `NodeComparator`, section 8.3.2) and the number of children the node has in the SSA graph.

`Tuple` provides an `equals` method to determine if two `Tuple`s are equal with respect to their value numbers. If both `Tuple`s' `Node`s are `MemRef`s, then the `Tuple`s are always considered to be unequal. If the `Node`s are not equal when compared using a `NodeComparator`, then the `Tuple`s are unequal. If the `Node`s have a different number of children in the SSA graph (operands), they are unequal. If the `Node`s children in the SSA graph have different value numbers, the `Tuple`s are unequal. If the `Node`s are `PhiStmt`s, the order of children does not matter. So, if corresponding children of the two `Node`s have the same value numbers, the `Tuple`s are considered to be equal.

If a `Tuple` has not already been created for the `Node` being value numbered, `valnum` simplifies the `Node` by calling the `simplify` method and creates a `Tuple` for the simplified `Node` to which the original `Node` is mapped. The table (either the optimistic or the valid depending on how `valnum` was called) is searched for the `Node` corresponding to the `Tuple`. If the `Tuple` is mapped to a `Node`, that `Node`'s value number is used. Otherwise, a new value number is used. Finally, each of the `Node`s that the `Node` is question is equivalent to (in the SSA graph) is examined. If the equivalent `Node` has a value number that is different from the value number of the `Node` in question, the equivalent `Node` is assigned the new value number and the contents of the table is therefore changed.

The rest of the `ComponentVisitor` in the `transform` method pretty much follows the algorithm in figure 8.5. If the SCC has more than one component, `valnum` is invoked for each component `Node` in reverse post-order using the optimistic table. This process is repeated until the optimistic table does not change (that is, no `Node` equivalent to the component has its value number changed). Finally, `valnum` is invoked on each component using the valid table.

8.4 Value Folding

Once the value numbers for the `Node`s in the control flow graph have been calculated, the information gathered by a `ValueFolder` can be used to eliminate redundant `Node`s and to propagate constants through the CFG. This elimination is performed by `bloat.trans.ValueFolding`.

The `transform` method of `ValueFolding` uses a `ComponentVisitor` to visit each component of the SCCs in the `FlowGraph`'s SSA graph. This visitor creates a mapping between a component `Node` and the `Node` to which it can be folded. The private `fold` method is invoked on each component until the mapping does not change.

`fold` searches the mapping for the `Node` to which the SSC node can be folded. If the folded node has no parent or has not been assigned a value number, it cannot be folded. If the folded `Node` is a `ConstantExpr` and its value number was mapped to a different `ConstantExpr` (or nothing at all), the mapping between value numbers and their `ConstantExpr`s maintained in the `ValueFolder` is updated. If the folded node is a non-constant `Expr` that was mapped to a constant, replace all occurrences of the folded `Node` with a clone of the `ConstantExpr`. Map the component `Node` to the `ConstantExpr`. Note that if the folded `Node` resides inside a `PhiCatchStmt`, its value is not replaced. Finally, if the component `Node` is not mapped to any `Node`, a `ValueFolder` is used to fold the component `Node`. If the component `Node` is folded, the mapping is updated appropriately.

Back in `transform` once all of the strongly connected components have been folded, `removeUnreachable` is called on the `FlowGraph` and that's it.

8.5 Expression Propagation

Expression propagation performs constant and copy propagation. Constant propagation removes unnecessary assignments by replacing variables that are assigned constant values with those constant values. For example, given $a_1 \leftarrow 4$, all uses of a_1 are replaced with 4 and a_1 and $a_1 \leftarrow 4$ are removed. Copy propagation eliminates unnecessary assignments of variables to other variables. For instance, given $a_1 \leftarrow b_1$, all uses of a_1 with b_1 are replaced and $a_1 \leftarrow b_1$ is removed.

Expression propagation is implemented by the `transform` method of `bloat.trans.ExprPropagation`. `transform`, in turn, invokes the private `propagate` method until no more expression can be propagated (i.e. the control flow graph does not change).

`propagate` examines each statement in the control flow graph. If a `StoreExpr` that stores into a local variable (`LocalExpr`) is encountered, then there is a possibility that expression propagation can take place. If the right hand side of the `StoreExpr` is also a `StoreExpr`, then we have a nested store:

```
L := (M := E)
```

If `M` is also a local variable (`LocalExpr`), then all uses of `M` can be replaced with `L` and `L := (M := E)` can be replaced by `L := E`. The private `propExpr` performs the propagating. If the right hand side of the `StoreExpr` is a `LeafExpr` (a local variable or a constant), then all uses of the left hand side can be replaced by the right hand side and the `StoreExpr` can be removed.

`PhiStmt`s may also be candidates for expression propagation. In the case that all of a `PhiStmt`'s operands are the same (that is, they are all the same local variable or they are all the same constant value), the target of the `PhiStmt` may be replaced by any of the operands (they are all the same).

`propExpr` does the work of propagating one expression (a local variable) to the uses of another. If the expression being replaced is a local variable that is used as an operand to a `PhiStmt`, no use of that variable can be replaced. Otherwise, all uses are replaced. If the expression being replaced is not a local variable, then all uses of it that are not operands of a `PhiCatchStmt` are replaced. If all of the uses of an expression are replaced, `propExpr` returns `true`.

8.5.1 An Example of Expression Propagation

The below method is a rather contrived example of expression propagation.

```
int f(boolean b) {
  int x, y, z;
  x = 1;
  if(b)
    y = 1;
  else
    y = x;
```

```
  z = x + y;
  return(z);
}
```

The control flow graph in SSA form for this method is given in Figure 8.6. Note that the value of program variable `y` (`Li3`) will be the same regardless of which branch of the if statement is taken. By propagating the constant value of `x` (`Li2`), both of the operands of the ϕ-statement for `Li3` in block 13 will be `1`. Thus, the value of `Li3_17` defined by the ϕ-statement can be propagated to its use in the `eval` in block 13. The constant value of `Li2` is also propagated to that `eval`. After expression propagation block 13 looks like this:

```
<block label_13 hdr=label_22>
label_13
Li1_29 := Phi(label_6=Li1_1, label_11=0)
eval (Li4_8 := (1 + 1))
return Li4_8 caught by []
```

The ϕ-statement for `Li1` (program variable `b`) remains, but note that its second operand has a value of `0`. Expression propagation does a good job of eliminating dead code. The `eval`s in blocks 0, 6, and 11 as well as the ϕ-statement for `Li3` in block 13 are removed from the control flow graph.

8.6 Eliminating Persistent Checks

BLOAT was designed to work with a Java Virtual Machine that interacts with a store of persistent objects that has a cache of objects in volatile memory. Two important operations on persistent objects are residency checks and update checks. A residency check is an explicit instruction that ensures that the object at a given offset into the stack is resident in the object cache. An update check marks an object at a given offset in the stack as being "updated". An updated object must be written back to the stable store upon eviction from the cache.

BLOAT represents residency checks by `bloat.tree.RCExpr` and update checks by `bloat.tree.UCExpr`. Each performs a check on some other expression. A residency check is redundant when it can be proven that the object it checks is always resident. Examples of values that are always resident are the `this` pointer and a value that results from a `new` statement in that method.

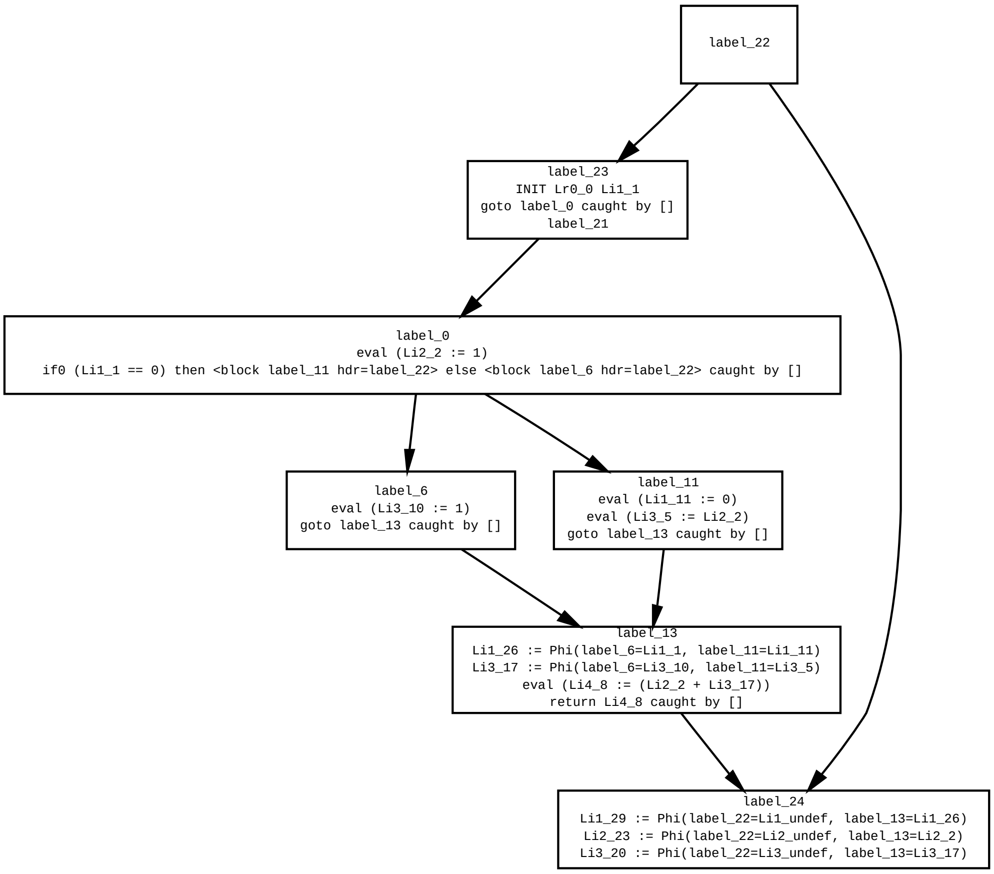


Figure 8.6: An Example of Expression Propagation

The class `bloat.trans.PersistentCheckElimination` examines the blocks in a control flow graph and removes persistent checks that are redundant. It eliminates checks in a manner similar to that of value folding (see section 8.3.2, page 122). The private `search` method does most of the work. The algorithm uses a bit vector for each type of check (`RC`, `AUPDATE`, and `SUPDATE`) to keep track of the value numbers on which checks have been performed. A pre-order traversal of the dominator tree of basic blocks is made. The nodes in each block's expression tree are visited in depth-first order.

When an expression that creates an object (`NewArrayExpr`, `NewMultiArrayExpr`, or `NewExpr`) is encountered, its value number is marked as being "seen" with respect to a residency check. When an `InitStmt` is encountered, the value number for the `this` pointer (the target of the first initialization in the `InitStmt`) is computed and is marked as being "seen" with respect to a residency check. When an `RCExpr` that checks an expression with a value number that has been "seen", the `RCExpr` may be redundant. If the expression being checked has no aliasing side effects (see section 8.3.2), then the `RCExpr` is replaced with the expression that it checks.

We can remove `UCExpr`s that check expressions that have already been updated by another `UCExpr`. When an `UCExpr` is encountered[5], we mark the value number of the expression being checked as being "seen". If some later `UCExpr` checks an expression with a value number that has been "seen", that `UCExpr` is redundant and is removed provided that it does not have any aliasing side effects.

8.7 Peephole Optimizations

Peephole optimizations consider a "window" of several consecutive instructions and look for places where particular characteristics of the instruction set may be exploited. For instance, a `push` instruction followed by a `pop` instruction is a useless operation. Peephole optimizations will recognize this fact and remove both instructions.

BLOAT performs peephole optimizations on Java bytecodes using the `bloat.trans.Peephole` class. Since peephole optimizations work on the bytecodes themselves, it is assumed that a method's control flow graph has already been converted back into bytecodes (see section 7.3). At this point, BLOAT operates on a method's `bloat.editor.MethodEditor` (see section 3.3.2).

[5]Remember that we perform a depth-first traversal of the expression tree. Children are visited first.

`Peephole`'s `transform` method performs peephole optimizations and then removes any unreachable code from the method. The method's code (`bloat.editor.Instructions` and `bloat.editor.Labels`) is visited in reverse so that redundant loads and stores may be eliminated in one pass. When an instruction that changes control flow (a goto) is encountered, some optimizations are performed. If the instruction that follows the goto is the target of the goto, the goto is useless and can be removed. Instructions that occur after the goto, but before a `Label` that starts a block, will never be executed and can be removed[6].

Then the peephole optimizations occur. Each pair of two consecutive `Instructions` is sent to the private `filter` method. `filter` looks for patterns of instructions that can be optimized. An instruction that pushes something onto the stack (ldc, $x$load, or dup) followed by a pop is a useless operation and both instructions can be removed. A load followed by a store to the same location is useless. A store instruction followed by a return is useless because all local variables are reset upon the return, so the store can be removed. Algebraic identities can be exploited. For example, a negation (ineg) followed by an add (iadd) can be replaced with a subtract (isub). Conditional instructions may be replaced by less expensive counterparts. For instance, an ldc 0 followed by an if_icmpeq can be replaced with an ifeq. Two consecutive stores to the same local variable can be replaced with a pop and a store. The result of the peephole optimizations (i.e. the new instruction(s)) are represented by an instance of the private `Filter` class. A `Filter` consists of an array of zero, one, or two `Instructions`.

If `filter` was successful in optimizing consecutive instructions, `transform` removes the old instructions and inserts the new ones. One last peephole optimization replaces jump instructions whose targets are themselves jumps (jumps to a jump) with the the target jump, thus removing the redundant jump. This is performed for both goto and switch instructions.

The final phase of the peephole optimizations is to remove unreachable code by invoking the `removeUnreachable` method. `removeUnreachable` makes a depth-first traversal of the instructions. It maintains a worklist of `Labels` that begin basic blocks. It also maintains a set of `Labels` that begin blocks whose code is known to be reachable. The first block in the method as well as all blocks that begin exception handlers (we must be conservative and assume they will be executed) are always reachable. The instructions in each of the reachable blocks is visited. When an instruction that jumps to another block is encountered, the label of the target block is added to

[6]I think this assumes that code was generated in trace order

the worklist. The code is iterated over one final time to remove unreachable code. All instructions that occur after a label that has not been marked as reachable by the above process are unreachable and are removed from the method.

8.8 Summary

BLOAT performs a number of optimizations on a Java method. Array initialization compaction replaces long sequences of bytecodes used for initializes arrays of integral types, with a loop that initializes arrays from a string stored in the constant pool. Dead code elimination takes advantage of SSA form to remove operations that do not contribute to the output of the method. Value numbering assigns numbers to the expressions in a control flow graph such that if two expressions have the same number, they have the same value. Value numbers are used in value folding, an optimization that replaces expressions with constants and removes redundent computations. Expression propagation removes unnecessary stores. Special optimizations are performed to remove unnecessary persistent checks. Finally, peephole optimizations are performed on generated bytecodes to remove instructions that have no meaningful effect.

Chapter 9

Type-Based Alias Analysis

BLOAT uses type-based alias analysis (TBAA) [DMM98] to determine the aliasing relationships between entities in Java programs. However, before TBAA can be properly discussed, BLOAT's class hierarchy management and type inferencing mechanism must be explained.

9.1 BLOAT's Class Hierarchy

Type-based alias analysis relies on knowing the type hierarchy of the objects in a program. BLOAT maintains the inheritance hierarchy using the `bloat.tbaa.ClassHierarchy` class. `ClassHierarchy` maintains two `bloat.util.Graph`s containing type information about classes. One `Graph` represents the class inheritance hierarchy ("who extends who"). Another `Graph` represents the interface implementation hierarchy ("who implements what"). The nodes in the graph are instances of the `TypeNode`, a class private to `ClassHierarchy` that stores a `bloat.editor.Type` for the class or interface represented by the node. The graphs are constructed such that a node's successor is its super class (or an interface that it implements).

A `ClassHierarchy` is constructed from a `Collection` of names of classes and a `bloat.editor.Editor` is used to obtain information about the classes. For each name in the `Collection` the private `addClass` method is invoked. `addClass` maintains a worklist of `Type`s to be added to the graphs and a list of `Type`s that are already in the graphs. The following process is repeated until there are no more `Type`s in the worklist. A `TypeNode` for the `Type` is created in both graphs. Then, a `bloat.editor.ClassEditor` for the `Type` is obtained from the `Editor`. From the `ClassEditor`, the `Type` of the superclass of the `Type` in question is obtained and an edge from the `Type`'s

`TypeNode` to the `TypeNode` representing its superclass is inserted into the "extends" graph. A similar procedure is carried out for the interfaces that a class implements.

To get the maximum amount of type information from a class, all types that the class references are added to the type worklist. The private `addType` method is invoked for the `Type` of a class's superclass, each interface it implements, each of its methods and fields, and each entry in its constant pool. `addType` extracts `Type` information from another `Type` and adds it to the worklist. If the `Type` is a method, then the `Types` of its parameters and its return `Type` are added to the worklist. If the `Type` is an array, the `Type` of its elements is added to the worklist. In any other case, if the `Type` does not represent one of the primitive (`int`, `float`, etc.) types, it is added to the worklist[1].

`ClassHierarchy` has a number of methods for obtaining information from the graphs. It has methods to obtain the superclass and subclasses of a given `Type`, the interfaces a `Type` implements, and classes that implement a given `Type`. It also has methods that can determine if one `Type` is a subtype of another. The `intersectType` method returns the intersection of two `Types`. The intersection of two types is the most refined type to which both `Types` may be assigned. If the types are assignment compatible (e.g. a `long` and an array type are **not** assignment compatible), then the `NULL Type` is returned. If one `Type` is a subtype of the other, the subtype is returned. Conversely, the `unionType` method computes the union of two `Types`. The union of two types is their most refined common supertype. If both types have a common supertype, then that `Type` is returned. Otherwise, the `java.lang.Object Type` is returned.

9.2 Type Inferencing

Type-Based Alias Analysis requires that the types of all of the entities in a program be known. For fields, method parameters, and method return types, this is not a problem, but the types of stack and local variables must be inferred. The `bloat.tbaa.TypeInference` class uses a simplified (intraprocedural) version of the type inference algorithm presented by Palsberg and Schwartzback in [PSb94]. A constraint model is used to compute

[1]This policy had to be modified due to performance. A `closure` parameter was added to `ClassHierarchy`. If closure is `false` only a class's superclass and interfaces are added to the hierarchy. Consequently, if a type is not present in the hierarchy when `getClassNode` or `getInterfaceNode` is called, we attempt to load it into the hierarchy.

types. Method parameters, the `this` variable, field references, and the result of method calls initialize the types of program variables. Assignments (including ϕ-statement) propagate type information.

The type inference process begins with `TypeInference`'s `transform` method. `transform` uses a `TreeVisitor` to determine the types of the variables initialized in each `InitStmt`. This type information is propagated to all uses of the `LocalExpr`s initialized in the `InitStmt`s. Additionally, all `Expr`s (except those `LocalExpr`s whose types have been initialized) have their type set to be undefined.

Type inferencing pays special attention to integral types by differentiating between `short`s, `char`s, `byte`s, `boolean`s, and `int`s. The range of values an integral type may taken on is represented by a bit vector (`BitSet`). `TypeInference`'s `typeToSet` method creates a bit vector representing a given integral type. For instance, if `typeToSet` is given a `short` type, all of the bit vector's bits between `MIN_SHORT` and `MAX_SHORT` will be set. There is also a `setToType` method that converts a bit set into the `Type` that minimally represents the range encoded in the bit set. The bit set representation is used so that computing the union of two integral type involves nothing more than "or-ing" two bit vectors together.

There are two helper methods that work with the constraint system, `start` and `prop`. `start` initializes a constraint by assigning a `Type` to a given expression (`Expr`). `start` will not assign an `Type` of undefined to an expression. If the expression already has a `Type` assigned to it, the expression's new `Type` is the union of the old type and the new type as determined by the `ClassHierarchy` (see section 9.1). If both the expression and the new type are integral types, then the new type is computed by "or-ing" together the two bit vectors that represent the two types. Finally, the type of each `Node` equivalent (in the `SSAGraph`) to the expression is set to the new type. The `prop` method propagates type information from one `Expr` to another. It uses `start` to do all of the real work.

Each component in the SCC of the `FlowGraph`'s `SSAGraph`[2] is visited by a `TypeInferenceVisitor`. The `TypeInferenceVisitor` is a `TreeVisitor` that does most of the type inference work. Depending on the kind of `Expr` being visited, a constraint is either started or propagated.

`ShiftExpr` If the expression being shifted is an integral type, then the `ShiftExpr` has type `Type.INTEGER`. Otherwise, the type of the expression being shifted is propagated to the type of the `ShiftExpr`.

---

[2]Oh yes, they're back!

`ArithExpr` If either of the `ArithExpr`'s operands is an integral type, then the entire `ArithExpr` has a type of `Type.INTEGER`. If the operation being performed is one of the compare operations (`ArithExpr.CMP`, etc.), then the `ArithExpr` has a type of `Type.INTEGER`. Otherwise, the type of the left-hand operand is propagated to the `ArithExpr`.

`NegExpr` If the expression being negated has an integral type, then the type of the `NegExpr` is `Type.INTEGER`. Otherwise, the type of the expression being negated is propagated to the `NegExpr`.

`ReturnAddressExpr` A `ReturnAddressExpr` always has type `Type.ADDRESS`.

`CheckExpr` A `CheckExpr` always has the same type as the expression it checks.

`InstanceOfExpr` An `InstanceOfExpr` always has a type of `Type.INTEGER`[3].

`ArrayLengthExpr` An `ArrayLengthExpr` always has a type of `Type.INTEGER`.

`VarExpr` If a `VarExpr` does not define a variable, then the type of the `VarExpr` defining expression is propagated to the `VarExpr`.

`StackManipStmt` A `StackManipStmt` is essentially an assignment from a set of "source" stack variables to a set of "destination" stack variables. Type information is propagated appropriately depending on the kind of `StackManipStmt`. Because it is a `Stmt`, its children are visited.

`StoreExpr` The type information of the expression on the right-hand side of the store is propagated to the left-hand side of the store. The type of the left-hand side of the store is propagated to the entire `StoreExpr`.

`CatchExpr` The type of the exception being caught is propagated to the type of the `CatchExpr`. If the type of the exception being caught is unknown, then `Type.THROWABLE` is used.

`PhiStmt` The type information of each of the `PhiStmt`'s operands is propagated to its target. Recall that the union of types is taken. If an operand is an undefined local variable, the type of the target is propagated back to the operand so it, too, will have a type.

[3]BLOAT does not make the distinction between boolean types and integral types.

ArrayRefExpr If the `ArrayRefExpr` is not a definition and its array type is defined and meets certain criteria (is not `Type.OBJECT`, nor serializable, nor cloneable, nor null), then the type of the array is propagated to the type of the `ArrayRefExpr`.

CallMethodExpr Because all of BLOAT's analysis is intraprocedural, the only propagation that occurs is the propagation of the return type of the method to the type of the `CallMethodExpr`.

CallStaticExpr The return type of the method is propagated into the `CallStaticExpr`.

CastExpr Type type to which the expression is being cast is propagated into the `CastExpr`.

ConstantExpr The type of the constant is propagated into the type of the `ConstantExpr`. If the constant's type is integral, then special care is taken to assign it the correct integral type based on its value.

FieldExpr If the `FieldExpr` is not a definition, the type of the field is propagated to the type of the `FieldExpr`.

NewArrayExpr If the element type of the array being created is defined, then the type of an array of the element type is propagated to the `NewArrayExpr`.

NewExpr The type of the object being created is propagated to the type of the `NewExpr`.

NewMultiArrayExpr If the element type of the array being created is defined, then the type of an array of those elements is propagated to the `NewMultiArrayExpr`.

StaticFieldExpr If the `StaticFieldExpr` is not a definition, then the type of the field is propagated to the type of the `StaticFieldExpr`.

9.2.1 An Example of Type Inferencing

The following program gives a simple example of what type inferencing can do.

```
public class Types {
  Object f(boolean b) {
```

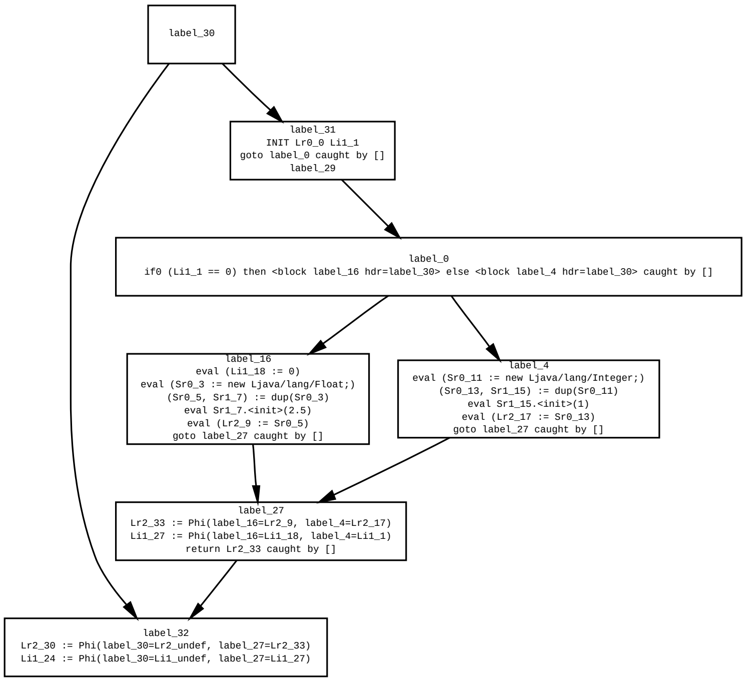


Figure 9.1: Control Flow Graph Demonstrating Type Inferencing

```
    Object o;
    if(b)
      o = new Integer(1);
    else
      o = new Float(2.5);
    return(o);
  }
}
```

The control flow graph in SSA form for this method is given in Figure 9.1. Table 9.1 gives the types of selected expressions in the method before type inferencing is performed. `Lr0_0`, the `this` pointer, has the expected `LTypes;` type. The first `Lr1_1` is from block 31 and has the correct `boolean` (`Z`) type. However, because its comparison to `0` in block 0, `Lr1_1` takes on an `integer` type. At this point, all objects have type `Ljava/lang/Object;`.

Table 9.2 shows the types of selected expressions after type inferencing. The type of `Li1_1` in block 0 is correctly identified as being `boolean`. The types of the `new` expressions are more precisely identified as `Ljava/lang/-`

Expression	Type
`Lr0_0`	LTypes;
`Li1_1` (block 31)	Z
`Li1_1` (block 0)	I
`new Ljava/lang/Integer;`	Ljava/lang/Object;
`Lr2_17`	Ljava/lang/Object;
`new Ljava/lang/Float;`	Ljava/lang/Object;
`Lr2_9`	Ljava/lang/Object;
`Lr2_33`	Ljava/lang/Object;

Table 9.1: Types Before Type Inferencing

Expression	Type
`Lr0_0`	LTypes;
`Li1_1` (block 31)	Z
`Li1_1` (block 0)	Z
`new Ljava/lang/Integer;`	Ljava/lang/Integer;
`Lr2_17`	Ljava/lang/Integer;
`new Ljava/lang/Float;`	Ljava/lang/Float;
`Lr2_9`	Ljava/lang/Float;
`Lr2_33`	Ljava/lang/Number;

Table 9.2: Types After Type Inferencing

`Integer;` and `Ljava/lang/Float;`. These types are also propagated to `Lr2_17` and `Lr2_9`. The local variable `Lr2_33` is the result of a ϕ-statement merging `Lr2_9` and `Lr2_17`. Its type should be the most refined common supertype of `Ljava/lang/Integer;` and `Ljava/lang/Float`. Type inferencing correctly identifies this type as `Ljava/lang/Number;`.

9.3 Type-Based Alias Analysis

Alias analysis examines pointers and disambiguates memory references so that instructions may be reordered safely. Many alias analyses have undesirable properties such as being slow and requiring an entire program. Type-based alias analysis [DMM98] performs alias analysis on a program written in a type-safe language such as Java and uses type declarations to

Notation	Name	Variable accessed
p.f	Field access	Field `f` of class instance to which p refers
$p[i]$	Array access	Component with subscript i of array to which p refers

Table 9.3: Elements of an Access Path

disambiguate memory references.

An *access path* is a non-empty sequence of memory references (see Table 9.3). An example access path would be $a.b[i].c$. If a lexically identical occurrence of an access path occurs in a program, it may be redundant. However, the fact that elements of an access path reference memory locations complicates matters. Some other program entity may reference the same memory location as one of the elements of the access path. In that case, we cannot guarantee that the second occurrence of the access path is truly redundant.

The declared (compile-time) type of an access path is denoted $Type(p)$. Any type that may be assigned to $Type(p)$ occurs in $Subtype(Type(p))$. TBAA considers the types and subtypes of each variable in a method to disambiguate references. Basically, if we know that two variables have incompatible types, then we can guarantee that they can never reference each other. Two access paths p and q may be aliases only if the $TypeDecl(p, q)$ relation holds:

$$TypeDecl(\mathcal{AP}_1, \mathcal{AP}_2) \equiv Subtypes(Type(\mathcal{AP}_1)) \cap Subtypes(Type(\mathcal{AP}_2)) \neq \emptyset$$

The alias test is further refined by considering an object's fields and is presented in Table 9.4. BLOAT uses the *FieldTypeDecl* relation to determine the alias relationship between two access paths.

9.3.1 Implementation

BLOAT's implementation of typed-based alias analysis is relatively straightforward. `MemRefExpr`s are expressions that reference locations in memory. Thus, an access path may consist of field references (`FieldExpr` or `StaticFieldExpr`) and array references (`ArrayRefExpr`). The class `bloat.tbaa.TBAA` has one method, `canAlias`, that determines whether or not two `Expr`s can alias each other.

Case	$\mathcal{AP}_1$	$\mathcal{AP}_2$	$\mathit{FieldTypeDecl}(\mathcal{AP}_1, \mathcal{AP}_2)$
1	p	p	true
2	$p.\mathtt{f}$	$q.\mathtt{g}$	$(\mathtt{f} = \mathtt{g}) \wedge \mathit{FieldTypeDecl}(p, q)$
3	$p.\mathtt{f}$	$q[i]$	false
4	$p[i]$	$q[j]$	$\mathit{FieldTypeDecl}(p, q)$
5	p	q	$\mathit{TypeDecl}(p, q)$

Table 9.4: $\mathit{FieldTypeDecl}(\mathcal{AP}_1, \mathcal{AP}_2)$

If either of the `Expr`s is not a `MemRefExpr`, then they obviously cannot alias each other. If the two `Expr`s are equal, then they can alias each other. Then the rules of *FieldTypeDecl* are followed. If one of the expression references an array and another references a field, then they cannot alias each other. If both expressions reference arrays, then each expression is checked to make sure that it is a valid array reference (i.e. the indices are integral and the array type is indeed an array type). If both are valid, then an optimization is performed. If both array references have constant indices of different value, then the array references cannot alias each other. Otherwise the `ClassHierarchy` is consulted to determine whether or not the types of the elements of the two arrays being referenced intersect. This is the *TypeDecl* relationship.

In any other case, both expressions reference fields. If either of the fields is volatile[4], then they may be aliases. If either of the fields is final, they cannot be aliases. If the fields have the same name, they may alias each other. Finally, if all else fails, the *TypeDecl* relationship is applied to the types of the two fields.

9.4 Summary

BLOAT uses type-based alias analysis to determine whether or not two access paths (memory references) and refer to the same object. If a redundant access path can alias another, the redundant access path cannot be removed. Type-based alias analysis requires type information that is maintained in a class hierarchy. Because type information for local and stack variables is not explicit in Java bytecode, it must be inferred.

[4]Recall that a volatile field is always reloaded from memory upon its use. It is expected that a volatile field's contents will be modified by other threads.

Chapter 10

Partial Redundancy Elimination

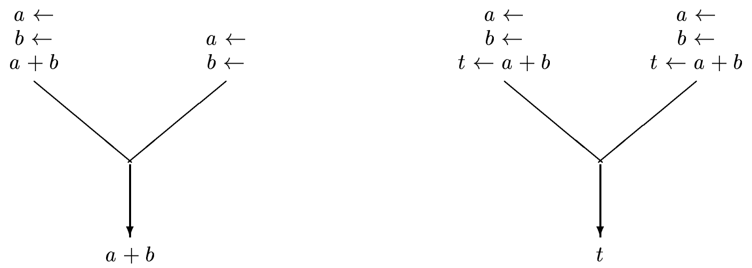
Well, folks, it's the chapter we've all been dreading: SSAPRE. Legend has it that it took Nate two weeks to get a feeling for it and a full six weeks to really understand it. I stared at Chow's paper for a week and was still clueless. However, now that we all know about SSA form and control flow graphs, it shouldn't be that bad. Right?

10.1 Background

In the world of optimization calculating something twice is bad. Let's suppose that a program calculates an expression, $a + b$, and then a little while later calculates $a + b$ again. If neither a nor b has changed, then the second calculation of $a+b$ is redundant. We could save the result of the first calculation of $a+b$ to a temporary variable and then replace the second calculation of $a + b$ with the temporary, thus avoiding the redundant calculation.

Partial Redundancy Elimination (PRE) eliminates redundant computations of expressions that do not necessarily occur on all control flow paths that lead to a given redundant computation. An example is given in Figure 10.1. The expression $a+b$ is redundant along the left-hand control flow path, but not the right. PRE recognizes this fact, and inserts a computation of $a+b$ along the right-hand path. Thus, the computation of $a+b$ at the merge point is fully redundant and can be replaced with a temporary variable.

PRE has been around for a while, but it used bit vectors (which do not work well with SSA form) to represent partially redundant expressions. Fred Chow and friends [CCK$^+$97] formulated a method for performing PRE on



(a) Before PRE (b) After PRE

Figure 10.1: Partial Redundancy Elimination

a control flow graph in SSA form, SSAPRE.

SSA form is mainly concerned with variables. Recall that each SSA variable has a unique definition. When a merge of control flow occurs, a ϕ-statement is used to factor together SSA variables that are available along the various incoming paths. Representing expressions in SSA form is kind of awkward. An expression is not "defined" like a variable is. SSA variables also complicate matters. Is $a_1 + b_1$ the same as $a_2 + b_3$? SSAPRE works with *lexically identical* expressions and ignores differences in SSA variable numbers. That is, $a_1 + b_1$ and $a_2 + b_3$ are lexically identical. Additionally, expressions are referred to by a hypothetical temporary, h, that represents a variable to which the expression could be assigned ($h \leftarrow a + b$). When expressions (represented by their hypothetical temporaries) reach a control flow merge point, they are merged together using a Φ-statement similar to the ϕ-statement for variables.

There are six steps to SSAPRE: Φ-Insertion, renaming, computing "down safety", determining where an expression "will be available", finalization, and code motion. Φ-insertion and renaming are similar to steps in SSA form conversion. Down safety and "will be available" determine which expressions are partially redundant and where additional computations need to be inserted. Finalization and code motion insert the additional computations and replace redundant computations with temporaries.

10.1.1 Φ-insertion

The input to SSAPRE is a control flow graph in SSA form that has had its critical edges removed. Recall that critical edges connect a block with more than one predecessor to a block with more than one successor. Critical

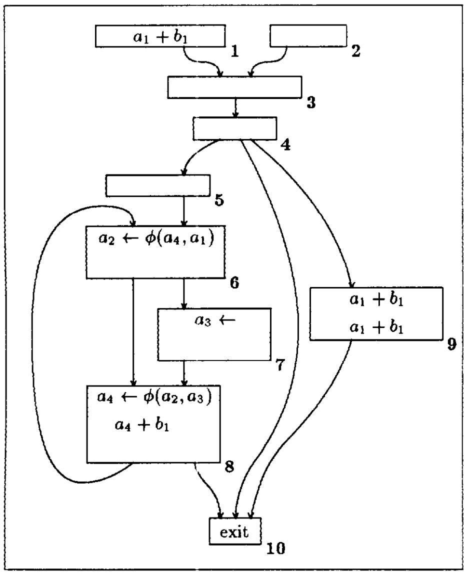


Figure 10.2: An example program

edges were removed by inserting an empty block along the edge (see Section 5.6.9). The control flow graph given in Figure 10.2 will serve as an example throughout the explanation of the SSAPRE algorithm[1].

A Φ-statement is needed whenever different values of an expression reach a common point in the program. There are two situations in which Φ-statements must be inserted. First of all, Φ-statements are inserted in the blocks in the iterated dominance frontier (see section 5.6.2) of the blocks in which the expression occurs. A Φ-statement is inserted in block 3 of Figure 10.3 because block 3 is in the iterated dominance frontier of block 1 (in which an occurrence of $a + b$ occurs). A Φ-statement must also be inserted in a block containing a ϕ-statement for one of the variables in the expression. The ϕ-statement signifies that the variable has been redefined and thus the expression in which the variable is used, may have a new value. The Φ-statements in blocks 6 and 8 in Figure 10.3 are inserted because the contain ϕ-statements for variable a. Note that it is not necessary to insert a Φ-statement in block 10 because there is no occurrence of the expression after block 10.

Figure 10.4 gives the algorithm for performing Φ-insertion. The set $DF\_phis[i]$ is used to keep track of the Φ-statements for expression E_i in-

[1] These figures and algorithms were taken from Fred Chow's SSAPRE paper [CCK+97].

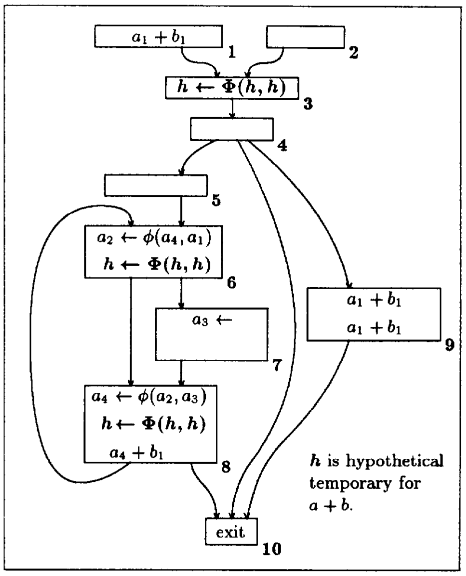


Figure 10.3: Inserting Φ-statements

serted due to the first (dominance frontier) situation. The set $Var\_phis[i][j]$ is used to keep track of the Φ-statements for expression E_i inserted due to the second (ϕ-statement defining the j^{th} variable in E_i) Φ-insertion criterion.

10.1.2 Renaming

Once Φ-statements of the form $h \leftarrow \Phi(h, h)$ have been inserted into the control flow graph, version numbers are assigned to the hypothetical temporaries, h. Occurrences of an expression with identical h-version have identical values and any control flow path that crosses two different h-versions must cross a definition of one of the expressions operands or a Φ-statement for the expression. The renaming algorithm for SSAPRE is similar to that of SSA construction except that there is a renaming stack for each (lexically distinct) expression.

An expression, h, may occur in one of three forms:

Real occurrence An evaluation of the expression as it occurs in the original program

Φ-statement The result (target) of a Φ-statement that was inserted during the previous step

procedure Φ-Insertion **begin**
  for each expression E_i **do**
    $DF\_phis[i] \leftarrow \emptyset$
    for each variable j in E_i **do**
      $Var\_phis[i][j] \leftarrow \emptyset$
  for each occurrence X of E_i in program **do**
    $DF\_phis[i] \leftarrow DF\_phis[i] \cup IDF(X)$
    for each variable occurrence v in X **do**
      if (V is defined by a ϕ-statement) **then**
        $j \leftarrow$ index of V in X
        $Set\_var\_phis(Phi(V), i, j)$
  for each expression E_i **do**
    for each variable j in E_i **do**
      $DF\_phis[i] \leftarrow DF\_phis[i] \cup Var\_phis[i][j]$
    insert Φ-statements for E_i according to $DF\_phis[i]$

procedure $Set\_var\_phis(phi, i, j)$ **begin**
  if ($phi \notin Var\_phis[i][j]$) **then**
    $Var\_phis[i][j] \leftarrow Var\_phis[i][j] \cup phi$
    for each operand V in phi **do**
      if (V is defined by a ϕ-statement) **then**
        $Set\_var\_phis(Phi(V), i, j)$

Figure 10.4: Algorithm for Φ-insertion

Φ-operand An operand of a Φ-statement: an evaluation of the expression that can be considered to occur in the predecessor block

The renaming algorithm performs a pre-order traversal of the dominator tree of the control flow graph and handles each occurrence, q, of a given expression as follows. If the occurrence is a Φ-statement, the occurrence is assigned a new version number. That is, the hypothetical defined by the Φ-statement is given a new version number. Otherwise, the variables that comprise the expression are considered. If each of the SSA versions on top of each variable's renaming stack matches the SSA versions of the variables in the expression, the occurrence q is assigned the version number on top of the expression's renaming stack. If any of the variables' version numbers do not match, then one of two situations occurs. If q is a real occurrence, then it is assigned a new version number. Otherwise, if q is a Φ-operand, then it is assigned the version number $\perp$ signifying that the expression is not computed along the control flow path corresponding to that Φ-statement operand.

Figure 10.5 shows the results of renaming the hypotheticals. Since the occurrence of $a + b$ in block 1 is the first occurrence, it is assigned a new version number (h_1). There is no occurrence of $a + b$ along the path (block 2) corresponding to the second operand of the Φ-statement in block 3, so it has the $\perp$ version number. Because neither a nor b is modified between the Φ-statement in block 3 and the real occurrence of $a + b$ in block 9, the occurrences in block 9 have the same version number as the Φ-statement in block 3. The second operand of the Φ-statement in block 8 is $\perp$ because a is defined in it predecessor block, block 7, but $a + b$ is not recomputed before its occurrence as a Φ-statement operand in block 7.

10.1.3 Computing Down Safety

In order for a computation of an expression to be inserted, the expression should be *down safe* at that point. Come to find out, we only care about the down safety of Φ-statements. A Φ-statement is down safe if all control flow paths from that Φ-statement to the program (method) exit either: (1) recompute the expression, or (2) redefine one of the variables in the expression. The only circumstances in which a Φ-statement will **not** be down safe are: (a) if there is a path from the Φ-statement to the exit on which the result of the Φ-statement is not used, or (b) if there a path to the exit on which the result of the Φ-statement is used only as an operand to another Φ-statement which itself is not down safe. The result of a down safe Φ-statement will be

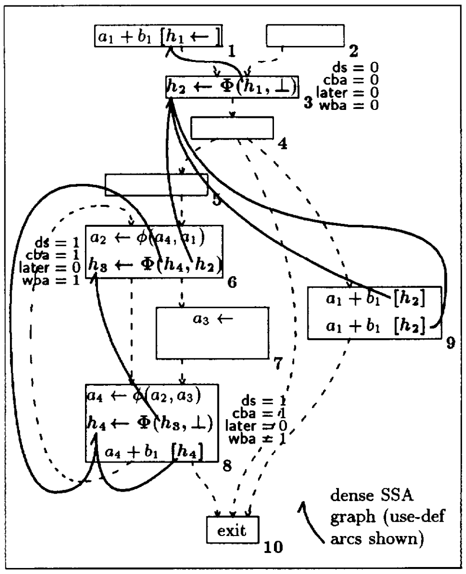


Figure 10.5: After renaming

used at least once on all paths from the Φ-statement to the exit node. If a Φ-statement is not down safe, it is not worth our while to add an earlier evaluation of the expression. From the above definition, it becomes clear that computing down safety is a backwards data flow problem.

Down safety information begins at Φ-statements meeting condition (a) and propagates backward in the program until condition (b) is not satisfied. Condition (b) will be satisfied until a real occurrence of the expression is encountered. Accordingly, each of the Φ-statement's operands has a "has real use" flag that is set when the operand (one of those "hypothetical temporaries") is defined by a real occurrence. Initially, all Φ-statements are marked as being down safe and all operands have their "has real use" flag set to false. The initially (condition (a)) down safety Φ-statements and the "has real use" flags can be computed during the renaming step. Recall that the renaming step makes a pre-order (with respect to the control flow graph's dominator tree) traversal of the SSA graph. When a new version number is assigned to a real occurrence of the expression or when program exit is encountered, if the top of the expression's rename stack is a Φ-statement, then that Φ-statement is not down safe because the version it defines is not used along the path to a real occurrence (or exit). When a version number is assigned to a Φ-statement's operand, its "has real use" flag is set if and

procedure *DownSafety* **begin**
  for each Φ-statement F in the program **do**
    if (**not**($down\_safe(F)$)) **then**
      for each operand o of F **do**
        $Reset\_downsafe(o)$

procedure $Reset\_downsafe(X)$ **begin**
  if ($has\_real\_use(X)$ **or** X not defined by a Φ-statement) **then**
    return
  $F \leftarrow$ Φ-statement that defines X
  if (**not**($down\_safe(F)$)) **then**
    return
  $down\_safe(F) \leftarrow false$
  for each operand o of F **do**
    $Reset\_downsafe(o)$

Figure 10.6: Computing Down Safety

only if the version on top of the expression's renaming stack is the same as the operand's and is defined by a real occurrence.

After the initial conditions are set up during renaming, the down safety information is propagated through the program using the algorithm found in Figure 10.6. Let us again consider Figure 10.5. The "has real use" flag of the first operand of the Φ-statement in block 3 is set because h_1 is defined by a real occurrence in block 1. Hypotheticals h_2 (second operand of the Φ-statement in block 6) and h_3 (first operand of Φ-statement in block 8) have false "has real use" flags because they are defined by Φ-statements. h_4 is a little peculiar. Its "has real use" flag is indeed set even though it is defined by a Φ-statement. The real occurrence of $a+b$ in block 8 causes the version on top of $a+b$'s renaming stack to correspond to a real occurrence. Thus h_4 has a real use[2]. Now let us turn out attention to the Φ-statements themselves. The Φ-statement in block 3 is not down safe because there is a path from the Φ-statement to the exit block (block 3, block 4, block 10) on which $a+b$ is not recomputed and neither a nor b is redefined. Both the Φ-statements in block 6 and in block 8 are down safe because $a+b$ is recomputed in block 8 before exit.

[2]Chow says an operand has a real use when "the path to the Φ operand crosses a real occurrence of the same version of the expression". I guess this case fits that description.

10.1.4 Will Be Available

The "will be available" step determines whether or not an expression's value will be available at each Φ-statement after insertions for PRE. The first step is to determine which Φ-statements "can be available"[3]. Initially all Φ-statements can be available. Then, all all Φ-statements that are not down safe and have at least one $\perp$ operand are marked as can **not** be available. The can't be available property is propagated along the def-use arcs of the SSA graph to other Φ-statements that are not down safe. The propagation stops when an operand is encountered that has a real use. Φ-statement operands that are not "can be available" are set to $\perp$ along the way. In Figure 10.5 the Φ-statement in block 3 cannot be available because it is not down safe and one of its operands is $\perp$. Because the Φ-statements in blocks 6 and 8 are down safe, they can also be available.

At this point, a Φ-statement is not "can be available" if and only if no down-safe[4] placement of computations can make the expression available. The Φ-statements that can be available designate the range of down-safe program areas for insertion of the expression, plus areas that are not down-safe but where the expression is fully available in the original program. The entry points to this region (the $\perp$-valued Φ operands) can be thought of as SSAPRE's earliest insertion points. At least that's what Fred says.

Next, the algorithm determines the latest palces in the program at which expressions can be inserted. A Φ-statement's "later" predicate is used. Initially, "later" is true whenever a Φ-statement can be available. Starting with the real occurrences of the expressions (the operands that have a real use), the false value of later is propagated from an operand to the Φ-statement in which it occurs. In Figure 10.5 the Φ-statement in block 3 is not later because its first operand has a real use. This lack of laterness is propagated to the Φ-statement in block 6 because the hypothetical defined by the Φ-statement in block 3, h_2, is used as an operand to Φ-statement in block 6. It is also propagated to the Φ-statement in block 8 because h_3 is an operand to that Φ-statement.

The value of the "will be available" predicate is given by:

$$will\_be\_avail = can\_be\_avail \wedge \neg later$$

[3] I'm surprised that Φ-statements don't have a "should be available next Thursday" flag. Sheesh.

[4] Remember a computation is down safe if all control flow paths from the expression to exit contain a recomputation of the expression or an alteration of one of the variables in the expression.

```
procedure Compute_can_be_avail begin
  for each Φ-statement F in the program do
    if (not(down_safe(F)) and can_be_avail(F) and ∃ an operand of F that is ⊥) then
      Reset_can_be_avail(F)

procedure Reset_can_be_avail(G) begin
  can_be_avail ← false
  for each Φ-statement F with operand o defined by G do
    if (not(has_real_use(o))) then
      set that Φ-operand to ⊥
      if (not(down_safe(F)) and can_be_avail(F)) then
        Reset_can_be_avail(F)

procedure Compute_later begin
  for each Φ-statement F in the program do
    later(F) ← can_be_avail(F)
  for each Φ-statement F in the program do
    if (later(F) and ∃ an operand o of F such that (o ≠ ⊥  and has_real_use(o))) then
      Reset_later(F)

procedure Reset_later(G) begin
  later(G) ← false
  for each Φ-statement F with operand o defined by G do
    if (later(F)) then
      Reset_later(F)

procedure WillBeAvail begin
  Compute_can_be_avail
  Compute_later
```

Figure 10.7: Will Be Available

The algorithm for computing "will be available" is given in Figure 10.7. Additionally, the "insert" predicate for a Φ-operand holds if and only if:

1. The Φ-statement will be available; and

2. The operand is ⊥, or it does not have a real use and is defined by a Φ-statement that will not be available.

10.1.5 Finalize

The finalize step transforms the SSA graph for the hypothetical temporary representing an expression into a valid SSA form in which no Φ-statement operand is $\perp$. The finalize step performs the following tasks:

1. It decides whether or not a real occurrence of the expression should be computed at that point or reloaded from a temporary (the "reload" flag). If the expression is to be computed, it also decides whether or not the result should be saved to a temporary (the "save" flag).

2. For Φ-statements whose "will be available" flag is true, computations of the expressions are inserted along the incoming edges on which the expression is not available (operand is $\perp$).

3. Φ-statements whose "will be available" flag is true may be transformed into ϕ-statements for the temporary variable. Φ-statements that will not be available are taken out of consideration. Edges in the SSA Graph that reference these Φ-statements are fixed up to refer to other (real or inserted) occurrences.

These tasks are performed with the help of the *Avail_def* table for each expression. The indices into *Avail_def* are the SSA versions for the hypothetical temporary, h. So, $Avail\_def[x]$ will refer to the definition of h_x, either a real occurrence or a Φ-statement that "will be available". The finalize algorithm performs a pre-order traversal of the control flow graph's dominator tree. When it encounters a defining occurrence, its value will be saved into a temporary, t. If another occurrence references t, it will either be a redundant computation that can be replaced by t or a Φ-operand of a Φ-statement that will be transformed into a ϕ-statement with t as an operand. The finalize step handles each kind of expression occurrence as follows.

Φ-statement If it will not be available[5], nothing interesting happens. Otherwise, it is a defining occurrence for h_x and $Avail\_def[x]$ is set accordingly.

A real occurrence Real occurrences have a hypothetical, h_x, associated with them. If $Avail\_def[x]$ is defined, but the defining occurrence does not dominate the current (real) occurrence, the current occurrence also defines h_x. $Avail\_def[x]$ is set to the current occurrence. If the defining occurrence does dominate the current occurrence, then we can reload the value from the temporary. Thus, the "save" flag is set for $Avail\_def[x]$ and the "reload" flag is set for the current occurrence.

[5] Reminds me of most of the women I try to date.

Φ-operand If the operand's Φ-statement will not be available, nothing happens. Otherwise, if the operand's "insert" flag is set, a computation of the expression is inserted at the end of the block corresponding to the operand (the predecessor block). If the operand's "insert" flag is not set, then the "save" flag of $Avail\_def[x]$ (the Φ-operand has hypothetical h_x) is set and the operand is updated to refer to $Avail\_def[x]$.

The finalize algorithm is given in Figure 10.9. The result of performing the finalize step the control flow graph of Figure 10.5 is shown in Figure 10.8. Block 9 is interesting. Because $Avail\_def[2]$ $(a + b)$ found in block 1 does not dominate block 9, the first $a + b$ in block 9 in also a defining occurrence. So, it is assigned a new version number. (It would have been nice if this had been stated somewhere in the algorithm.) Consequently, the version number of the second real occurrence of $a + b$ in block 9 is changed to h_7. Let us also consider block 8. Since the Φ-statement's second operand satisfies insert (because its Φ-statement will be available and the operand is $\perp$), a computation of $a + b$ is inserted at the end of the preceding block (and given the version h_6). The same thing happens to the second operand of the Φ-statement in block 6. However, that operand satisfies "insert" because its Φ-statement will be available, the operand does not have a real use, and the Φ-statement that defines the operand will not be available.

10.1.6 Code Motion

Now that the hypothetical temporaries, h, are in valid SSA form, real temporaries, t, can be inserted into the program to eliminate redundant computations. The code motion step makes a traversal of h's SSA graph. When a real occurrence with a set "save" flag is encountered, the result of the real occurrence is saved into a new temporary, t. If the real occurrence's "reload" flag is true, then the real occurrence is replaced by a use of t. When one of the real occurrences that was inserted during the finalize step is encountered, its value is saved into a new t. When a Φ-statement is encountered, it is replaced by an equivalent ϕ-statement.

Figure 10.10 shows our long-suffering control flow graph after code motion. No ϕ-statement was generated for the Φ-statement in block 3 because one of its operands was $\perp$. The real occurrence inserted in blocks 5 and 7 were replaced with a computation. The Φ-statements in blocks 6 and 8 were replaced with ϕ-statements. The defining real occurrence in block 9 had its value saved to a temporary which was then used in place of the redundant real occurrence.

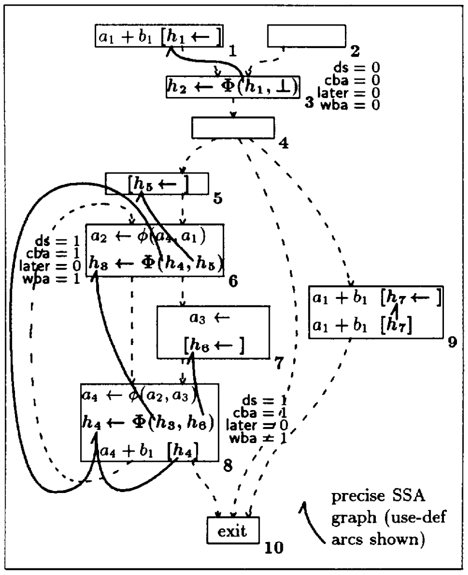


Figure 10.8: After finalize

10.1.7 But, wait. There's more!

The first part of this chapter presents the fundamental idea of SSA-based Partial Redundancy Elimination. However, there are a couple of optimizations that can be made to the algorithm to improve its efficiency.

Using a Worklist

Because SSAPRE is a sparse algorithm, storage space can be reduced if the lexically distinct expressions are handled one at a time. Thus, a worklist of expressions can be maintained. The "collect occurrences" step examines the program (method in our case) and builds a worklist of expressions that need to be worked on by SSAPRE. The collect occurrences step only considers *first order* (non-nested) expressions. So, for the expression $(a + b) - c$ only $a + b$ will be entered into the worklist. However, if as a result of SSAPRE $a + b$ is replaced by a temporary, t, the code motion step will add the new first order expression $t - c$ into the worklist.

Delayed Renaming

Because we are now using the worklist-driven approach, the renaming step does not need to pass over the entire control flow graph. However, the orig-

```
procedure Finalize_visit(block begin
  for each occurrence X of E_i in block do
    save(X) ← false
    reload(X) ← false
    x ← version(X)
    if (X is a Φ-statement) then
      if (will_be_avail(X)) then
        Avail_def[i][x] ← X
    else if (Avail_def[i][x] is ⊥  or Avail_def[i][x] does not dominate X) then
      Avail_def[i][x] ← true
    else if (Avail_def[i][x] is a real occurrence) then
      save(Avail_def[i][x]) ← true
      reload(X) ← true
  for each S in Succ(block) do
    j ← WhichPred(S, block)
    for each Φ-statement F in S do
      if (will_be_avail(F)) then
        i ← WhichExpr(F)
        if (j^th operand of F satisfies insert) then
          insert E_i at the exit of block
          set j^th operand of F to inserted occurrence
        else
          x ← version(j^th operand of F)
          if (Avail_def[i][x] is real) then
            save(Avail_def[i][x] ← true)
            set j^th operand of F to Avail_def[i][x]
  for each K in Children(DT, block) do
    Finalize_visit(K)

procedure Finalize begin
  for each version x of E_I in program do
    Avail_def[i][x] ← ⊥
  Finalize_visit(Root(DT))
```

Figure 10.9: Finalize

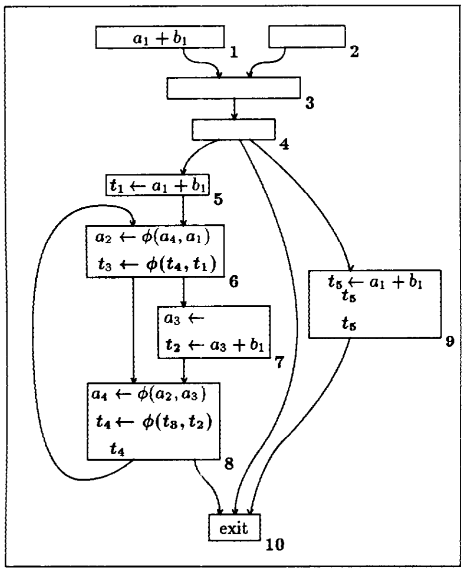
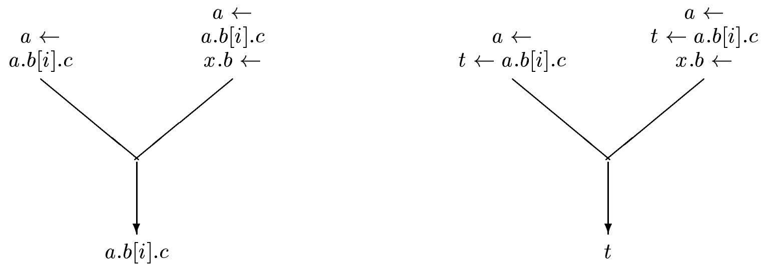


Figure 10.10: After code motion

inal renaming algorithm forces us to keep track of the current SSA numbers of each variable. The purpose of the variable stacks in the original renaming algorithm was to enable us to determine when the value of an available expression is no longer current. If any of the variables have versions different from those in the expression, the expression is no longer current. For real occurrences, the current versions of the variables are those found in the expression. The variable stacks are only necessary when renaming Φ-statement operands.

The "delayed renaming" step consists of two passes. The first pass behaves the same as the original renaming except that it does not use a variable renaming stack. When it encounters a Φ-operand it optimistically assumes that the current variable version is that found in the expression on top of the expression stack. The correct renaming occurs during the second pass which relies on seeing later real occurrences. The first pass constructs a worklist of all the real occurrences that are defined by Φ-statements (such as the real occurrences in block 9). From the versions of the variables found in the block in which a Φ-statement occurs, it determines the versions of the variables at each of the predecessor blocks taking into account any ϕ-statements in the merge block that cause a redefinition of one of the variables. If the Φ-operand versions are different from those assigned in the first pass, the



(a) Before PRE (b) After PRE

Figure 10.11: Access Path PRE

operator's version is invalidated by setting it to $\perp$. If the Φ-operand is defined by a Φ-statement, it is added to the worklist. Consider block 8 in Figure 10.3. The first pass will set the second operand of the Φ-statement to h_3, the h defined by the Φ-statement in block 6. The second pass will correctly set it to $\perp$.

10.1.8 PRE for Access Paths

We now come to the topic of Nate's Thesis: Partial Redundancy Elimination for Access Paths. Recall that an access path is a non-empty sequence of memory references (see section 9.3). In Java, access paths consist of field and array accesses. BLOAT performs partial redundancy elimination on access paths as well as first order expressions. An example of this is shown in Figure 10.11. Access path aliasing may effect the amount of code motion that PRE can perform. So, type-based alias analysis is used to determine whether or not two memory references can refer to the same location.

10.2 Implementation

Now that you've digested the background information, it's time for the implementation. For the most part, the implementation follows the algorithm. A couple of things need to be added to deal with access paths and certain Java characteristics. Here we go.

Partial Redundancy Elimination is implemented by the public `transform` method of the `bloat.trans.SSAPRE` class. The `transform` method first calls `collectOccurrences` to create a worklist of expressions on which to perform SSAPRE and then calls the private `transform` on each expression in the worklist until the worklist is empty. Each `SSAPRE` keeps track of information such as the `FlowGraph` on which SSAPRE is being performed and the `bloat.editor.Editor` containing information about all the classes. It implements the SSAPRE algorithm using a number of methods and several auxiliary classes.

10.2.1 Kills

Several properties of the Java language hinder code motion and, specifically, the partial redundancy elimination of access paths. BLOAT identifies *alias definition points*, places in the method where aliased variables may be modified. Of course, memory references may cause aliasing. However, monitor synchronization points may also result in aliasing. Java's thread model specifies that any change made to a variable in one thread must be available to all threads. Thus, we must assume that calling a method can cause an alias. Additionally, code that may throw an exception cannot be hoisted out of a protected region.

`SSAPRE` represents expressions that hinder code motion with the abstract `Kill` class. Each `Kill` consists of an `Expr` and a integer key. `Kills` are inserted into the worklist along with expressions (see section 10.2.5). `MemRefKill` represents the situation in which two memory references may alias each other. They are also used with synchronized blocks of code.

10.2.2 Modeling Φ-statements

Occurrences of an expression are modeled by the subclasses of the abstract `Def` class. A `Def` consists of an integer version number that uniquely identifies the `Def` instance and is equivalent to the "h" variable. There are two concrete subclasses of `Def`, `RealDef` and `Phi`. A `RealDef` contains an `Expr`.

A `Phi` models a Φ-statement. As expected, the implementation logically follows the algorithm. Each `Phi` is contained in a `Block` and has operands that are themselves `Defs`. Each `Phi` has flags such as "down safe", "can be available", "later", and an array of flags representing the "has real use" information for the operands. It also has a list of leaf expressions.

10.2.3 The Worklist

`SSAPRE` maintains a worklist of expressions on which PRE may be performed. The worklist is implemented as the class `ExprWorklist` which contains a linked list of `ExprInfo` objects. Each `ExprInfo` contains information about an `Expr` in the control flow graph. Each `ExprInfo` is uniquely identified by an `ExprKey` class. `ExprKey` consists of an `Expr` and an integer hashcode computed from the `Expr`'s own hashcode (see `NodeComparator.hashCode`, section 8.3.2) and the hashcode of its type. `ExprKey`'s `listChildren` method returns the children of the `Expr` unless a child is a `StoreExpr` in which case just the left hand side (target) is added to the children list. It also has an `equals` method. Two `ExprInfo`s are equal if their `Expr`s are of the same type, they have the same number of children, and their children have the same type (i.e. are both `StackExpr`s or are both `VarExpr`s and are "phi-related".

`ExprInfo` also keeps track of the number of uses the expression has, the real occurrences of the expression at a given block, the Φ-statements (`Phi`) for the expression at a given block, and a mapping between an expression and its `Def` (see section 10.2.2). A `FinalChecker` is used to determine whether or not the expression references a `final` field. `FinalChecker` is simply a `TreeVisitor` that makes note when a `FieldExpr` or `StaticFieldExpr` references a `final` field. A `bloat.trans.SideEffectChecker` is used to determine whether or not the expression has any side effects. For the purposes of PRE, residency checks, update checks, stores, and possible reassignment are not considered side effects. Additionally, if the expression is a `CheckExpr` that references the stack, we make note of it. `ExprInfo` also maintains several mappings between expressions and flags used to model the "save" and "reload" flags, and to store the "availDefs".

`ExprWorklist` has two methods for constructing the worklist, `addReal` and `addKill`. `addReal` constructs an `ExprKey` for the occurrence (`Expr`) and determines whether or not it has already been placed in the worklist. If not, a new `ExprInfo` is created for the occurrence and it is added to the worklist. In any case, the real occurrence is noted in the `ExprInfo`. `addKill` makes note of a `Kill` at a given block. `SSAPRE` maintains a list of the `Kill`s at each basic block.

10.2.4 Is an Expression First Order?

SSAPRE works with first order (non-nested) expressions. `SSAPRE`'s `isFirstOrder` method determines whether or not an `Expr` is first order. `isFirstOrder` uses a `FirstOrderChecker` object to do the lion's share of the work. `FirstOrderChecker` is a `TreeVisitor` that does not visit any of the `Expr`'s children. `FirstOrderChecker` relies on the private `isLeaf` method to determine whether or not an `Expr` is a leaf (that is, has no children). An `Expr` is a leaf if it is a `LocalExpr` (local variable), a `ConstantExpr`, or a `StoreExpr` whose target is a `LocalExpr`. `FirstOrderChecker` handles various kinds of `Expr`s as follows:

`CheckExpr` If the `CheckExpr` is a leaf or if it checks a `StackExpr` (stack variable), it is first order. We allow `CheckExpr`s of stack variables to be first order because they (the `CheckExpr`s) can be eliminated and replaced with stack variables. They cannot, however, be hoisted.

`ArithExpr` If both the left and right operands are leafs, then the `ArithExpr` is first order.

`ArrayRefExpr` If both the array expression and the index are leafs, then the `ArrayRefExpr` is first order.

`CastExpr` If the expression being cast is a leaf, then the `CastExpr` is first order.

`FieldExpr/StaticFieldExpr` If the field is volatile, we do not consider it to be first order. Otherwise, it is first order.

`InstanceOfExpr/NegExpr` If the expression being operated on is a leaf, then the entire expression is first order.

`ShiftExpr` If both the expression being shifted and the expression for the number of bits to shift are leafs, then the `ShiftExpr` is first order.

10.2.5 Constructing the Worklist

The private `collectOccurrences` method creates a worklist of expressions to which PRE will be applied. Along the way, it keeps track of the maximum value number (see section 8.3) encountered and inserts `Kill` statements into the worklist. The first thing `collectOccurrences` does is to determine which blocks in the control flow graph begin protected regions. This is done by calling the private `beginTry` method. `beginTry` examines each block in the control flow that begins an exception handler (the so-called "catch blocks", see section 5.5.2) and in a somewhat round about way that is not worth describing, computes a list of the predecessors of these blocks.

The bulk of the work done by `collectOccurrences` consists of a `TreeVisitor` that builds the worklist. As each `Node` is visited its value number is examined and the largest value number is remembered. Additionally, each `Expr`'s "key" is set to its number in a pre-order traversal of the control flow graph. The visitor is summarized as follows:

`Block` If the `Block` begins a protected region, then an `ExceptionKill` occurs at the `Block`.

`PhiStmt` After visiting the `PhiStmt`'s children, each operand is examined. If the operand is a `VarExpr` (local variable or stack variable) with a non-null definition, the operand and the variable defined by the `PhiStmt` are "phi-related".

`CatchExpr` An `ExceptionKill` is noted at the block in which the `CatchExpr` occurs.

`MonitorStmt` If the `NO_THREAD` flag is false, then a `MemRefKill` is noted at the `MonitorStmt`'s block. Remember that the values of fields may change at synchronization points.

`CallExpr` A `MemRefKill` is noted at the block in which the `CallExpr` occurs.

`MemRefExpr` Recall that a `MemRefExpr` models field and array references. If the `MemRefExpr` is first order, then its children are visited. This is calculated by the private `isFirstOrder` method.

Once the worklist of `ExprInfo`s has been constructed, the `transform` method is called on each `ExprInfo` in the worklist. `transform` implements the remaining steps of SSAPRE on each expression. If the `ExprInfo` for the expression of interest does not have any uses (`numUses` method), no actions are performed on it.

10.2.6 Inserting Φ-statements

The first step is place the Φ-statements for the expression into the control flow graph. This is performed by the private `placePhis` method. Recall that Φ-statements are placed at the iterated dominance frontier of each occurrence of the expression and in any block in which a ϕ-statement for one of the variables in the expression occurs (see section 10.1.1).

`placePhis` begins by constructing a list of all of the blocks in which expression occurs. This list is used as an input to `FlowGraph`'s `iterated-DomFrontier` to compute the blocks in the expression's iterated dominance frontier. Each block in the control flow graph is visited and a worklist of `PhiStmts` is constructed. Each real occurrence of the expression in question at a given block is examined. If the parent of one of the expression's operands is a `PhiStmt` that is not yet in the list, the `PhiStmt` is added. The blocks that contain the `PhiStmt` are added to the worklist containing the expression's iterated dominance frontier. If any of the arguments of the `PhiStmts` is defined by a `PhiStmt`, the block containing the defining `PhiStmt` is also added to the list. Finally, Φ-statements are inserted into the blocks in the worklist using the `ExprInfo`'s `addPhi` method.

10.3 That's all, folks

Well, boys and girls this is the end. It's now October and it's time for me to start thinking about **my** thesis. I apologize for not completing the implementation of SSAPRE. However, I was having extreme difficulties figuring out the code. Maybe someday someone will be motivated enough to finish this chapter, or better yet, just rewrite the implementation. Anyway, I'll send you off with a couple of examples of SSAPRE. Happy BLOATing!

10.4 Examples of PRE

This section discusses several examples of SSA-based partial redundancy elimination. This first is a simple example of PRE of an expression. Figure 10.12 shows the control flow graph for the below Java code in SSA form after type inferencing has occurred.

```
int f(boolean b, int i, int j) {
  int k, l;
  if(b)
    k = i + j;
  else
    k = 3;
  l = (i + j) * k;
  return(l);
}
```

In this example, the expression `i + j` (`Li2_2 + Li3_3`) is partially redundant in block 15. The control flow graph after SSA-based PRE has been performed is given in Figure 10.13. As was expected, a computation of `Li2_2 + Li3_3` was added to block 12[6] and the occurrence in block 15 was replaced by a variable (`Li6_53`) whose value is computed by the ϕ-statement `Phi(label_12=Li6_47, label_4=Li6_49)`.

The second example shows the partial redundancy elimination of an access path.

[6] I was hoping that the expression would be hoisted completely out of the if-statement, but I guess it couldn't figure that out. Sigh.

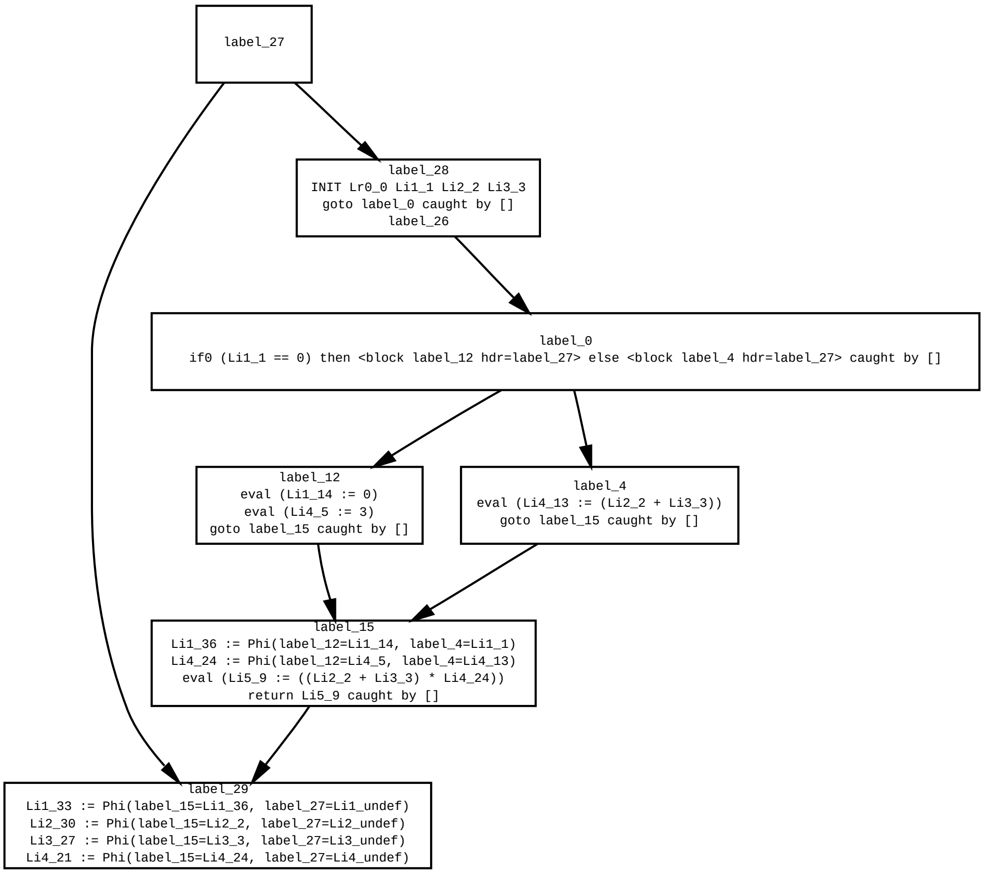


Figure 10.12: Expression PRE Example Before PRE

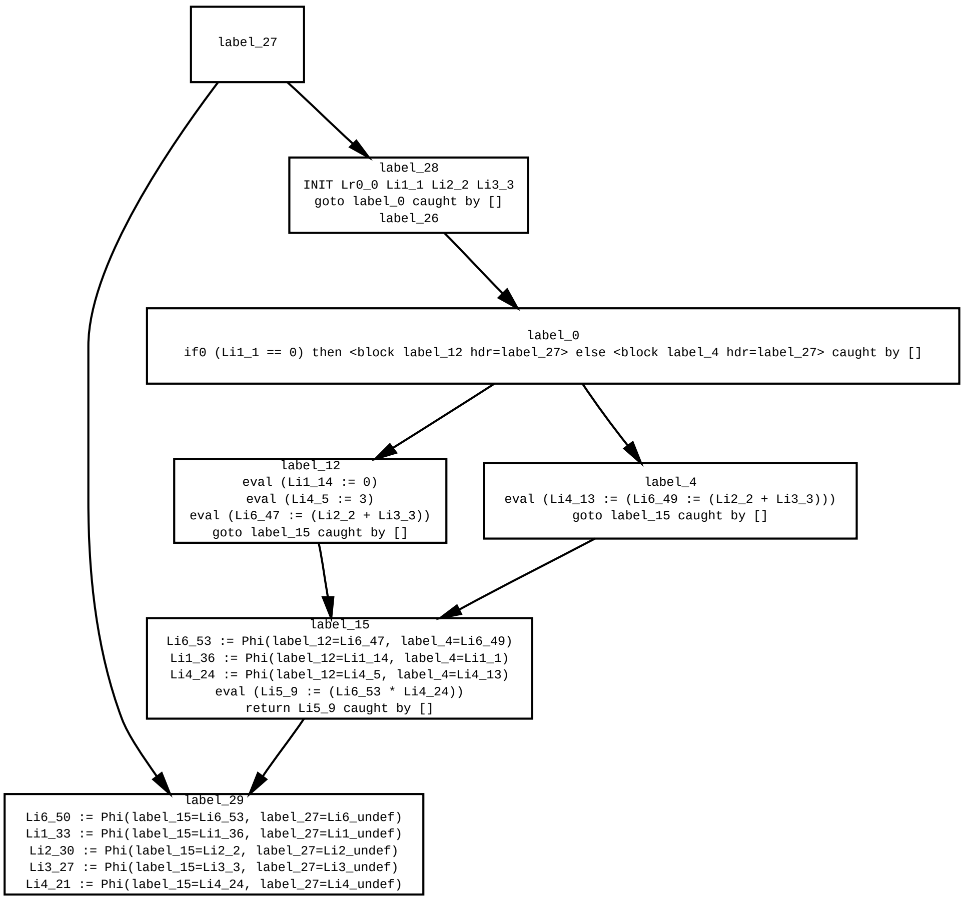


Figure 10.13: Expression PRE Example After PRE

```
public class PREPath {
  int f(boolean c) {
    A a; B b; int y;
    b = new B();
    b.x = 1;
    if(c) {
      a = b;
      y = a.x;
    } else {
      y = 2;
    }
    return(y + b.x);
  }
}
class A {
  int x;
}
class B extends A { }
```

The control flow graph for the method `f` is given in Figure 10.15. The CFG has been converted into SSA form and expression propagation, dead code elimination, type inferencing, value number, and value folding have been performed on it. In this example, the access path `b.x` (`Lr3_8.x`) is redundant in block 31. Note that variable `a` is an alias for variable `b` (`a` was eliminated from the control flow graph in Figure 10.14 during expression propagation).

The control flow graph after PRE has been performed on it is given in Figure 10.15. The partially redundant `Lr3_8.x` in block 31 has been replaced with the variable `Li5_52`. The PRE analysis has recognized that neither `Lr3` nor `Lr3.x` will be modified between the two occurrences and thus creates a new variable, `Li5_52`, to hold the value of `Li3.x`. The occurrences of `Li3.x` in block 17 and 31 are replaced with `Li5_32`.

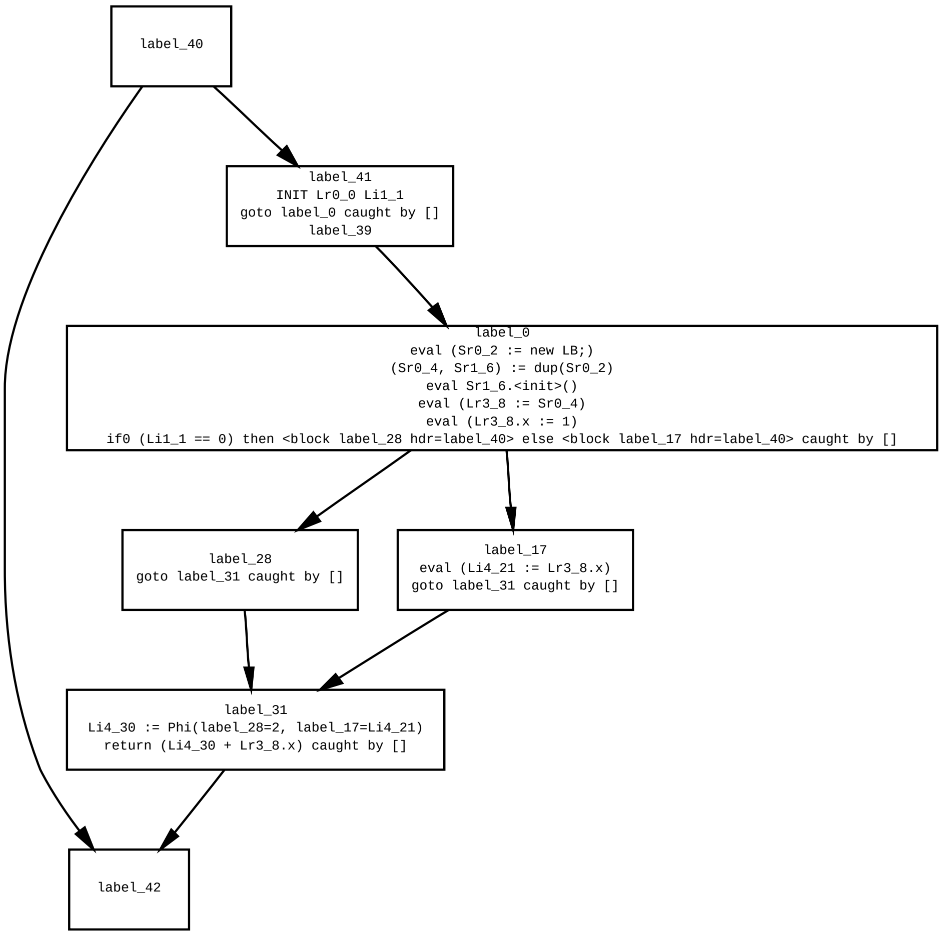


Figure 10.14: Access Path PRE Example Before PRE

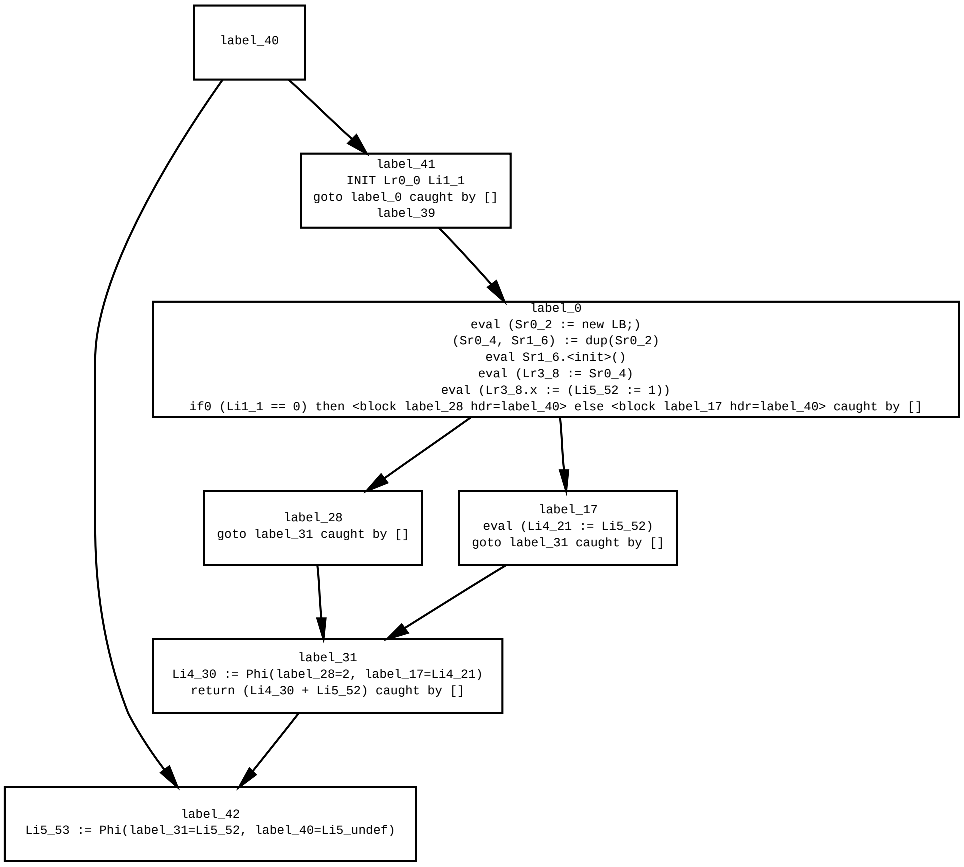


Figure 10.15: Access Path PRE Example After PRE

Bibliography

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software–Practice and Experience*, 1(1), January 1998. Revised July 1997.

[CCK+97] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, volume 32, pages 273–286, May 1997.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kennth Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM Transactions on Programming Languages and Systems*, volume 13, pages 451–490, October 1991.

[CS95] Keith Cooper and Taylor Simpson. Ssc-based value numbering. Technical Report CRPC-TR95636-S, Rice University, October 1995.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. volume 33, pages 106–117, June 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Weslet, 1995.

[Hav97] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Language Systems*, 19(4):557–567, 1997.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

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

[Nys98] Nathaniel John Nystrom. *Bytecode-Level Analysis and Optimization of Java Classes.* PhD thesis, Purdue University, August 1998.

[PM72] Paul W. Purdom and Edward F. Moore. Immediate predominators in a directed graph. *Communications of the ACM*, 15(8):777–778, August 1972.

[PSb94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* Wiley, 1994.

[Sim96] Loren Taylor Simpson. *Value-Driven Redundancy Elimination.* PhD thesis, Rice University, April 1996.

Index

Φ-insertion, 145
Φ-statement, 144
⊥, 148, 153
ϕ-blocks, 63
ϕ-catch statement, 87
ϕ-return Statement, 87
ϕ-statement, 42, **85**, 102, 117, 119, 153
`Filter` class, 130
`Node`'s key, 115
`Tuple` class, 124
`this` pointer, 122, 123, 127, 129

aaload instruction, 44
aastore instruction, 45, 114
`AbstractCollection` class, 22
`AbstractList` class, 22
`AbstractMap` class, 22
`AbstractSet` class, 22
access flags, *see* modifiers
`addCatchPhiOperands` method of `SSA`, 91
`addHandlerEdges` method of `FlowGraph`, 58, 60
`addCall` method of `Tree`, 50
`addCatchPhi` method of `SSAConstructionInfo`, 90
`addClass` method of `ClassHierarchy`, 133
`addInst` method of `Tree`, 43
`addInstruction` method of `MethodEditor`, 105
`addInstruction` method of `Tree`, 43, 59
`addLabel` method of `MethodEditor`, 106
`addPhi` method of `SSAConstructionInfo`, 90
`AddressStoreStmt` class, 73, 104, 107
`AddressStoreStmt` class, **39**, 45
`addRetPhis` method of `SSAConstructionInfo`, 90
`addStore` method of `MemExpr`, 44
`addStore` method of `Tree`, 46, 49
`addType` method of `ClassHierarchy`, 134
alias definition points, 159
`allocReturnAddresses` method of `CodeGenerator`, 104
aload instruction, 44
anewarray instruction, 50
areturn instruction, 41, 107
`ArithExpr` class, **36**, 47, 67, 73, 101, 107, 108, 115, 123, 135, 161
`ArithExpr.CMP` constant, 135
array, 38
array initialization compaction, 113
arraylength instruction, 36, 51, 108
`ArrayLengthExpr` class, **36**, 51, 67, 73, 108, 121, 136
`ArrayList` class, 23, 24, 34

ArrayRefExpr class, **38**, 44, 45, 73, 106, 108, 115, 121, 136, 140, 161
Arrays class, 23
AsbtractSequentialList class, 22
Assign class, 37, 40, 42, **42**
astore instruction, 39, 43, 45, 55, 59, 107, 114
aswizzle instruction, 40, 51, 108
aswizzleRange instruction, 40, 51
aswrange instruction, 108
athrow instruction, 41, 51, 57, 58, 108
Attribute class, 17, **17**, 18
attributes, 17
aupdate instruction, 51, 107

back edge, **64**, 65
baload instruction, 44
basic block, 28, 31, 33, 35, **53**
basic block types, 55
bastore instruction, 45
bipush instruction, 113
bloat.editor package, 25–31
bloat.util package, 23–24
Block class, 34, 35, 40, 41, 43, **54**, 55, 56, 59, 73, 121, 161
buildLoopTree method of FlowGraph, 66
buildTreeForBlock method of FlowGraph, 58, 60
buildBlocks method of FlowGraph, 56
buildFrontier method of DominanceFrontier, 62
buildLoopTree method of FlowGraph, 56
buildTree method of DominatorTree, 62
buildTreeForBlock method of FlowGraph, 43
buildTrees method of FlowGraph, 57, 59

cache, 25
calcFrontier method of DominanceFrontier, 62, 63
CallExpr class, **37**, 162
CallMethodExpr class, **37**, 50, 73, 106, 108, 115, 121, 137
CallStaticExpr class, 37, 50, 73, 115, 121, 137
caload instruction, 44
can be available, 151
canAlias method of TBAA, 140
CastExpr class, **36**, 51, 67, 73, 108, 115, 121, 123, 137, 161
castore instruction, 45
Catch class, **15**, 18, 30, 55
catch Block, 90, 91
catch block, 57, 58, 63, 161
catch body, 57, 58
catch targets, 104
CatchExpr class, **36**, 57, 73, 109, 115, 121, 136, 162
catchTargets method of JumpStmt, 40
cfg constant, 55
checkcast instruction, 51, 108
CheckExpr class, **37**, 120, 122, 136, 160
children method of SSAGraph, 120
ClassFileLoader class, **16**, 17
ClassCastException class, 21
ClassEditor class, 25, **27**, 30, 133
ClassFile class, 16, **16**, 18
ClassFormatException class, 16
ClassHierarchy class, 25, **133**, 140
ClassInfo class, **13**, 16, 27

`ClassInfoLoader` class, **13**, 25
`ClassNotFoundException` class, 16
`clearCode` method of `MethodEditor`, 105
coalescing, 102
`Code` class, **18**
code motion, 154
`CodeGenerator` class, 26, **102**
collect occurrences, 155
`Collection` class, 21, 25, 38, 40
`Collections` class, 23
`collectOccurrences` method of `SSAPRE`, 158
`collectVars` method of `SSA`, 89
commit, 25
`commit` method of `ClassFile`, 17
`CompactArrayInitializer` class, 113
`Comparable` class, 22
`Comparator` class, 22
`compare` method of `Compator`, 22
`compare` method of `IdentityComparator`, 24
`ComponentVisitor` class, 120
`computeIntersections` method of `Liveness`, 99
`CondExpr` class, **38**
conditional jump, 60
conditional statements, 68
congruence of variables, 117
`Constant` class, **14**, 17, 26
constant pool, 14, 26, 28, 113
constant propagation, 125
`ConstantExpr` class, **36**, 43, 44, 47, 73, 107, 108, 122, 125, 137, 160
`ConstantPool` class, 26, 27
`ConstantValue` class, **18**
control flow graph, 23, 42, 48, **53**
copy propagation, 125
`copyBlock` method of `FlowGraph`, 67
`createUndefinedStore` method of `CodeGenerator`, 102
`createStore` method of `CodeGenerator`, 102
critical edge, 68, 87, 144

daload instruction, 44
dastore instruction, 45
dcmpg instruction, 48
dcmpl instruction, 48
ddiv instruction, 47
DEAD, 115
dead code elimination, 115
`DeadCodeElimination` class, **115**
debugging information, 28, 30
`Def` class, 159
`DefExpr` class, **38**, 42, 75, 109, 116
defining expression, 35
`defs` method of `Assign`, 40, 42
delayed renaming, 157
design patterns, 30
  visitor, 30, 34
dload instruction, 44
dneg instruction, 109
dominance frontier, **54**, 62
`DominanceFrontier` class, 55, **62**
dominate, **53**
dominator, 61
dominator tree, 53, 55, 61, 127, 148, 153
`DominatorTree` class, 55, **61**
dot, 76
down safe, 148
dreturn instruction, 107
dstore instruction, 44
dup instruction, 35, 40, 43–45, 107, 108, 113, 114, 130
dup2 instruction, 45, 108

dup2_x1 instruction, 45, 108
dup2_x2 instruction, 45, 108
dup_x1 instruction, 45, 46, 108
dup_x2 instruction, 45, 108

editField method of Editor, 25
Editor class, **25**, 27, 133, 158
editor constant, 36, 39, 50
entry block, 53, *see* source block, 104
Enumeration class, 22
equals method of Compator, 22
equals method of NodeComparator, 122
eval, 73
ExceptionKill class, 161, 162
Exceptions class, **19**
exceptions, 15, 19, 51, 55, 87
exit block, 53, *see* sink block, 104
exit node, 53
Expr class, 34, **35**, 73, 102
expression, 33, 35
expression propagation, 125
expression tree, 33
ExprInfo class, 159
ExprKey class, 159
ExprPropagation class, **125**
ExprStmt class, **39**, 44, 45, 47, 50, 73, 106, 107, 116
ExprWorklist class, 159

faload instruction, 44
fastore instruction, 45
fcmpg instruction, 48
fcmpl instruction, 48
fdiv instruction, 47
Field class, 17, **18**
FieldEditor class, **27**
FieldExpr class, **38**, 49, 67, 74, 106–108, 115, 121, 137, 140, 160, 161
FieldInfo class, **13**, 27
file package, 16–19
fillArray method of CompactArray-Initializer, 113, 114
filter method of Peephole, 130
final, 141
FinalChecker class, 160
finalize, 152
finally, 41, 55
FirstOrderChecker class, 160
fload instruction, 44
FlowGraph class, 34, 43, 54, **56**, 74, 89, 99, 106, 120, 124, 125, 158
fneg instruction, 109
freturn instruction, 107
fstore instruction, 44

GenericAttribute class, **17**
genIfCmpStmt method of CodeGenerator, 106
genIfZeroStmt method of Code-Generator, 106
genPostponed method of CodeGenerator, 105
getChars method of String, 114
getfield instruction, 49, 108
getstatic instruction, 109
goto instruction, 48, 60, 65, 106, 107
GotoStmt class, **41**, 48, 57, 64, 71, 73, 74, 103, 106, 115
Graph class, **23**, 56, 99, 133
GraphNode class, 23, **23**, 54

Handler class, **55**, 56, 58, 65
has real use, 149
hashCode method of NodeComparator, 122, 159

hashCode method of Object, 24
HashMap class, 23
Havlak, Paul, 64
hierarchy method of Editor, 25
hypothetical temporary, 144

IdentityComparator class, **24**
iaload instruction, 44
iand instruction, 47
iastore instruction, 45, 113
iconst instruction, 113
identityHashCode method of System, 24
if instruction, 28
if_cmpeq instruction, 130
IfCmpStmt class, **41**, 48, 69, 106, 122, 123
ifnonnull instruction, 48
ifnull instruction, 48
IfStmt class, **41**, 64, 71, 115
IfZeroStmt class, **42**, 48, 69, 73, 74, 106, 123
IGNode class, 99
iinc instruction, 29, 47, 107
iload instruction, 44, 100
immediate dominator, **53**, 62
ImmutableIterator class, **24**
IncOperand class, **29**, 47
ineg instruction, 109
inheritance, 133
init block, 56, 58, 59
initialize method of FlowGraph, 61
initLocals method of Tree, 58
InitStmt class, **40**, 43, 58, 74, 102, 122, 129, 135
insert flag, 152, 153
insertConditionalStores method of FlowGraph, 69
insertEdgesToSink method of DominatorTree, 61
insertCode method of SSA, 93
instanceof instruction, 38, 52, 108
InstanceOfExpr class, **38**, 52, 74, 108, 136, 161
Instruction class, **28**, 30, 43, 44, 57, 59, 105, 129
instructions, 28
InstructionVisitor class, 29, **29**, 35, 43, 55
interface, 133
interference graph, 99
intersectType method of ClassHierarchy, 134
invokeinterface instruction, 50, 108
invokespecial instruction, 50, 108
invokestatic instruction, 37, 50
invokevirtual instruction, 50, 108
ior instruction, 47
ireturn instruction, 107
IRREDUCIBLE constant, 66
isDef method of Assign, 43
isFirstOrder method of SSAPRE, 160
ishl instruction, 47
ishr instruction, 47
isReturnAddress method of LocalExpr, 39
istore instruction, 44
iterated dominance frontier, 63, 85, 145, 162
iteratedDomFrontier method of FlowGraph, 63, 162
Iterator class, 21, 22, 24
iushr instruction, 47
ixor instruction, 47

Jar file, 16
java.class.path, 16

java.io.DataInputStream class, 16, 18
java.io.File class, 16, 17
java.io.PrintStream class, 76
java.io.PrintWriter class, 34, 73
java.util package, 21–23
JDK1.2, 21, 22
jsr instruction, 41, 43, 48, 55, 57, 59, 60, 66, 107
JsrStmt class, **41**, 48, 64, 71, 74, 103, 107, 115
JumpStmt class, **40**, 58, 64, 65, 104, 116

key, 161
Kill class, 159, 161
kills, 159

Label class, **28**, 29, 30, 43, 54, 57, 59, 60, 129
LabelStmt class, 35, **40**, 60, 67, 74, 103, 106, 116
laload instruction, 44
land instruction, 47
lastore instruction, 45
later flag, 151
ldc instruction, 44, 108, 130
ldc_w instruction, 44
LeafExpr class, 39, **43**, 67, 69
LineNumberDebugInfo class, **15**, 19
LineNumberTable class, **19**
LinkedList class, 23
List class, 21, 22
ListIterator class, 21, 22
live, 115
live out, 100
Liveness class, **99**
liveness analysis, 99
lload instruction, 44
lneg instruction, 109
loadClass method of ClassFileLoader, 16
loadClassFromFile method of ClassFileLoader, 16
loadClassFromStream method of ClassFileLoader, 16
local variables, 28, 35, 40, 99
LocalDebugInfo class, **15**, 19
LocalExpr class, 35, 39, **39**, 43, 44, 46, 47, 74, 91, 99, 107, 109, 115, 121, 122, 126, 135, 160
LocalVariable class, **28**, 29, 30, 55
LocalVariableTable class, **19**
lookupswitch instruction, 29
loop, **54**, 56
  depth, 66
  header, 64
  irreducible, 67
  level, 66, 67
  reducible, 54, 64, 65
loop header, **54**, 57, 64
loop inversion, 67
loop peeling, 67
loop splitting, 64
loop tree, 56, 66, 67
LoopNode class, 66
lor instruction, 47
lreturn instruction, 107
lshl instruction, 47
lshr instruction, 47
lstore instruction, 44
lushr instruction, 47
lxor instruction, 47

magic number, 17
major number, 17
manip method of Tree, 46
Map class, 22, 23, 42

markLive method of DeadCodeElimination, 116
maxLocals, 18
maxStack, 18
MemberRef class, 26, **27**, 37, 38, 49, 50, 108
MemExpr class, 37, **38**, 44
MemRef class, 124
MemRefExpr class, **38**, 140, 162
MemRefKill class, 159, 162
Method class, 17, **18**
MethodEditor class, **30**, 56, 57, 59, 105, 106, 113, 129
MethodInfo class, **13**, 30
methodParams method of FlowGraph, 58
methods, 18
minor number, 17
Modifiers class, **14**
modifiers, 17, 27
MonitorStmt class, 74
monitorenter instruction, 40, 52, 106
monitorexit instruction, 40, 52, 106
MonitorStmt class, **40**, 52, 106, 115, 121, 162
multianewarray instruction, 29, 36, 50
MultiArrayOperand class, **29**, 50, 109
multinewarray instruction, 109
munchCode method of MethodEditor, 28, 30

NameAndType class, 26, **27**
NaN, 48
Nystrom, Nate, 21
negate method of IfStmt, 41
NegExpr class, **36**, 47, 74, 109, 123, 136, 161
new instruction, 36, 50, 109
newarray instruction, 36, 50, 109, 114
NewArrayExpr class, **36**, 50, 74, 109, 115, 121, 122, 129, 137
newBlock method of FlowGraph, 57
NewExpr class, **36**, 50, 74, 109, 115, 121, 122, 129, 137
NewMultiArrayExpr class, **36**, 74, 109, 115, 121, 122, 129, 137
next method of Iterator, 22
NO_THREAD, 162
Node class, **33**
NodeComparator class, 22, **122**
non-local variables, 85, 90
NON_HEADER constant, 66

opcName field of class Opcode, 28
Opcode class, **28**
opcodes, *see* instructions
opcSize field of class Opcode, 28
opcXMap field of class Opcode, 28
operand stack, 34, 44, 46, 59
OperandStack class, **34**, 35, 44
Optimistic Table, 124
optimistic table, 119

partial redundancy elimination, 143, 158
path, 56, 57, 59, 90
peeking, 34, 51
PEEL_ALL_LOOPS constant, 67
PEEL_LOOP_LEVEL constant, 67
PEEL_NO_LOOPS constant, 67
peelLoops method of FlowGraph, 67
Peephole class, **129**
peephole optimizations, 129

persistent checks, 127
persistent store, 51
PersistentCheckElimination class, **127**
Phi class, 159
PhiCatchStmt class, **42**, 74, 90, 91, 99–102, 125, 126
PhiJoinStmt class, **42**, 74, 90–92, 99, 101, 103, 122
PhiReturnStmt class, 90–92
PhiStmt class, **42**, 43, 72, 89, 91, 93, 101, 115, 120, 126, 136, 161, 162
PJama, 40
placePhiFunctions method of SSA, 90
placePhis method of SSAPRE, 162
pop instruction, 45, 106, 130
pop2 instruction, 45, 106
post-order traversal, 23
postdominance frontier, 54
postdominate, **53**
postdominator tree, 53, 55
Postscript, 76
pre-live, 115
pre-order traversal, 23
predacessors, 23
prependStmt method of Tree, 93
PRESERVE_DEBUG constant, 30
previous method of Iterator, 22
print method of FlowGraph, 76
printGraph method of FlowGraph, 76
PrintVisitor class, **34**, **73**, 76
prop method of TypeInference, 135
propagate method of ExprPropagation, 126
propExpr method of ExprPropagation, 126
protected block, 58
protected region, 91
Purdum-Moore, 61
putfield instruction, 49, 108, 114
putstatic instruction, 109, 114

rc instruction, 37, 105, 106
RCExpr class, **37**, 75, 105, 106, 115, 121, 122, 127, 129
RealDef class, 159
red-and-black tree, 22, 23
REDUCIBLE constant, 66
reducible backedge, 64
reducible loop, 54, 64, 65
reference count, 25
reflect constant, 36
reflect package, 13–16
RegisterAllocator class, **101**
reload flag, 153, 154
remove method of ImmutableIterator, 24
removeEmptyBlocks method of CodeGenerator, 103
removeUnreachable method of Peephole, 130
rename method of SSA, 91
renaming (SSAPRE), 146
replaceCatchPhis method of CodeGenerator, 102
replaceJoinPhis method of CodeGenerator, 102
replacePhis method of CodeGenerator, 102
ReplaceVisitor class, 33
ReplaceTarget class, **64**, 65
ReplaceVisitor class, **34**
residency check, 37, 127
ResizableArrayList class, **24**, 26
ret instruction, 41, 43, 49, 55, 59, 64, 107

RetStmt class, **41**, 49, 64, 71, 73, 75, 103, 107, 115
return instruction, 28, 41, 107
return address, 36
ReturnAddressExpr class, **36**, 48, 75, 109, 136
ReturnExprStmt class, **41**, 49, 75, 107, 115
ReturnStmt class, **41**, 107, 115
reverse roots, 23
roots, 23

saload instruction, 44
sastore instruction, 45
save flag, 153, 154
saveLabels method of FlowGraph, 60
saveStack method of Tree, 35, 43
SCC, 118, 121
SCStmt class, **40**, 51, 75, 108, 115
search method of PersistentCheck-Elimination, 127
search method of SSA, 91
Semi-Pruned SSA Form, 85
Set class, 21, 55
setBlockTypes method of FlowGraph, 66
setClassPath method of Class-FileLoader, 16
setDef method of Expr, 91
setDomParent method of Block, 62
setOutputDir method of Class-FileLoader, 16
ShiftExpr class, 75
ShiftExpr class, **37**, 47, 109, 123, 135, 161
short, 101
side effects, 121, 129
SideEffectsChecker class, **121**
SideEffectChecker class, 160
simplify method of ValueNumbering, 124
simplifyControlFlow method of CodeGenerator, 103
sink block, 56, 58, 60
size method of ResizableArrayList, 24
source block, 56, 58, 66
splitIrreducibleLoops method of FlowGraph, 65
splitPhiBlocks method of FlowGraph, 64
splitEdge method of FlowGraph, 65, 68
SRStmt class, **40**, 51, 75, 108, 115
SSA class, **89**
SSA form, 42, 99
SSA graph, 117, 151, 152, 154
SSAConstructionInfo class, 89, 93
SSAGraph class, **120**, 124, 135
SSAPRE class, 158
stack, 45
stack height, 30, 106
stack machine, 34
stack variable, 35, 39, 103
StackExpr class, 35, **39**, 40, 45, 57, 75, 91, 107, 109, 115, 116, 121
stackHeight method of Type, 26, 34
StackManipStmt class, **40**, 43, 45, 75, 108, 115, 120, 121, 136
start method of TypeInference, 135
statement, 33
statements, 39
static single assignment form, *see* SSA Form

StaticFieldExpr class, 38, 75, 109, 121, 137, 140, 160, 161
Stmt class, 35, **39**, 75
StoreExpr class, 35, **37**, 43, 44, 47, 57, 69, 75, 101, 102, 106, 107, 115, 116, 120–122, 126, 136, 159, 160
strictly dominates, **53**
String class, 26
Strongly Connected Components, *see* SCC
Subroutine class, 39, 41, 43, 48, 55, **55**, 57, 59, 90, 107
subroutine, 41, 48, 55, 63, 87
  entry, 56
  path, 56, 57, 59
successors, 23
supdate instruction, 51, 107
superclass, 17, 27
swap instruction, 40, 45, 108
Switch class, **29**, 49, 60
switch, 29, 30, 41, 49, 60
switch instruction, 108
SwitchStmt class, **41**, 49, 64, 70, 71, 75, 108, 115, 123
swizzle check, 40

tableswitch instruction, 28, 29, 49
Tarjan's Algorithm, 119
TBAA class, **140**
ThrowStmt class, **41**, 51, 75, 108, 115
trace, 54, 56, 99, 129
trans constant, 33, 114
transform method of CompactArray-Initializer, 113
transform method of DeadCode-Elimination, 115
transform method of ExprPropagation, 125
transform method of Peephole, 129
transform method of SSAPRE, 158
transform method of SSA, 89
transform method of TypeInference, 135
transform method of ValueFolder, 125
transform method of ValueNumbering, 124
Tree class, **35**, 43, 54, 55, 57–59
tree constant, 37, 45
TreeMap class, 23
TreeVisitor class, **34**, 64, 71, 73, 99, 103, 105, 116, 121, 122, 135, 160, 161
try-catch blocks, 30, 40
TryCatch class, **30**, 31, 55, 57, 58, 106
Type class, 26, **26**, 27, 28, 30, 34, 35, 50, 55, 109, 114, 133
type descriptor, 26, 30, 33
type-based alias analysis, 133
TypeInference class, **134**
TypeInferenceVisitor class, 135
TypeNode class, 133
typeToSet method of TypeInference, 135

uc instruction, 37, 105
UCExpr class, 37, 51, 75, 107, 115, 121, 123, 127, 129
UnionFind class, **24**, 66
unionType method of ClassHierarchy, 134
update check, 37, 127
use-def, 85
USE_STACK field of class Tree, 35
UTF8 String, 14, 114

valid table, 119, 124
`valnum` method of `ValueNumbering`, 124
value graph, *see* SSA Graph
value number, 33, 71, 72, 127, 161
value numbering, 117, 124
`ValueFolder` class, **122**, 125
`ValueFolding` class, **125**
`ValueNumbering` class, 120, 124
`VarExpr` class, **39**, 42, 89, 102, 116, 120, 136, 161
variable congruence, 117
`Vector` class, 23
`VerifyCFG` class, **71**
`verifyTargets` method of `VerifyCFG`, 71
version number, 38
`visit` method of `Instruction`, 29
`visitChildren` method of `Node`, 120
`visitComponents` method of `SSAGraph`, 120
`visitConstantExpr` method of `TreeVisitor`, 34
`visitExpr` method of `TreeVisitor`, 34
visitor, 30
volatile, 141, 161

wide instruction, 34
width, 102
will be available, 151, 153

`ZeroCheckExpr` class, 75
`ZeroCheckExpr` class, **37**, 47, 49, 109, 115, 121, 123
zip file, 16