University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Recursion Removal in Java

Scientific Adviser:
Prof. dr. ing. Lorina Negreanu

Author:
Andrei Silviu Dragnea

Bucharest, 2017

Contents

List of Figures

Notations and Abbreviations

API – Application Programming Interface
AST – Abstract Syntax Tree
CFG – Control Flow Graph
IDE – Integrated Development Environment
IR – Intermidiate Representation
JLS – Java Language Specification
JVMS – Java Virtual Machine Specification
PSI – Program Structure Interface
SDK – Software Development Kit

Chapter 1

Introduction

The purpose of this thesis is to present an algorithm for transforming a recursive method in the Java programming language into a method which simulates the recursion using an explicit stack in the user program. This transformation preserves the semantics of the original code, but since the main purpose of this refactoring[4] is to avoid `StackOverflowErrors`, the performance of the transformed method may be worse than the performance of the original one. This refactoring is a temporary solution for the bigger problem of improving the original algorithm.

The algorithm of the refactoring is split into multiple steps. The main idea of the algorithm is to construct a reduced version of the control flow graph of the method, which highlights the points of return from recursion in the method body. These points need to be highlighted because control flow has to jump there after returning from a recursive call and continue executing the following statements. Because the Java programming language does not support the `goto` statement, jumping into the method body is accomplished by enclosing the basic blocks of the control flow graph inside a `switch` statement, each block having its corresponding `case` label. The `switch` statement itself has to be placed inside a `while` statement, which loops until the simulated stack becomes empty.

The reduced control flow graph contains basic blocks corresponding only to statements which contain at least one recursive call. The statements which do not contain recursive calls do not get transformed, thus preserving the code of the original method as much as possible.

The first few steps of the algorithm are preparatory ones. Their purpose is to remove some ambiguities which may appear in later steps of the algorithm. The first preparatory step consists of renaming some of the local variables in the method body such that no name clashes are generated later on. Then, `foreach` loops containing at least one recursive call are converted to explicit `for` loops, because otherwise the control flow graph could not be built. Single statements which may appear as the branches of `if` statements or as the bodies of `for`, `while` and `do-while` statements are converted to block statements containing the original statements, because some steps of the algorithm may expand the original statements to multiple ones.

Since a recursive call may be part of a bigger expression and the first statement to be executed after returning from the recursive call has to be separated from the call itself, all the recursive calls get extracted to their own statements (if they are not already part of their own statement), by declaring a local variable initialized with the value of the recursive call and replacing the original call with a reference to this variable. Recursive calls of methods with `void` return type are already separated from other statements.

A new static nested class containing fields corresponding to the local variables of the method

is declared inside the class containing the recursive method. Instances of this class are pushed to and popped from a stack object which simulates the call stack of the original method. For each recursive call, including the first call of the recursive method by another method, there is a corresponding `push` call to the stack. For each implicit or explicit return from a recursive call, there is a corresponding `pop` call from the stack.

Before further steps of the algorithm can take place, the body of the original method is incorporated inside the statements which will simulate the call stack, in order to ensure the consistent state of the code between further transformations. This transformation is necessary in order to be able to make use of the framework of the IDE inside which this refactoring is implemented, which relies on the consistent state of the code to work correctly.

Each reference to a local variable (including formal parameters of the method) is then replaced with an access to the corresponding field of the `frame` object at the top of the simulated stack. Local variable declarations having initializers from the original method body are also replaced by assignments to the corrresponding fields of the `frame` object.

The control flow graph of the method is then generated using a visitor which performs specific tasks for each type of node in the abstract syntax tree representing the body of the method. The visitor takes special care of the statements affecting control flow, including `if` statements, looping constructs (`for`, `while`, `do-while`, `switch` statements) and also `break`, `continue` and `return` statements.

Before replacing the original body of the method with a list of `case` labels, one for each basic block in the control flow graph, the unreachable blocks which may have been previously generated are removed. The trivial blocks containing only jumps to other blocks are also replaced with jumps from the previous blocks directly to the blocks to which the trivial blocks jump. The blocks with only one predecessor which are not generated because of a previous recursive call (but are generated when visiting statements affecting control flow) can also be recursively inlined inside the predecessor blocks, without modifying the semantics of the code and thus reducing the number of the generated basic blocks to the bare mininum and improving the readability of the tranformed code.

The last step of the algorithm is replacing the `return` statements with `pop` calls from the simulated stack.

1.1 Project Description

The refactoring is implemented as a plugin for the Intellij IDEA IDE (Integrated Development Environment). There is an open source community edition of the IDE available[1]. Intellij IDEA features a framework for static analysis of Java code with many existing code inspections[2] which detect compiler and potential runtime errors, but also potential code inefficiencies. The plugin introduces a new code inspection in the *Performance Issues* group of inspections for the Java programming language. The default severity of the inspection is *Warning*. The *Remove Recursion* inspection can be seen in Figure 1.1.

The Intellij Platform SDK[1] (Software Development Kit) presents the API (Application Programming Interface) for manipulating the source code in a file. A PSI (Program Structure Interface) File[3] is the root of a structure representing the contents of a file as a hierarchy of PSI elements. A PSI element[4] can have child PSI elements. The `PsiElement` class is the common

[1] https://github.com/JetBrains/intellij-community
[2] https://www.jetbrains.com/help/idea/code-inspection.html
[3] http://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_files.html
[4] http://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_elements.html

base class for all PSI elements. There is a `PsiElement` subclass for each element in the Java programming language.

In Intellij IDEA, new Java code inspections are created by extending the abstract class `com.siyeh.ig.BaseInspection`[1]. Each code inspection needs to override the abstract method `buildVisitor()` in `com.siyeh.ig.BaseInspection` to provide a custom instance of `com.siyeh.ig.BaseInspectionVisitor`[2], which visits each element of a PSI File and detects problems specific to that inspection. Depending on the inspection severity, the problem is highlighted in the editor accordingly. In the case of the recursion removal inspection, the visitor verifies each method call expression to see if it resolves to the containing method and if the qualifier is either absent or `this`. If this is the case, then the visitor registers an error on the method call expression and it appears highlighted in the editor as in Figure 1.2.

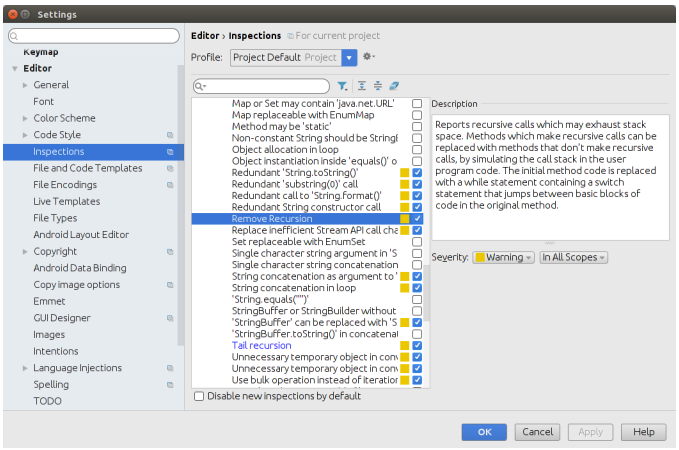


Figure 1.1: Remove Recursion inspection

```
50  private static void dfsCTC(Graph g, Node node) {
51    node.setIndex(g.time);
52    node.setLowLink(g.time);
53    g.time += 1;
54    g.stack.push(node);
55    node.setInStack(true);
56
57    for (Node n : g.getNeighbours(node)) {
58      if (!n.wasVisited()) {
59        dfsCTC(g, n);
60        node.setLowLink(Math.min(node.getLowLink(), n.getLowLink()));
61      } else if (n.isInStack()) {
62        node.setLowLink(Math.min(node.getLowLink(), n.getIndex()));
63      }
64    }
65
66    if (node.getLowLink() == node.getIndex()) {
67      final List<Node> ctc = new ArrayList<>();
68      Node n;
69      do {
70        n = g.stack.pop();
71        n.setInStack(false);
72        ctc.add(n);
73      } while (!n.equals(node));
74      g.ctc.add(ctc);
75    }
76  }
```

Figure 1.2: Recursion Highlighted as a Warning

[1] https://github.com/JetBrains/intellij-community/blob/master/plugins/InspectionGadgets/InspectionGadgetsAnalysis/src/com/siyeh/ig/BaseInspection.java

[2] https://github.com/JetBrains/intellij-community/blob/master/plugins/InspectionGadgets/InspectionGadgetsAnalysis/src/com/siyeh/ig/BaseInspectionVisitor.java

Chapter 2

Prerequisites

2.1 Defining a recursive method in Java

The JLS (Java Language Specification)[2] states the following:

> Resolving a method name at compile time is more complicated than resolving a field name because of the possibility of method overloading. Invoking a method at run time is also more complicated than accessing a field because of the possibility of instance method overriding.

Given the complexity of the method overloading and overriding mechanisms, the definition of a recursive method in this paper is restricted in such a way in order to avoid these mechanisms.

A *recursive method* is defined as a method whose body contains one or more method call expressions for which the static method in Figure 2.1 returns `true`.

```
public static boolean isRecursive(@NotNull final PsiMethodCallExpression expression, @NotNull final PsiMethod method) {
  final PsiReferenceExpression methodExpression = expression.getMethodExpression();
  if (!method.getName().equals(methodExpression.getReferenceName())) {
    return false;
  }
  final PsiMethod calledMethod = expression.resolveMethod();
  if (!method.equals(calledMethod)) {
    return false;
  }
  if (method.hasModifierProperty(PsiModifier.STATIC) || method.hasModifierProperty(PsiModifier.PRIVATE)) {
    return true;
  }
  final PsiExpression qualifier = ParenthesesUtils.stripParentheses(methodExpression.getQualifierExpression());
  return qualifier == null || qualifier instanceof PsiThisExpression;
}
```

Figure 2.1: Test for a recursive method call expression

For performance reasons, Intellij IDEA code base contains many fail-fast methods. This method is also a fail-fast one. The first check verifies if the name of the method in the method call expression matches the name of the expected method. Only after passing this simple check does the test verify that the method to which the call expression resolves is indeed the expected method. Resolving the method is an expensive operation compared with a string equality test.

The `PsiMethod` instance returned by the call to `resolveMethod()` represents the *compile-time declaration*[3] for the method invocation `expression`. In order to guarantee that the method which gets called at runtime is the same as the method corresponding to the compile-time declaration, further checks are needed. JLS specifies that[4]:

[2] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.12
[3] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.12.3
[4] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.12.3

> If the compile-time declaration has the static modifier, then the invocation mode is static. Otherwise, if the compile-time declaration has the private modifier, then the invocation mode is nonvirtual.

It also specifies that[1]:

> The strategy for method lookup depends on the invocation mode. If the invocation mode is static, no target reference is needed and overriding is not allowed. Method m of class T is the one to be invoked. If the invocation mode is nonvirtual, overriding is not allowed. Method m of class T is the one to be invoked.

What this means is that in case of `static` and `private` methods, the method invoked at runtime is the method corresponding to the compile-time declaration. The last check of the test method corresponds to the case when the method is an non-`private` instance method (not `static`). In order to avoid the overriding mechanism, the only methods which this refactoring will consider are those which contain unqualified (or qualified with `this`) method call expressions. This means that the *target reference* of the method remains unmodified.

2.2 Fields of the frame class

When calling another method, the caller creates a new stack frame in which the callee will execute[2]. This is done in order to restore the state of the local variables in the caller when the callee returns. The JVM also stores all the parameters of a method as local variables[3]. The length of the local variables array is determined at compile-time and the storage for the variables lives until the method returns.

Given the fact that the output of the refactoring is still Java code and not bytecode, the restrictions on what variables need to be saved in the frame object are not so tight. However, the scope of the variables needs to be taken into account with respect to the recursive calls in order to decide which variables need to be saved.

In particular, the only variables (formal parameters and local variables) that need to be saved in the current frame are those whose scope contains at least one recursive call. Otherwise, the variables are declared, initialized and their value is used only before or after the recursive call, so they do not need to be saved.

JLS specifies that[4]:

> The scope of a formal parameter of a method, constructor, or lambda expression is the entire body of the method, constructor, or lambda expression.
>
> The scope of a local variable declared in the *FormalParameter* part of an enhanced `for` statement is the contained *Statement*.

Because the formal parameters of a recursive method are always in scope in the method body, they always get saved in the current frame object. However, if the formal parameter is declared in the header of an enhanced `for` statement, it will have a corresponding field in the frame object only if the body of the statement contains at least one recursive call.

JLS also specifies that[5]:

[1] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.12.4.4
[2] http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.6
[3] http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.6.1
[4] https://docs.oracle.com/javase/specs/jls/se8/html/jls-6.html#jls-6.3
[5] https://docs.oracle.com/javase/specs/jls/se8/html/jls-6.html#jls-6.3

> The scope of a local variable declaration in a block is the rest of the block in which the declaration appears, starting with its own initializer and including any further declarators to the right in the local variable declaration statement.
>
> The scope of a local variable declared in the *ForInit* part of a basic `for` statement includes all of the following:
>
> - Its own initializer
> - Any further declarators to the right in the *ForInit* part of the `for` statement
> - The *Expression* and *ForUpdate* parts of the `for` statement
> - The contained *Statement*

A local variable declared in a block will have a corresponding field in the frame object only if its scope contains at least a recursive call. Intuitively speaking, this is the case when the variable is declared in the initialization part of a `for` statement containing a recursive call, usually in its body, or when the variable is declared in a block and after the declaration there is a recursive call in one of the remaining statements of the block.

Chapter 3

The algorithm

This chapter presents all the passes which are applied to the code of the method until the final result is achieved. Each pass is represented by a visitor which usually first collects information about the change that has to be made and then alters the code.

3.1 Replacing single statements with block statements

Given the fact that Java allows some statements to appear outside of a block, they may cause problems in later steps of the algorithm when processing the statements of a block. For example, if a new statement needs to be added before or after another statement which is not part of a block, the initial statement needs to be first enclosed in a block, in which the other statement will be added at the right place. The places where this pass is applied are the branches of `if`[2] statements and the bodies of `for`[3], `foreach`[4], `while`[5] and `do-while`[6] statements which contain single statements. If the single statement is an instance of `PsiEmptyStatement`[7], it gets replaced with an empty block instead of a block containing an empty statement, because this would be redundant.

A special case is the `switch`[8] statement. The pass replaces the statements after each `switch` label (both `case` and `default` labels) up to the next `switch` label (no next one when the label is the last) with a block containing these statements. This transformation does not alter the semantics of the `switch` statement.

This pass also transforms statements which may not contain recursive calls, because there are elements which the refactoring alters in order to remove recursion and they are not only method calls (for example, `return` statements). An example is provided in Figure 3.1.

3.2 Renaming local variables to unique names

This code transformation is needed in order to avoid name clashes later on, when the static nested class which holds the variables potentially used after returning from a recursive call

[2] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.9
[3] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.14.1
[4] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.14.2
[5] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.12
[6] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.13
[7] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.6
[8] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.11

```java
static int fib(int n) {
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else return fib( n: n - 1) + fib( n: n - 2);
}
```

(a) Before

```java
static int fib(int n) {
  if (n == 0) {
    return 0;
  }
  else {
    if (n == 1) {
      return 1;
    } else {
      return fib( n: n - 1) + fib( n: n - 2);
    }
  }
}
```

(b) After

Figure 3.1: Replacing single statements with block statements

(both formal parameters and local variables) is generated. The fields of this class simulate the stack frame of the method.

The JVMS[7] (Java Virtual Machine Specification) states the following[1]:

> The Java Virtual Machine uses local variables to pass parameters on method invocation. On class method invocation, any parameters are passed in consecutive local variables starting from local variable 0. On instance method invocation, local variable 0 is always used to pass a reference to the object on which the instance method is being invoked (`this` in the Java programming language). Any parameters are subsequently passed in consecutive local variables starting from local variable 1.

In the context of this refactoring, the recursive call is defined only as a method call expression whose qualifier is either absent or `this`, so the target reference of the instance method invocation never changes. This is why it is not a field of the frame class. In the case of class method invocations, there is no target reference (`this`).

JLS specifies that[2]:

> It is a compile-time error if the name of a formal parameter is used to declare a new variable within the body of the method, constructor, or lambda expression, unless the new variable is declared within a class declaration contained by the method, constructor, or lambda expression.
>
> It is a compile-time error if the name of a local variable v is used to declare a new variable within the scope of v, unless the new variable is declared within a class whose declaration is within the scope of v.

This means that all the names of formal parameters and local variables whose scopes overlap are unique in the method body. In other words, a compile-time error occurs for the program in Figure 3.2, but not for the program in Figure 3.3.

In the second example, it also means that the storage for the first `i` variable can be reused as the storage for the second `i` variable, thus reducing the number of fields in the generated frame class, as at any moment during the execution of the method, only one variable (formal parameter or local variable) of a certain name is in scope, meaning that its value can be used in the code.

There are still problems if there is another variable in the method which shares the same name with other variable, but whose type is different. If this other variable is not renamed, when

[1] http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.6.1
[2] https://docs.oracle.com/javase/specs/jls/se8/html/jls-6.html#jls-6.4

```java
public class Test1 {
  public static void main(String[] args) {
    int i;
    for (int i = 0; i < 10; i++)
      System.out.println(i);
  }
}
```

Figure 3.2: Attempted shadowing of a local variable

```java
class Test3 {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++)
            System.out.print(i + " ");
        for (int i = 10; i > 0; i--)
            System.out.print(i + " ");
        System.out.println();
    }
}
```

Figure 3.3: Two non-overlapping scopes

generating the frame class, a compile-time error will occur because JLS specifies that[1]:

> It is a compile-time error for the body of a class declaration to declare two fields with the same name.

So all the relevant variables in the method (formal parameters and local variables) are collected as fields of a static nested frame class, which is used by the recursive method to save the state of the method call when recursing, thus simulating the call stack. The rule is that for all the variables with the same name and type, there is only one corresponding field in the frame class, but if there are variables with the same name, but different type, each variable in a type group has a different name and a different corresponding frame class field.

In other words, the first pass ensures that all the variables in a method which have the same name also have the same type.

3.3 Replacing `foreach` loops with `for` loops

In Java, the `for` statement[2] has two forms: the basic one and the enhanced one. Since the meaning of the enhanced `for` statement is given by translation into a basic `for` statement[3], the enhanced form is merely syntactic sugar for the basic one.

This pass first collects all the enhanced `for` statements in the method body which contain at least one recursive call in pre-order fashion by using a visitor processing children nodes before the current node. Then it replaces these statements by their equivalent basic `for` statements. This conversion is necessary because control flow needs to be explicit for later stages in the refactoring process. The *Condition* and *Update* parts of a `for` statement in particular need to be explicit because they get executed after returning from a recursive call in the body of the `for` statement.

There are two cases for this conversion: the first when the type of *Expression* is a subtype of `Iterable` (exemplified in Figure 3.4) and the second when *Expression* has an array type `T[]` (exemplified in Figure 3.5), where *Expression* is the value over which the enhanced `for` statement iterates.

3.4 Extracting recursive calls to statements

This pass extracts recursive calls in the method body to separate statements (*LocalVariableDeclarationStatements*[4] with the method call as the *VariableInitializer*[5]), if the call is not already

[1] https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-FieldDeclaration
[2] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.14
[3] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.14.2
[4] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-LocalVariableDeclarationStatement
[5] https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-VariableInitializer

```java
final List<Integer> list = Arrays.asList(0, 1, 2);
for (Integer integer : list) {
  System.out.println(integer);
}
```

(a) Before

```java
final List<Integer> list = Arrays.asList(0, 1, 2);
for (Iterator<Integer> iterator = list.iterator();
    iterator.hasNext(); ) {
  Integer integer = iterator.next();
  System.out.println(integer);
}
```

(b) After

Figure 3.4: `Foreach` loop to iterator `for` loop conversion

```java
final int[] array = {0, 1, 2};
for (int n : array) {
  System.out.println(n);
}
```

(a) Before

```java
final int[] array = {0, 1, 2};
for (int i = 0; i < array.length; i++) {
  int n = array[i];
  System.out.println(n);
}
```

(b) After

Figure 3.5: `Foreach` loop to indexed `for` loop conversion

the *VariableInitializer* of a *LocalVariableDeclarationStatement* or the right-hand side of an *Assignment*[1] having an *ExpressionStatement*[2] as parent. The pass does not affect recursive methods with `void` return type, because recursive calls to these methods are already part of a separate *ExpressionStatement*.

This code modification is necessary because the expression in which a recursive call may be embedded gets evaluated only after returning from the recursive call. By separating the point of return from recursion from the following computations, it becomes possible to jump in the method body to the statements after the call. An example is provided in Figure 3.6.

```java
static int fib(int n) {
  if (n == 0) {
    return 0;
  }
  else {
    if (n == 1) {
      return 1;
    } else {
      return fib( n: n - 1) + fib( n: n - 2);
    }
  }
}
```

(a) Before

```java
static int fib(int n) {
  if (n == 0) {
    return 0;
  }
  else {
    if (n == 1) {
      return 1;
    } else {
      int temp = fib( n: n - 1);
      int temp1 = fib( n: n - 2);
      return temp + temp1;
    }
  }
}
```

(b) After

Figure 3.6: Extracting recursive calls to statements

3.5 Adding the frame class

Some of the previous passes potentially add new local variables to the method. *Renaming variables to unique names* adds new variables by renaming the ones which could generate name clashes in this pass. *Replacing `foreach` loops with `for` loops* adds a new `iterator` variable

[1] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-Assignment
[2] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-ExpressionStatement

for iterator-based `foreach` loops and an index variable `i` for array-based `foreach` loops. If there is already a local variable named `iterator` or `i`, in the method body, the pass generates a unique variable name by appending indexes to these names, to avoid name clashes. The third pass which may add new local variables is *Extracting recursive calls to statements*, because the recursive calls are extracted to local variable declaration statements named `temp`, with potential name clashes being again avoided with indexes, if necessary.

This pass generates a `private static` nested class named by the capitalized name of the method followed by the `Frame` suffix. The fields of the frame class represent the variables of the method (both formal parameters and local variables) which are relevant after a recursive call (whose scope contains at least a recursive call). They are declared as `private`, because the enclosing class can still access the fields of this class. The frame class also contains a `private` constructor which initializes only the fields representing the formal parameters of the method, because the fields corresponding to local variables are initialized by assignment in the method body, after another pass. Until they are initialized in the method body, they have the corresponding default values. By virtue of *Definite Assignment*[1], provided that the method compiles without errors before the refactoring, there will not be any problems involving using uninitialized local variables.

There is an additional `int` field named `block` in the frame class, which represents the index of the block of code to which the method jumps when returning from the callee to continue executing the statements in the caller after the recursive call.

An example of a frame class in provided in Figure 3.7. The frame class contains the formal parameters of the method as fields. The field `iterator` has been generated because the `foreach` loop has been converted to a `for` loop, since it contains a recursive call. The variable `n` is added as a corresponding field to the frame class, because its scope contains a recursive call. On the other hand, the local variable `ctc` does not have a corresponding field in the frame class, because its scope does not include a recursive call. Finally, the `block` field is added to the fields of the frame class.

3.6 Incorporating the body

The passes so far have been preparatory ones, such that the main part of the algorithm can take place without problems. This pass may seem out of place, but its purpose is to ensure that each code manipulation until the final result still produces valid code, because this requirement is necessary later for resolving the method calls to the called methods and finding the usages of the local variables using the infrastructure provided by Intellij IDEA.

The call stack is simulated in the method body by using a `stack` object. Whenever there is a call to the containing method, a new frame object is pushed onto the `stack`. This is why after declaring the `stack` object, the next statement in the body of the method is a `push` call, with the arguments equal to the formal parameters. This frame represents the first method call from outside this method.

If the return type of the method is not `void`, the next statement in the method body is a declaration statement of a local variable named `ret`, which is used to save the return value of the callee, to be used later by the caller in a recursive call.

The next statement is a `while` statement which loops as long as the simulated call `stack` is not empty. Because the body of the statement executes the code in the context of a current stack `frame`, the first statement in the `while` body is a declaration statement of a local variable named `frame`, initialized as the frame at the top of the call stack. After this statement, the original body of the method is included here as a block statement.

[1] https://docs.oracle.com/javase/specs/jls/se8/html/jls-16.html

```java
private static void dfsCTC(Graph g, Node node) {
  node.setIndex(g.time);
  node.setLowLink(g.time);
  g.time += 1;
  g.stack.push(node);
  node.setInStack(true);

  for (Iterator<Node> iterator = g.getNeighbours(node).iterator();
       iterator.hasNext(); ) {
    Node n = iterator.next();
    if (!n.wasVisited()) {
      dfsCTC(g, n);
      node.setLowLink(Math.min(node.getLowLink(), n.getLowLink()));
    } else {
      if (n.isInStack()) {
        node.setLowLink(Math.min(node.getLowLink(), n.getIndex()));
      }
    }
  }

  if (node.getLowLink() == node.getIndex()) {
    final List<Node> ctc = new ArrayList<>();
    Node n;
    do {
      n = g.stack.pop();
      n.setInStack(false);
      ctc.add(n);
    } while (!n.equals(node));
    g.ctc.add(ctc);
  }
}
```

(a) Before

```java
private static void dfsCTC(Graph g, Node node) {
  node.setIndex(g.time);
  node.setLowLink(g.time);
  g.time += 1;
  g.stack.push(node);
  node.setInStack(true);

  for (Iterator<Node> iterator = g.getNeighbours(node).iterator();
       iterator.hasNext(); ) {
    Node n = iterator.next();
    if (!n.wasVisited()) {
      dfsCTC(g, n);
      node.setLowLink(Math.min(node.getLowLink(), n.getLowLink()));
    } else {
      if (n.isInStack()) {
        node.setLowLink(Math.min(node.getLowLink(), n.getIndex()));
      }
    }
  }

  if (node.getLowLink() == node.getIndex()) {
    final List<Node> ctc = new ArrayList<>();
    Node n;
    do {
      n = g.stack.pop();
      n.setInStack(false);
      ctc.add(n);
    } while (!n.equals(node));
    g.ctc.add(ctc);
  }
}

private static class DfsCTCFrame {
  private Graph g;
  private Node node;
  private Iterator<Node> iterator;
  private Node n;
  private int block;

  private DfsCTCFrame(Graph g, Node node) {
    this.g = g;
    this.node = node;
  }
}
```

(b) After

Figure 3.7: Adding the frame class

If the return type of the method is not `void`, there is a final `return` statement in the method body after the `while` statement, which returns the value of the `ret` local variable, which holds the return value of the method after the control flow exits the `while` statement.

An example of this pass is provided in Figure 3.8.

3.7 Replacing references with field accesses

This pass replaces references to the relevant formal parameters and local variables in the original body of the method with field accesses to the corresponding fields of the `frame` object, where relevant refers to variables whose scope contains at least one recursive call.

An example of this pass is provided in Figure 3.9. The three references to the local variable `n` are replaced with field accesses to the corresponding field of the `frame` object.

3.8 Replacing declarations having initializers with assignments

This pass replaces the local variable declarations in the method body which have initializers with assignments to the corresponding fields of the `frame` object. An example of this pass is provided in Figure 3.10. The local variable `mid` having an initializer gets replaced with an assignment to the corresponding field of the `frame` object.

```java
static int factorial(int n) {
  if (n == 0) {
    return 1;
  }
  else {
    int temp = factorial( n: n - 1);
    return n * temp;
  }
}

private static class FactorialFrame {
  private int n;
  private int block;

  private FactorialFrame(int n) {
    this.n = n;
  }
}
```

(a) Before

```java
static int factorial(int n) {
  final Deque<FactorialFrame> stack = new ArrayDeque<>();
  stack.push(new FactorialFrame(n));
  int ret = 0;
  while (!stack.isEmpty()) {
    final FactorialFrame frame = stack.peek();
    {
      if (n == 0) {
        return 1;
      } else {
        int temp = factorial( n: n - 1);
        return n * temp;
      }
    }
  }
  return ret;
}

private static class FactorialFrame {
  private int n;
  private int block;

  private FactorialFrame(int n) {
    this.n = n;
  }
}
```

(b) After

Figure 3.8: Incorporating the body

3.9 Generating the control flow graph

The body of the `while` statement previously generated contains a `switch` statement, whose body itself contains `cases` for each *basic block* in the original body of the method, as defined below.

Because the Java programming language does not provide a `goto` statement, there is no direct a way of jumping at a later point in code (useful when returning from a recursive call to execute the statements after the call). But this problem can be solved if the code is broken into *basic blocks*, by generating the control flow graph of the method body. Each basic block corresponds to a `case` inside the `switch` statement from above. Jumping between different blocks is possible by using the `block` field of the `frame` object.

The structure of the CFG (Control Flow Graph) presented here is inspired from LLVM IR (Intermidiate Representation)[1].

In this context, a `Block` is defined as a list of non-terminal `NormalStatements` terminated by a `TerminatorStatement`. This definition is inspired from the definition of a *basic block* as presented in *Chapter 8.4* of [2].

A `NormalStatement` is a wrapper over a `PsiStatement`.

A `TerminatorStatement` is either a `ReturnStatement`, a `ConditionalJumpStatement`, an `UnconditionalJumpStatement` or a `SwitchStatement`.

A `ReturnStatement` is a wrapper over a `PsiReturnStatement`.

A `ConditionalJumpStatement` jumps from the containing `Block` to one of two `Blocks`, based on the value of a `PsiExpression`.

An `UnconditionalJumpStatement` jumps from the containing `Block` to another `Block`.

A `SwitchStatement` corresponds to a `PsiSwitchStatement`. It jumps to one of multiple blocks corresponding to the blocks in the body of the original `PsiSwitchStatement`, based on its associated `PsiExpression`.

[1] https://llvm.org/docs/LangRef.html

```java
static int factorial(int n) {
  final Deque<FactorialFrame> stack = new ArrayDeque<>();
  stack.push(new FactorialFrame(n));
  int ret = 0;
  while (!stack.isEmpty()) {
    final FactorialFrame frame = stack.peek();
    {
      if (n == 0) {
        return 1;
      } else {
        int temp = factorial( n: n - 1);
        return n * temp;
      }
    }
  }
  return ret;
}

private static class FactorialFrame {
  private int n;
  private int block;

  private FactorialFrame(int n) {
    this.n = n;
  }
}
```

(a) Before

```java
static int factorial(int n) {
  final Deque<FactorialFrame> stack = new ArrayDeque<>();
  stack.push(new FactorialFrame(n));
  int ret = 0;
  while (!stack.isEmpty()) {
    final FactorialFrame frame = stack.peek();
    {
      if (frame.n == 0) {
        return 1;
      } else {
        int temp = factorial( n: frame.n - 1);
        return frame.n * temp;
      }
    }
  }
  return ret;
}

private static class FactorialFrame {
  private int n;
  private int block;

  private FactorialFrame(int n) {
    this.n = n;
  }
}
```

(b) After

Figure 3.9: Replacing references with field accesses

The list of `Blocks` for the method body is generated by a `JavaRecursiveElementVisitor`, which performs different operations for each type of `PsiElement` encountered. Each `Block` has an `id` field which represents the label of its corresponding `case`.

The visitor initializes a `myCurrentBlock` field before visiting the actual body of the method. This is the entry block of the function. The visitor also keeps two private mappings (`myBreakTargets` and `myContinueTargets`) from potential `break` and `continue` target statements (`for`, `while`, `do-while` loops and `switch` statements) to the basic blocks to which control flow jumps after executing the corresponding `break` or `continue` statement. The following subsections present how the visitor processes each type of relevant `PsiElement` encountered.

3.9.1 Visiting a `PsiCodeBlock`

The body of a method is represented as a `PsiCodeBlock`. The visitor processes each `PsiStatement` in the `PsiCodeBlock`.

In this context, processing means that if the `PsiStatement` contains recursive calls or is an instance of `PsiReturnStatement`, `PsiBreakStatement` or `PsiContinueStatement`, it accepts the visitor for further processing.

Otherwise, a `BreakContinueReplacerVisitor` visits the `PsiStatement` in order to replace the `break` and `continue` statements with inlined `UnconditionalJumpStatements` to the target `blocks` of these statements. The only statements affected are those whose exited or continued statement has been previously visited by the CFG generator. The target `blocks` of these statements also get their `doNotInline` flag set to `true`, so as not to be inlined by later passes. Here, an inlined `UnconditionalJumpStatement` means an assignment to the `block` field of the `frame` object with the `id` of the target `block`, followed by a `break` statement. After the replacer visitor processing finishes, a `NormalStatement` wrapper over the `PsiStatement` is added to the `myCurrentBlock`.

An important last step after processing all the `PsiStatements` in the `PsiCodeBlock` of the method is adding an explicit `ReturnStatement` to the `myCurrentBlock`, if the return type

```
void displayArray(int first, int last, List<Integer> result) {
  final Deque<DisplayArrayFrame> stack = new ArrayDeque<>();
  stack.push(new DisplayArrayFrame(first, last, result));
  while (!stack.isEmpty()) {
    final DisplayArrayFrame frame = stack.peek();
    {
      if (frame.first == frame.last) {
        frame.result.add(array[frame.first]);
      } else {
        int mid = frame.first + (frame.last - frame.first) / 2;
        displayArray(frame.first, frame.mid, frame.result);
        displayArray( first: frame.mid + 1, frame.last, frame.result);
      }
    }
  }
}

private static class DisplayArrayFrame {
  private int first;
  private int last;
  private List<Integer> result;
  private int mid;
  private int block;

  private DisplayArrayFrame(int first, int last, List<Integer> result) {
    this.first = first;
    this.last = last;
    this.result = result;
  }
}
```

(a) Before

```
void displayArray(int first, int last, List<Integer> result) {
  final Deque<DisplayArrayFrame> stack = new ArrayDeque<>();
  stack.push(new DisplayArrayFrame(first, last, result));
  while (!stack.isEmpty()) {
    final DisplayArrayFrame frame = stack.peek();
    {
      if (frame.first == frame.last) {
        frame.result.add(array[frame.first]);
      } else {
        frame.mid = frame.first + (frame.last - frame.first) / 2;
        displayArray(frame.first, frame.mid, frame.result);
        displayArray( first: frame.mid + 1, frame.last, frame.result);
      }
    }
  }
}

private static class DisplayArrayFrame {
  private int first;
  private int last;
  private List<Integer> result;
  private int mid;
  private int block;

  private DisplayArrayFrame(int first, int last, List<Integer> result) {
    this.first = first;
    this.last = last;
    this.result = result;
  }
}
```

(b) After

Figure 3.10: Replacing declarations having initializers with assignments

of the method is `void` and the body does not already contain an explicit `return` statement. This is almost always the case because these methods do not specify a final `return` statement, since it is redundant.

3.9.2 Visiting a `PsiBlockStatement`

Since the `PsiBlockStatement` is merely a wrapper over a `PsiCodeBlock`, visiting it translates to processing the statements of its contained `PsiCodeBlock`, just as already specified above.

3.9.3 Visiting a `PsiReturnStatement`

A `ReturnStatement` wrapper over the `PsiReturnStatement` is added to the `myCurrentBlock`.

3.9.4 Visiting a `PsiBreakStatement`

The `exitedStatement` corresponding to the `break` statement is computed. If the `exitedStatement` is not contained in the `myBreakTargets` map, a `NormalStatement` wrapper over the `PsiBreakStatement` is added to the `myCurrentBlock`. Otherwise, an `UnconditionalJumpStatement` to the target `block` is added to the `myCurrentBlock`.

3.9.5 Visiting a `PsiContinueStatement`

The `continuedStatement` corresponding to the `continue` statement is computed. If the `continuedStatement` is not contained in the `myContinueTargets` map, a `NormalStatement` wrapper over the `PsiContinueStatement` is added to the `myCurrentBlock`. Otherwise, an `UnconditionalJumpStatement` to the target `block` is added to the `myCurrentBlock`.

3.9.6 Visiting a `PsiMethodCallExpression`

Given the previous passes and the processing of `PsiStatements` presented above, the visitor actually visits only recursive calls. These calls appear only as right-hand sides of `PsiAssignmentExpressions` or as initializers of `PsiVariables` in `PsiDeclarationStatements` (generated by the *Extracting recursive calls to statements* pass).

The recursive call is transformed in a `push` call to the `stack`, where the new `frame` object is initialized using the `PsiExpressions` provided as arguments to the recursive call. A new `Block` is generated and it is marked so as not to be inlined by a later pass. Then an `UnconditionalJumpStatement` which jumps to the new block is added to the `myCurrentBlock`. After this, the `myCurrentBlock` is changed to the new block. The `PsiStatement` in which the recursive call appears is added to the new `myCurrentBlock`, but having the recursive call replaced with a `PsiExpression` referring to the `ret` variable.

3.9.7 Visiting an `PsiIfStatement`

Two new blocks are generated: the `thenBlock` and the `mergeBlock`. An optional third block (the `elseBlock`) gets generated if the `if` statement has an `else` branch. A `ConditionalJumpStatement` is added to the `myCurrentBlock`. It jumps to either the `thenBlock` or the `elseBlock`. If there is no `elseBlock`, it jumps to the `mergeBlock` instead.

The `myCurrentBlock` is set to the `thenBlock` and the `then` branch accepts the visitor. After this, an `UnconditionalJumpStatement` to the `mergeBlock` is added to the `myCurrentBlock`, which may be other than the `thenBlock`, because of the visitor possibly changing it when visiting the `then` branch.

If there is an `else` branch, it is treated similarly to the `then` branch. Finally, the `myCurrentBlock` is set to the `mergeBlock`.

The description above is depicted in Figure 3.11. The grey boxes represent basic blocks which are optional and the dotted lines represent arbitrary subgraphs.

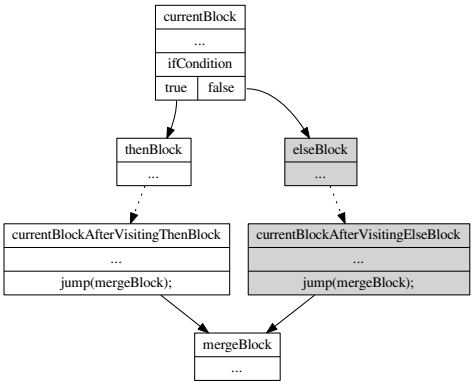


Figure 3.11: If statement CFG

3.9.8 Visiting loop statements

There are four possible types of loop statements in a Java PSI tree: `PsiWhileStatement`, `PsiDoWhileStatement`, `PsiForeachStatement` and `PsiForStatement`. The `PsiForeachStatements` have been replaced with equivalent `PsiForStatements` in the *Replacing `foreach` loops with `for` loops* pass. The other three types of loop statements are similar, with `PsiWhileStatement` and `PsiDoWhileStatement` being special cases of `PsiForStatement`.

In the case of a `PsiForStatement`, if it has any initialization, the corresponding `PsiStatements` are added to the `myCurrentBlock`. There are three cases here, based on how Intellij IDEA parses the initialization part of a `for` statement. If the initialization is an instance of `PsiEmptyStatement`, there is no actual statement to add. If the initialization is an instance of `PsiExpressionStatement`, this statement is added to the `myCurrentBlock`. Finally, if the initialization is an instance of `PsiExpressionListStatement`, each expression in this list is added as a statement to the `myCurrentBlock`. The case when the initialization is a`PsiDeclarationStatement` is not valid here, because of the *Replacing declarations having initializers with assignments* pass.

If the `condition` expression of a loop statement is either absent (it is optional in a `PsiForStatement`) or is a constant expression[1] which evaluates to `true`, the `conditionBlock` is `null`. Otherwise, the `conditionBlock` is generated.

The `bodyBlock` is always generated because all loop statements have a body.

The `update` part of a loop statement is always `null` for `PsiWhileStatements` and `PsiDoWhileStatements`. It is also optional for `PsiForStatements`. If it is not `null`, the `updateBlock` is generated.

The `mergeBlock` is always generated because control flow jumps there after exiting the loop.

The `actualConditionBlock` is the `conditionBlock` if the `conditionBlock` is not `null` or the `bodyBlock` otherwise. The `actualUpdateBlock` is the `updateBlock` if the `updateBlock` is not `null` or the `actualConditionBlock` otherwise.

The `break` target of the loop statement is the `mergeBlock` and the `continue` target of the loop statement is the `actualUpdateBlock`.

An `UnconditionalJumpStatement` is added to the `myCurrentBlock`. If the body of the loop statement has to be executed at least once (in the case of a `PsiDoWhileStatement`), the jump target is the `bodyBlock`. Otherwise, it is the `actualConditionBlock`.

If the `conditionBlock` is not `null`, the `myCurrentBlock` is set to the `conditionBlock` and a `ConditionalJumpStatement` is added to the `myCurrentBlock`. It jumps to either the `bodyBlock` or the `mergeBlock` based on the `condition` of the loop statement.

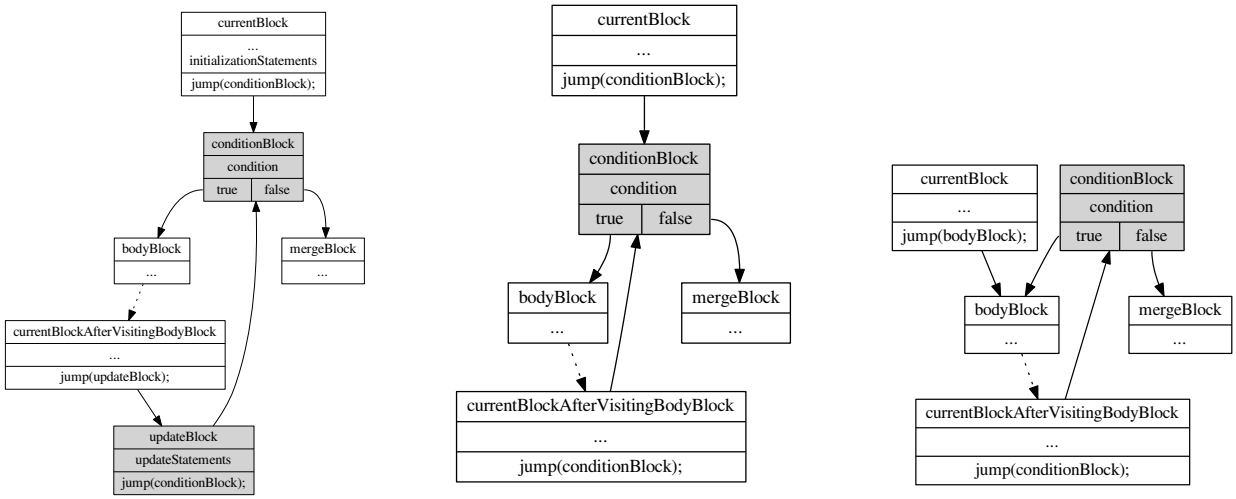
If the loop statement has an update part (only in the case of `PsiForStatements`), the `myCurrentBlock` is set to the `updateBlock`. The statements corresponding to the update part of the `for` statement are added to the `myCurrentBlock`, in a similar fashion to the statements corresponding to the initialization part of the `for` statement. Then an `UnconditionalJumpStatement` to the `actualConditionBlock` is added to the `myCurrentBlock`.

The `myCurrentBlock` is set to the `bodyBlock` and the visitor visits the `body` of the loop statement. After this, an `UnconditionalJumpStatement` to the `actualUpdateBlock` is added to the `myCurrentBlock`, which may not be the `bodyBlock` anymore, because of the visitor possibly changing it when visiting the body of the loop statement.

[1] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.28

Finally, the `myCurrentBlock` is set to the `mergeBlock`.

The description above is depicted in Figure 3.12. The grey boxes represent basic blocks which are optional and the dotted lines represent arbitrary subgraphs. The CFG in Figure 3.12a is the most general one. In the case of a `PsiWhileStatement`, the `updateBlock` is absent and the CFG becomes the one in Figure 3.12b. In the case of a `PsiDoWhileStatement`, the only difference from the `while` statement CFG is the target of the `UnconditionalJumpStatement`, which is the `bodyBlock` and not the `conditionBlock`, because the body is executed at least once in a `do-while` statement.



(a) For statement CFG (b) While statement CFG (c) Do-while statement CFG

Figure 3.12: Loop statements CFGs

3.9.9 Visiting a `PsiSwitchStatement`

After the *Replacing single statements with block statements* pass, the body of a `switch` statement contains only either `PsiSwitchLabelStatements` or `PsiBlockStatements`.

First, a `mergeBlock` is generated, the `myCurrentBlock` is saved in an `oldCurrentBlock` variable and the `previousCurrentBlock` is set to `null`. An empty list of `Statements` called `statements` is also initialized. Then each statement in the body of the `switch` statement is processed.

If the current statement is an instance of `PsiSwitchLabelStatement`, then a `NormalStatement` wrapping the `PsiSwitchLabelStatement` is added to the `statements` list.

If the current statement is an instance of `PsiBlockStatement`, then a `newBlock` is created and an `UnconditionalJumpStatement` to this `newBlock` is added to the `statements` list. If the `previousCurrentBlock` is not `null` and it is not finished (its last statement is not a `TerminatorStatement`), then an `UnconditionalJumpStatement` to the `newBlock` is added to the `previousCurrentBlock`. This happens when the control flow falls through the `previousCurrentBlock` to the `newBlock`. Then the `myCurrentBlock` is set to `newBlock` and the visitor visits the current `PsiBlockStatement`. After this, the

previousCurrentBlock is set to the actual myCurrentBlock, which may be other than before the visitor visits the PsiBlockStatement.

After processing all the statements in the switch body, if the previousCurrentBlock is not null and it is not finished, an UnconditionalJumpStatement to the mergeBlock is added to the previousCurrentBlock. This happens when control flow falls through the last block in the switch body to the mergeBlock.

Then the myCurrentBlock is restored from oldCurrentBlock and a SwitchStatement is added to the myCurrentBlock, containing the the PsiExpression associated with the PsiSwitchStatement and the computed statements list. The myCurrentBlock is then set to the mergeBlock.

An example of the generated basic blocks is provided in Figure 3.13. Each block is assigned to a case inside a switch statement which replaces the incorporated body of the method. The switch statement has the block field of the frame object as its expression. The CFG of the method body can be better visualized in the Figure 3.14. There are two basic blocks with ids equal to 6 and 3, which are unreachable from the first block of the CFG (with id equal to 0). The red lined blocks are blocks which should explicitly not be inlined, because they are blocks which appear after a recursive call or because the number of incoming blocks is not equal to 1 (it is 0 in this particular case).

3.10 Removing unreachable blocks

There could be blocks generated in the previous pass which are *unreachable*, meaning that there is no path from the *entry block* (whose id is equal to 0) of the control flow graph to these blocks. The entry block is reachable by default.

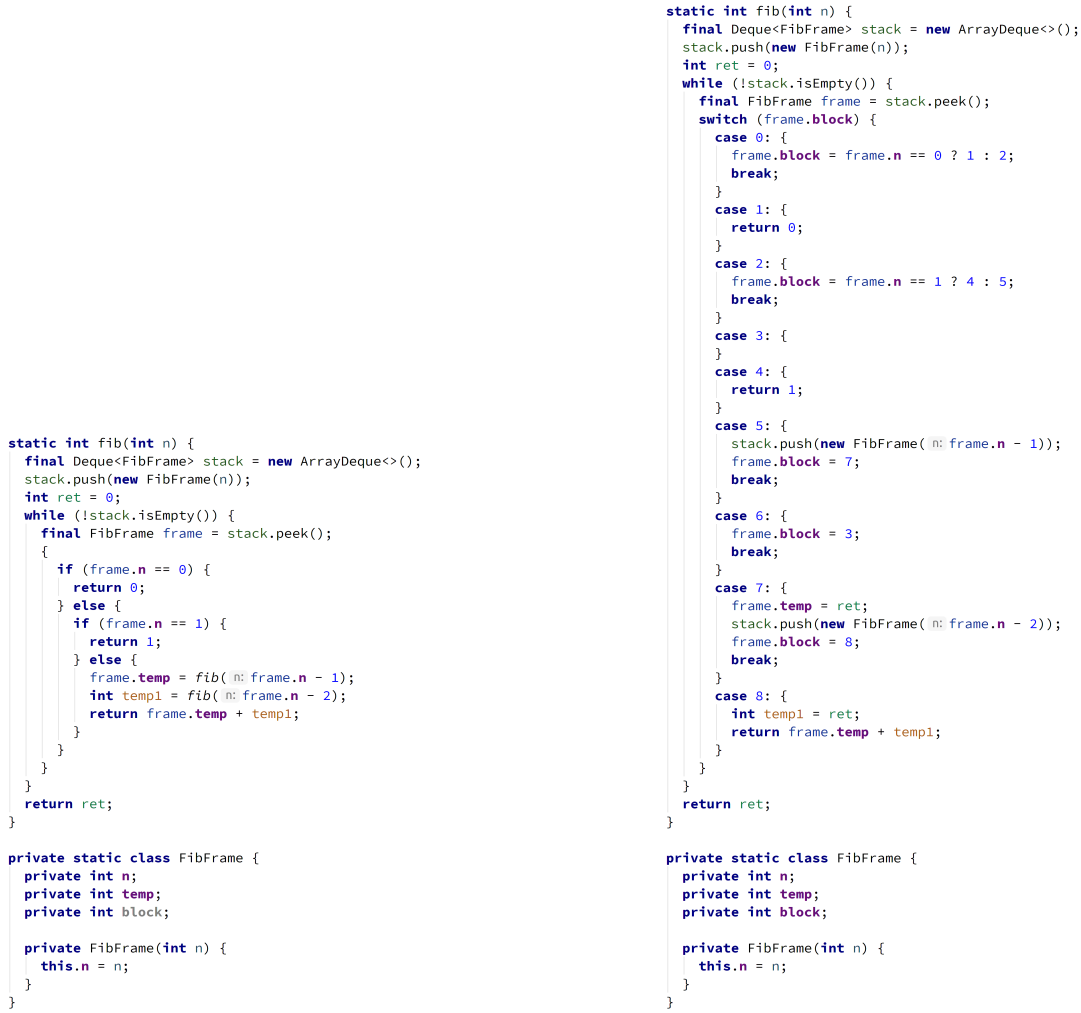
This pass visits the control flow graph, which is a directed graph. If the method body does not contain loops, the graph is acyclic; otherwise, it contains at least one cycle. A breadth-first search algorithm is used to compute the set of reachable blocks from the entry block of the control flow graph. After the search is finished, the blocks which are not contained in the set of reachable blocks are removed from the list of all blocks representing the control flow graph.

When this pass is applied to the CFG in Figure 3.14, because the blocks with ids equal to 3 and 6 are unreachable from the entry block, they get removed. Even though the block with id 3 is reachable from the block with id 6, it is not reachable from the entry block (with id 0), so it still gets removed.

3.11 Removing trivial blocks

A *trivial block* is defined as a block which is not the first block (having id equal to 0) and contains only an UnconditionalJumpStatement. These blocks can be removed be making the incoming blocks jump directly to the block to which the trivial block jumps. A trivial block can appear, for example, when the last statement in the body of a loop is an if statement. Thus the mergeBlock of the if statement contains only an UnconditionalJumpStatement to the actualUpdateBlock.

An example of this pass is provided in Figure 3.15. The block with id equal to 6 is trivial and it gets removed by making the incoming blocks (with ids 7 and 5) jump directly to the block to which the trivial block jumps, namely the block with id equal to 1. The blocks with ids 1 and 6 cannot be inlined because they both have two incoming blocks and the block with id 7 cannot be inlined either because it is a block appearing after a recursive call.



```java
static int fib(int n) {
  final Deque<FibFrame> stack = new ArrayDeque<>();
  stack.push(new FibFrame(n));
  int ret = 0;
  while (!stack.isEmpty()) {
    final FibFrame frame = stack.peek();
    {
      if (frame.n == 0) {
        return 0;
      } else {
        if (frame.n == 1) {
          return 1;
        } else {
          frame.temp = fib( n: frame.n - 1);
          int temp1 = fib( n: frame.n - 2);
          return frame.temp + temp1;
        }
      }
    }
  }
  return ret;
}

private static class FibFrame {
  private int n;
  private int temp;
  private int block;

  private FibFrame(int n) {
    this.n = n;
  }
}
```

(a) Before

```java
static int fib(int n) {
  final Deque<FibFrame> stack = new ArrayDeque<>();
  stack.push(new FibFrame(n));
  int ret = 0;
  while (!stack.isEmpty()) {
    final FibFrame frame = stack.peek();
    switch (frame.block) {
      case 0: {
        frame.block = frame.n == 0 ? 1 : 2;
        break;
      }
      case 1: {
        return 0;
      }
      case 2: {
        frame.block = frame.n == 1 ? 4 : 5;
        break;
      }
      case 3: {
      }
      case 4: {
        return 1;
      }
      case 5: {
        stack.push(new FibFrame( n: frame.n - 1));
        frame.block = 7;
        break;
      }
      case 6: {
        frame.block = 3;
        break;
      }
      case 7: {
        frame.temp = ret;
        stack.push(new FibFrame( n: frame.n - 2));
        frame.block = 8;
        break;
      }
      case 8: {
        int temp1 = ret;
        return frame.temp + temp1;
      }
    }
  }
  return ret;
}

private static class FibFrame {
  private int n;
  private int temp;
  private int block;

  private FibFrame(int n) {
    this.n = n;
  }
}
```

(b) After

Figure 3.13: Generating the control flow graph

3.12 Inlining blocks

The blocks which have only one incoming block and are not blocks that appear after a recursive call can be inlined in the incoming block. This is accomplished by marking the blocks that can be inlined first and then visiting the blocks that cannot be inlined in order to generate the new blocks.

An example of this pass is provided in Figure 3.16. A second example is provided in Figure 3.17. This is the reduced control flow graph obtained from Figure 3.15b.

3.13 Replacing `return` statements

If the return type of the method is not `void`, then an assignment to the `ret` variable with the right-hand side representing the expression of the `return` statement is added in the place of the `return` statement. Then the current stack frame is popped from the stack by a call to

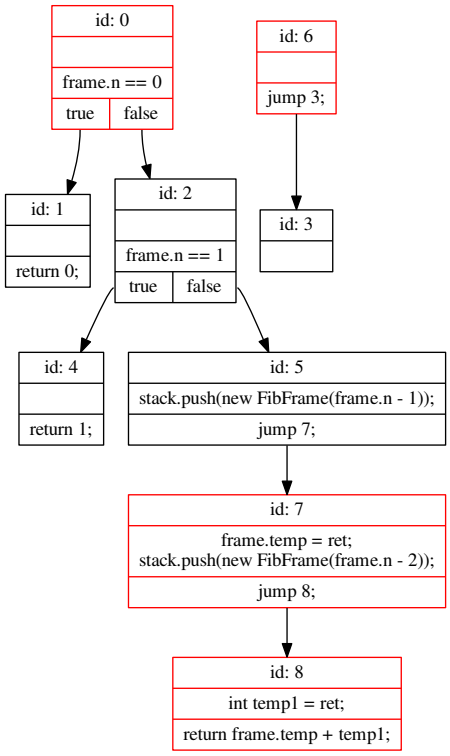
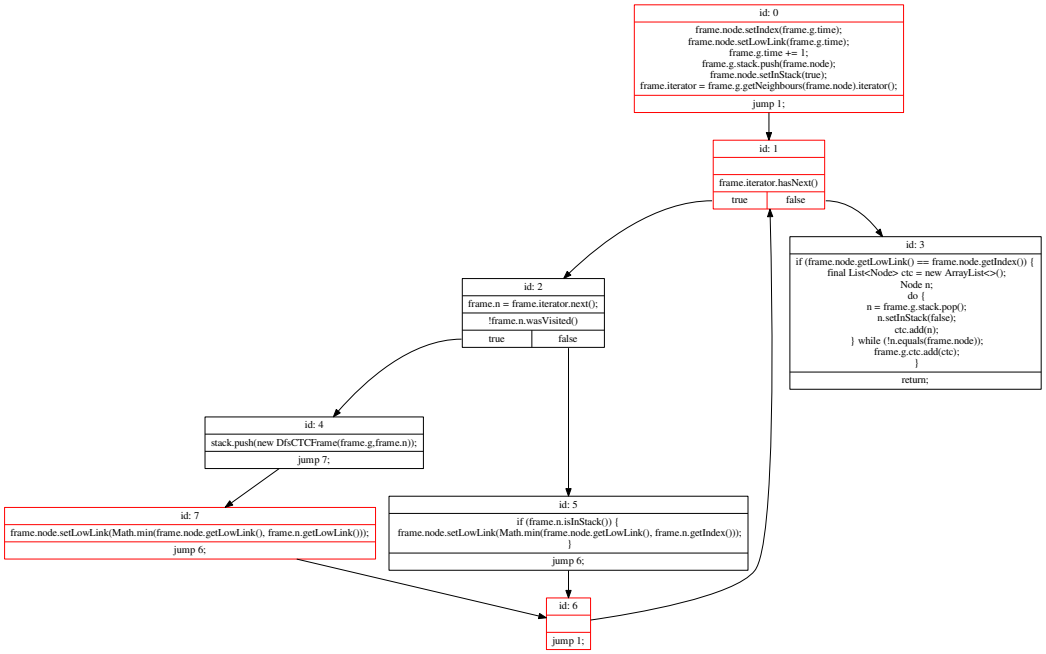


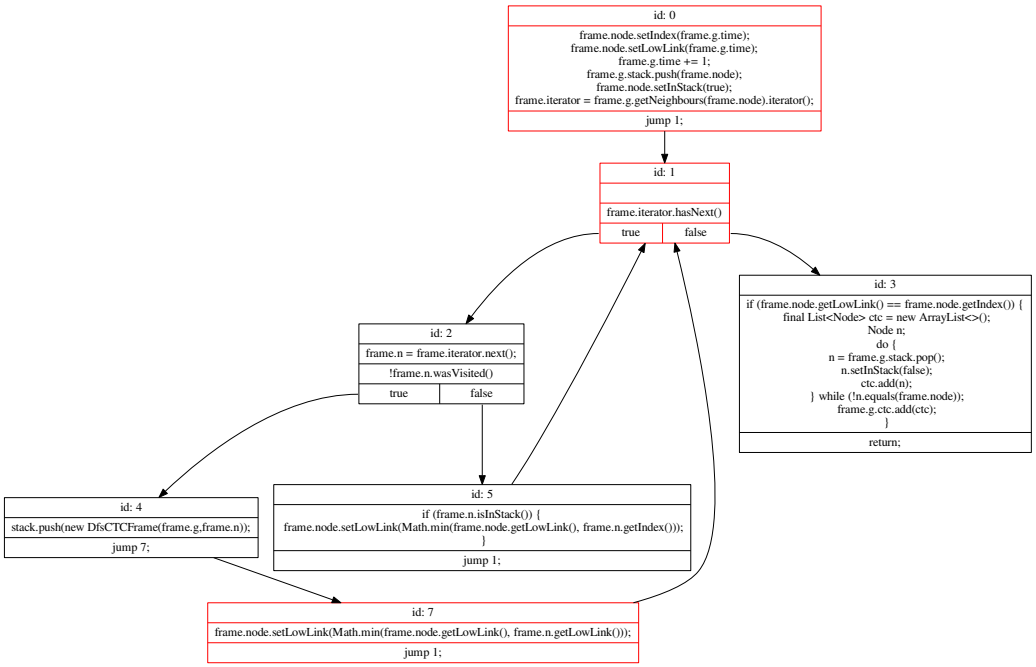
Figure 3.14: The CFG

`pop()` on the `stack` object. Finally, a `break` statement is added to the parent block of the `return` statement. The target of this `break` statement needs to be the `switch` statement enclosing the basic blocks. This is why if there is at least one `return` statement in the method which is still included in a statement which can be a `break` target after generating the basic blocks, the `switch` statement enclosing the basic blocks is prepended with a label called `switchLabel`. The `break` statements corresponding to these `return` statements also have the label `switchLabel`. Otherwise, these `break` statements would transfer control to the statement after the enclosing statement and not to the statement after the `switch` statement, as it should happen.

An example of this pass is provided in Figure 3.18.

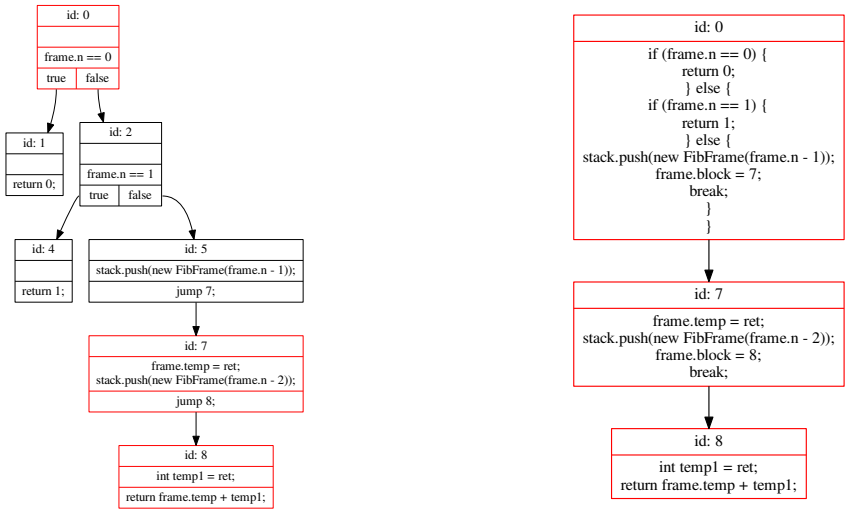


(a) Before



(b) After

Figure 3.15: Removing trivial blocks



(a) Before (b) After

Figure 3.16: Inlining blocks

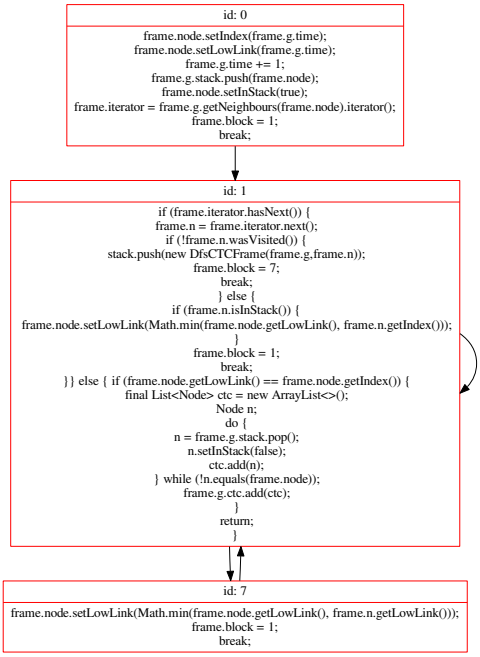


Figure 3.17: The reduced CFG

```java
static int fib(int n) {
  final Deque<FibFrame> stack = new ArrayDeque<>();
  stack.push(new FibFrame(n));
  int ret = 0;
  while (!stack.isEmpty()) {
    final FibFrame frame = stack.peek();
    switch (frame.block) {
      case 0: {
        if (frame.n == 0) {
          return 0;
        } else {
          if (frame.n == 1) {
            return 1;
          } else {
            stack.push(new FibFrame( n: frame.n - 1));
            frame.block = 7;
            break;
          }
        }
      }
      case 7: {
        frame.temp = ret;
        stack.push(new FibFrame( n: frame.n - 2));
        frame.block = 8;
        break;
      }
      case 8: {
        int temp1 = ret;
        return frame.temp + temp1;
      }
    }
  }
  return ret;
}

private static class FibFrame {
  private int n;
  private int temp;
  private int block;

  private FibFrame(int n) {
    this.n = n;
  }
}
```

(a) Before

```java
static int fib(int n) {
  final Deque<FibFrame> stack = new ArrayDeque<>();
  stack.push(new FibFrame(n));
  int ret = 0;
  while (!stack.isEmpty()) {
    final FibFrame frame = stack.peek();
    switch (frame.block) {
      case 0: {
        if (frame.n == 0) {
          ret = 0;
          stack.pop();
          break;
        } else {
          if (frame.n == 1) {
            ret = 1;
            stack.pop();
            break;
          } else {
            stack.push(new FibFrame( n: frame.n - 1));
            frame.block = 7;
            break;
          }
        }
      }
      case 7: {
        frame.temp = ret;
        stack.push(new FibFrame( n: frame.n - 2));
        frame.block = 8;
        break;
      }
      case 8: {
        int temp1 = ret;
        ret = frame.temp + temp1;
        stack.pop();
        break;
      }
    }
  }
  return ret;
}

private static class FibFrame {
  private int n;
  private int temp;
  private int block;

  private FibFrame(int n) {
    this.n = n;
  }
}
```

(b) After

Figure 3.18: Replacing `return` statements

Chapter 4

Testing

Each of the thirteen steps of the algorithm have been tested independently, by developing a code inspection for each one. Testing a code inspection involves writing a unit test which applies the inspection on an input file and then compares the result on an expected output file. The tested inspection runs as part of an instance of the IDE without a Graphical User Interface. These tests verify that each step of the algorithm modifies only what it states to do and nothing else.

There are also unit tests which test the refactoring algorithm as a whole. These tests verify that the refactored recursive methods preserve their original semantics.

Chapter 5

Evaluating performance

The performance of the refactoring algorithm itself is tightly coupled to the code representation as an abstract syntax tree in the form of a tree of `PsiElements`. Given the recursive nature of the AST, the `Visitor` design pattern is the main mechanism used in the refactoring, because it clearly separates the algorithm (the transformations of each pass) from the data structure it operates on. The algorithm is thus implemented by overriding a method for each type of relevant `PsiElement` for the transformation in a certain pass. So the time complexity of each of the passes is roughly proportional to the size of the tree of `PsiElements` it operates on.

When it comes to evaluating the performance of the refactored methods, this can be difficult because the main use case of the refactoring is avoiding `StackOverflowErrors`. So this refactoring, when applied to cases of recursive methods which raise `StackOverflowErrors`, generates an equivalent method which does not raise `StackOverFlowErrors`. Since the original use case results in an error condition, it is harder to make a comparison of the refactored method to the original one.

To examine the execution time of a recursive method before and after applying the refactoring, a program which computes the sum of prime numbers who are smaller than a given value has been chosen. The program is given in Figure 5.1.

The average execution time of the `sumOfPrimes` recursive method (measured as a mean of ten consecutive executions) before the refactoring is `2444us`. The same execution time after applying the refactoring is `6026us`.

So the performance of the refactored method is generally worse than that of the original one. There are many reasons why this is the case. One of them is the fact that the code of the method is split across blocks inside a `switch` statement contained in a `while` loop, so there are more jumps through code during the execution than in the original method. This happens because given the lack of the `goto` statement in Java, a workaround which simulates it has been used.

Another performance penalty may come from the fact that after the refactoring, all the local variables in the method body are now part of the `frame` object, so there is an additional overhead associated with creating these objects and also with each access to a field of this object, which is not as fast as the access to a local variable on the stack in the original method.

There is also an additional performance penalty associated with the `stack` object in the refactored method, which uses an array-based implementation of a stack. Its capacity doubles when it gets full and the elements in the original stack need to be copied to the array of the expanded stack when this happens.

The code of the refactored method is generally more complex than the original code of the

```java
public class Primes {
  private static final int NUM_ITERATIONS = 10;

  private static boolean isPrime(long x) {
    long limit = (long) Math.sqrt(x);
    for (long i = 2; i <= limit; i++) {
      if (x % i == 0) {
        return false;
      }
    }
    return true;
  }

  static long sumOfPrimes(long a) {
    if (a == 1)
      return 0;
    if (isPrime(a)) {
      return a + sumOfPrimes( a: a - 1);
    } else {
      return sumOfPrimes( a: a - 1);
    }
  }

  public static void main(String[] args) {
    final long before = System.nanoTime();
    for (int i = 0; i < NUM_ITERATIONS; i++) {
      sumOfPrimes( a: 10_000);
    }
    final long after = System.nanoTime();
    System.out.println("Elapsed time: " +
      (after - before) / 1_000 / NUM_ITERATIONS + "us");
  }
}
```

(a) Before

```java
static long sumOfPrimes(long a) {
  final Deque<SumOfPrimesFrame> stack = new ArrayDeque<>();
  stack.push(new SumOfPrimesFrame(a));
  long ret = 0L;
  while (!stack.isEmpty()) {
    final SumOfPrimesFrame frame = stack.peek();
    switch (frame.block) {
      case 0: {
        if (frame.a == 1) {
          ret = 0;
          stack.pop();
          break;
        }
        if (isPrime(frame.a)) {
          stack.push(new SumOfPrimesFrame( a: frame.a - 1));
          frame.block = 4;
          break;
        } else {
          stack.push(new SumOfPrimesFrame( a: frame.a - 1));
          frame.block = 5;
          break;
        }
      }
      case 4: {
        long temp = ret;
        ret = frame.a + temp;
        stack.pop();
        break;
      }
      case 5: {
        long temp1 = ret;
        ret = temp1;
        stack.pop();
        break;
      }
    }
  }
  return ret;
}
```

(b) After

Figure 5.1: Computing the sum of primes smaller than a given number

method, so it is expected to also be less efficient. However, there is a gain when the original method raises `StackOverflowErrors`, because the refactored code resolves this problem. For the example program above, when the number under whose value the sum of primes is computed gets bigger than about `10000`, the original method raises a `StackOverflowError`. In the case this number is equal to `2000000`, as in the original problem, the execution time of the refactored method is about `2` seconds. It may seem much, but given the result cannot be compared to the original method, it is better than nothing.

This refactoring should be used only when a temporary solution to `StackOverflowErrors` is needed, as it will not generate better code than the original one. When performance is critical, a redesign of the original algorithm to avoid recursion is still needed.

Chapter 6

Conclusion

The purpose of this refactoring is to offer a temporary solution for `StackOverflowErrors`. This is achieved by simulating the call stack in the user program and building a reduced version of the control flow graph for the recursive method, which enables jumps to arbitrary places in the code. The process of embedding the control flow graph inside a `switch` and a `while` statement is necessary, given the limitation of the Java programming language of not supporting the `goto` statement, which would have enabled jumps to any point in the method body. The refactoring does not improve either the performance or the readability of the original code of the recursive method. The algorithm still needs to be improved if performance is critical.

The most difficult part of the algorithm is preserving the semantics of the original code. Given the fact that there is no formal proof for the corectness of the transformations and that not all the Java language features are supported, there is no strong guarantee about the semantic equivalence. Writing a refactoring that is accurate in almost any situation is difficult, because there are many use cases which are hard to predict beforehand.

The refactoring has been tested on about twenty different recursive methods in order to confirm its correctness in the most common use cases, by checking their behaviour before and after the refactoring. The user of the refactoring should have a way of testing the recursive method before and after applying the refactoring, probably by writing some unit tests, in order to be sure the semantics are preserved.

Chapter 7

Further development

The refactoring does not support recursive calls inside `try` statements. This is a rather obscure use case, because the exception the `try` statement would catch should be generated somewhere else inside the same method. No real usable code would be written that way.

There is basic and unstable support for recursive calls inside `switch` statements. They are supported by replacing each block in the body of the original `switch` statement with jumps to the basic blocks corresponding to these blocks. The original `switch` statement thus becomes a new terminator statement for the basic block inside which the statement appears.

A special case to be considered is tail recursion. In the case of tail recursive methods, which are methods whose last statement is a recursive call, the recursion can be removed by placing the body of the method inside a `while` loop and then replacing the `return` statements with assignments to the formal parameters of the method, ensuring that the order of assignments respects the order of evaluation of the arguments to the recursive call. Auxiliary variables can be used here, if necessary.

In the case of tail recursive methods with `void` return type, the transformation is more complicated because it is difficult to determine the last statement executed on each code path in the method body, as opposed to methods with no `void` return type, where the last statements executed on each code path are explicit `return` statements.

Bibliography

[1] IntelliJ Platform SDK Documentation. https://www.jetbrains.org/intellij/sdk/docs.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[4] Martin Fowler. *Refactoring: Improving the design of existing code.* Addison Wesley, 1999.

[5] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification, Java SE 8 Edition. https://docs.oracle.com/javase/specs/jls/se8/html/index.html, February 2015.

[6] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[7] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java Virtual Machine Specification, Java SE 8 Edition. https://docs.oracle.com/javase/specs/jvms/se8/html/index.html, February 2015.