Karlsruher Institut für Technologie

Fakultät für Informatik

Lehrstuhl Programmierparadigmen
– IPD Snelting

Bachelor Thesis

Towards a Dalvik Frontend for Programme Analysis

Computing an Intermediate Representation of Dalvik Bytecode in SSA-Form

Patrick Kuhn

February 19, 2012

Examiner: Prof. Dr.-Ing. Gregor Snelting
Supervisor: Dipl.-Inf. Univ. Jürgen Graf

Karlsruher Institut für Technologie

Fakultät für Informatik

Lehrstuhl Programmierparadigmen
– IPD Snelting

Bachelorarbeit

Entwicklung der Basis eines Dalvik Frontends zur Programmanalyse

Berechnung einer Zwischensprache aus Dalvik Bytecode in der SSA-Form

Patrick Kuhn

February 19, 2012

Prüfer: Prof. Dr.-Ing. Gregor Snelting
Betreuer: Dipl.-Inf. Univ. Jürgen Graf

For Katja,
who will give me cake for mentioning her here.

Contents

Erklärung zur eigenständigen Erarbeitung

Hiermit erkläre ich, die vorliegende Ausarbeitung selbstständig und ohne unzulässige fremde Hilfe angefertigt zu haben.
Außer den angegebenen Quellen habe ich keine weiteren Hilfsmittel verwendet.

Schwetzingen, den 19. Februar 2012

Patrick Kuhn

1. Introduction

On October 21, 2008 Google made the Android operating system for mobile phones available. Android had a fast grow in functionality and distribution throughout the world. In 2010 Android's market share was 22.7% which made it the second most sold operating system, preceded only by Symbian.

Android's base is a Linux-kernel, usually programmes are written in Java and must be compiled with the Android-SDK. Normally compiling Java code creates bytecode (see next chapter) which is run in a virtual machine (VM), Android uses a special form of bytecode. The main difference is, that Bytecode of Java on PC is run in the Java-VM, which is stack-based, Android uses its own virtual machine called Dalvik-VM, which uses register-based instructions.

There are many Android applications, so-called apps. Each day the amount of apps is growing. For many reasons it makes sense to analyse the code of an app, e.g. to follow "confidential" data to see whether it is passed outside unintentionally or to find spots for potential optimisations.

There are already frameworks to create analysis tools, such as:

- WALA [Wal]
- Soot [Soo]

Unfortunately there are no frameworks which support data-flow analysis on Android bytecode. Our target is to create such an analysis tool for Android bytecode by extending one of the currently available frameworks. These frameworks use an intermediate language to increase extendability for new languages and to simplify later algorithms, the static single assignment (SSA) form showed to be useful for that. Frontends which translate this intermediate language are already available for Java, JavaScript and Bytecode.
In SSA, per definition, each variable may only be defined once. After branches, i.e. `if...else`, we would need to decide which of the definitions is used later in the programme. This is decided by so-called ϕ-functions. Besides inserting those ϕ-functions, we have to rename all variables to have a given code in SSA-form.
This and further explanations will be shown in the next chapter.

As such a frontend is missing for Android bytecode, it appears practical to create one for WALA. This work takes the first important steps needed to create such a frontend.

We will extract control flow information from the bytecode (see section 2.2 and 4.4), compute the dominance relation and frontier (see section 2.3 and 4.5), and finally we will compute the SSA-form (see section 2.4 and 4.6).

2. Theoretical Fundamentals

This chapter introduces the basics needed to create the SSA-form from Android bytecode.

Let us introduce an example for this entire paper:

```
public static int boing() {
    int i = 1;
    int j = 1;
    int k = 0;

    while (k < 100) {
        if (j < 20) {
            j = i;
            k = k + 1;
        } else {
            j = k;
            k = k + 2;
        }
    }
    return j;
}
```

Listing 2.1: Example code of a simple method

This example is very suitable for several reasons. Obviously it is very short. However it contains two important control flow statements. We have a loop and a choice statement. In those two statements we have redefinitions of variables, which will be later eliminated by converting it to the SSA-form.

2.1. Bytecode

Source code is a human-readable programme representation which must be translated to machine readable code in order to be executed. For example a C/C++ programme is compiled into architecture dependent instructions and can be run natively.

The Java Compiler uses a different approach: It will not create a architecture dependent executable file. It creates Java runnable files, i.e. Java SE and Dalvik "class"-files in form of bytecode, which are run in a virtual machine (VM). If you compare Bytecode with Assembler Language it appears to be pretty similar.

As mentioned before, Android uses a special form of bytecode called Dalvik Bytecode: The most significant difference is that Dalvik is register-based whilst Java bytecode is stack-based. Hence instructions are slightly different as well.
Both have in common that they are compiled of the same code and most bytecode instructions are similar.

2.1.1. Java SE [Jav]

The Java SE virtual machine is stack-based, i.e. instructions use a stack to access and store values. Every thread has its own stack which stores frames. Frames (see figure 2.1) are created on invocation of a method and keep all important data for this method, such as operands, local variables and the return point.

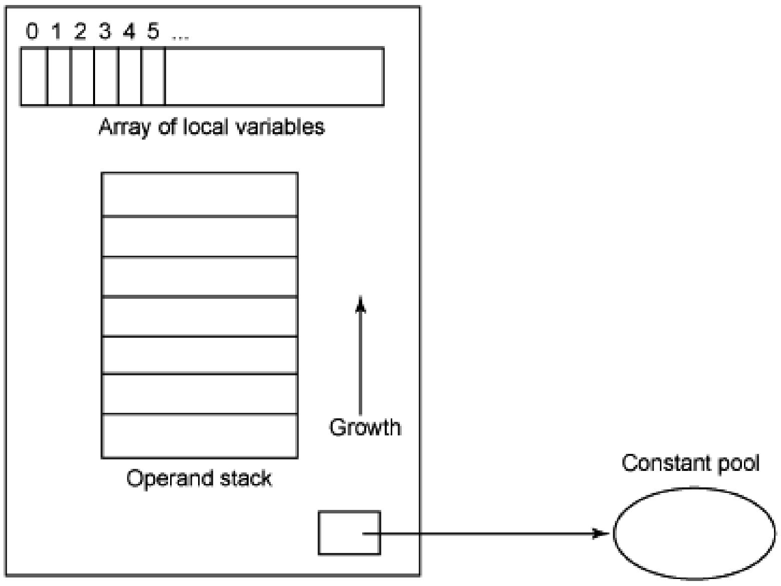


Figure 2.1.: A conceptual frame of a Java VM [Jav]

Now again, let us have a look at the Java example in listing 2.1. There is a loop which runs as long as `k` is lower than `100`. Inside the loop there are two branches, which redefine j and k.
In addition we look at the Bytecode result (listing 2.2). First thing we notice, as already mentioned above, Bytecode uses a stack for all instructions.
Lines 0 to 5 load and store the variables i, j, and k and their initial values. To perform the **`while`** loop 100 is pushed onto the stack in line 7. Afterwards `if_icmpge` checks whether the top item in stack is greater or equal the second top item. If so it jumps to the end of this method by using `goto` statements, else it will proceed with instruction 12.

```
  public static int boing();
Code:
   0: iconst_1
   1: istore_0
   2: iconst_1
   3: istore_1
   4: iconst_0
   5: istore_2
   6: iload_2
   7: bipush        100
   9: if_icmpge     36
  12: iload_1
  13: bipush        20
  15: if_icmpge     27
  18: iload_0
  19: istore_1
  20: iload_2
  21: iconst_1
  22: iadd
  23: istore_2
  24: goto          6
  27: iload_2
  28: istore_1
  29: iload_2
  30: iconst_2
  31: iadd
  32: istore_2
  33: goto          6
  36: iload_1
  37: ireturn
```

Listing 2.2: Bytecode of listing 2.1

Although Bytecode has many more features we will skip those and have a look at the Dalvik Bytecode, which is the base for the further work.

2.1.2. Dalvik [Dal]

Dalvik is the VM of Google's Android operating system. Unlike the normal Java VM, Dalvik has a register-based architecture, i.e. contrary to the stack-based Java SE VM it accesses and stores values by using registers. The code from listing 2.1 could look like this in Dalvik:

```
.method public static boing()I
.registers 4
```

```
    .prologue
    .line 14
    const/4 v0, 0x1

    .line 15
    .local v0, i:I
    const/4 v1, 0x1

    .line 16
    .local v1, j:I
    const/4 v2, 0x0

    .line 18
    .local v2, k:I
    :goto_3
    const/16 v3, 0x64

    if-ge v2, v3, :cond_13

    .line 19
    const/16 v3, 0x14

    if-ge v1, v3, :cond_f

    .line 20
    move v1, v0

    .line 21
    add-int/lit8 v2, v2, 0x1

    goto :goto_3

    .line 23
    :cond_f
    move v1, v2

    .line 24
    add-int/lit8 v2, v2, 0x2

    goto :goto_3

    .line 27
    :cond_13
    return v1
.end method
```

Listing 2.3: Dalvik bytecode of listing 2.1

Again the variables and their initial values are loaded. Contrary to normal bytecode we cannot put a constant, i.e. 100 from the `while`'s condition, into an instruction but have to put it into a register at first. We see that instructions, like the `if-ge` instruction, look more compact. By having registers it is easier to recognize a variables origin, for instance $j = i$ is in bytecode two instructions: `iload_0; istore_1`, Dalvik just needs one instruction `move v1, v0`. Again the flow is controlled by `if` and `goto`.
Contrary to normal Bytecode the frames have fixed size upon creation. The amount of registers is specified by the method, each register containing 32 bits. Further advantages are the easily readable syntax, destination-then-sources.

In conclusion, having the same origin code, Dalvik bytecode is obviously easier to read. On the other hand we cannot use constants directly, which we can in Java bytecode; hence converting from Dalvik bytecode to Java bytecode will create larger code, as there are more operations needed.
However having a register-based bytecode makes it easy to find variables, whilst in a stack-machine we had to check the stack's push and pop operations. Additionally, as SSA is register-based, there is no translation necessary from stack-code to register-code.

To translate bytecode to the SSA-form we have to rename the variables and place ϕ-functions in the right place. This requires the dominators and the dominance frontier, which are computed by using the flow information of a control flow graph (CFG).

2.2. Control Flow Graph

A control flow graph (CFG) is a graph which represents all paths that might be traversed through a programme during its execution. Each vertex represents an instruction, the control flow is being represented by edges. Per definition there has to be one $START$-node and one $EXIT$-node.
Figure 2.2 shows the CFG of listing 2.1. This graph is the basis for the algorithm we will later use to create the SSA-form of Bytecode.

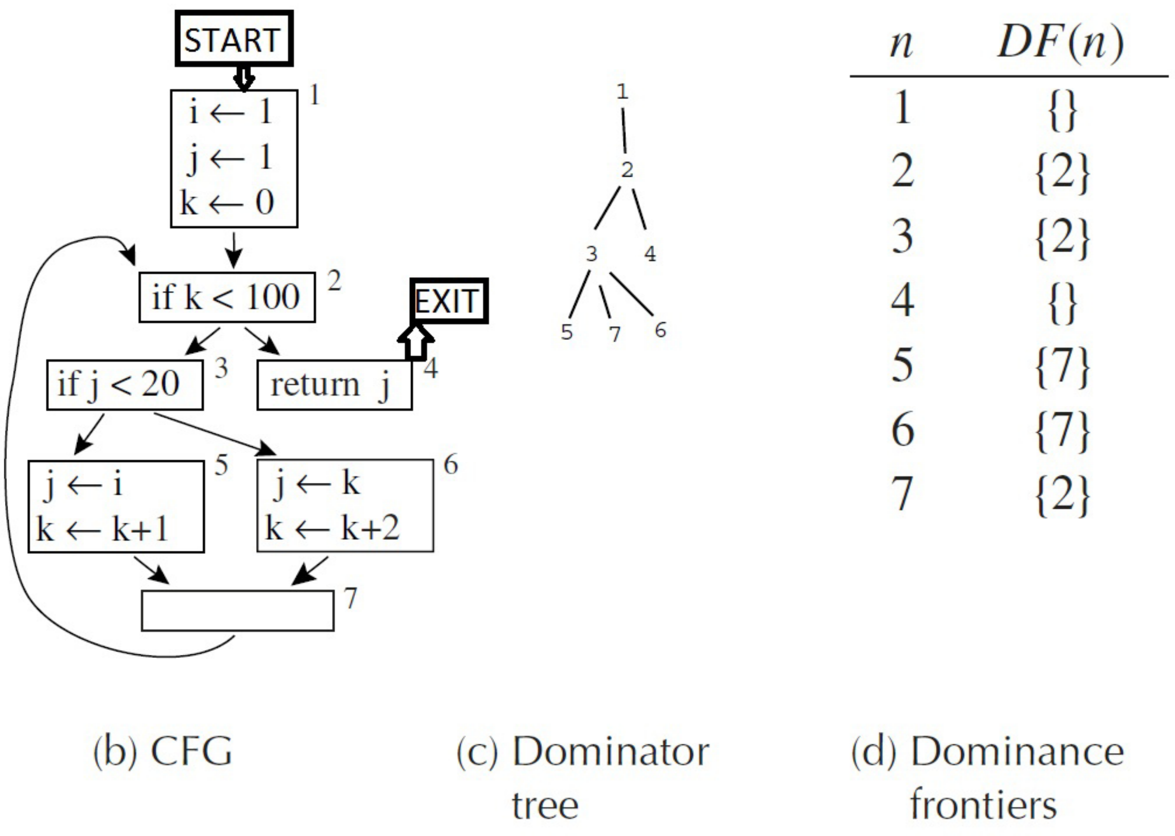
A further possible application for a CFG is reachability analysis, which target it is to find unreachable code, which then can be safely removed; it is possible to find some infinite loops by detecting an unreachable exit vertex as well.

2.3. Dominators [Wika]

From the CFG we can extract even more information, that we will need. Such as the dominators, immediate dominators, the dominance frontier and subsequently the dominance tree. Before a certain instruction is executed its dominator has to be executed. To find the variable which has to be renamed, we can use this relation by looking where the variable is defined and which instructions are dominated by that instruction.

- If a vertex d dominates a vertex n, each path from the start node to n must go through d. d is the **dominator** of n.
- A vertex d **strictly dominates** a vertex n if d dominates n and $d \neq n$.
- The **immediate dominator** (IDOM) of a vertex n is the unique vertex that strictly dominates n but does not strictly dominate any other vertex that strictly dominates n.
- The **dominance frontier** (DF) of a node d is a set of all vertices n such that d dominates an immediate predecessor of n.
- Node x is an **ancestor** of y if there is a path $x \rightarrow y$ of tree edges, and is a **proper ancestor** if that path is nonempty

In other words a dominates b means: in order to reach a node b we must pass a. The



n	$DF(n)$
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

Figure 2.2.: Programme 2.1 as a CFG, DOM and DF [App98]

dominator tree is the graph representation of the dominator relation. The dominator tree for our example code is shown in figure 2.2. For instance to reach node 3 we have to pass node 2. So 2 dominates 3 and this makes an edge from 2 to 3 in the dominator tree.

We will use the Lengauer-Tarjan-algorithm for the dominator calculation. It has a runtime complexity of $O(|N| + |E|)$ by exploiting certain properties of the dominance relation:

Theorem 1 ***Semidominator Theorem*** *[LEJ79]:*
To find the semidominator of a node n consider all predecessors v of n in the CFG.

- *If v is a proper ancestor of n in the spanning tree (so $dfnum(v) < dfnum(n)$) then v is a candidate for $semi(n)$.*
- *If v is a nonancestor of n (so $dfnum(v) > dfnum(n)$), then for each u that is an ancestor of v (or $u = v$), let $semi(u)$ be a candidate for $semi(n)$.*

By having the semidominators we can compute the dominators from them. With s being the semidominator of n. If there is a path above s which bypasses s then s is not a dominator.
If there is a node y between s and n with the smallest-numbered semidominator, and $semi(y)$ is a proper ancestor of s, so the immediate dominator of y also immediate dominates n.
We can put that to the dominator theorem.

Theorem 2 ***Dominator Theorem*** *[LEJ79]:*

$$idom(n) = \begin{cases} semi(n) & \text{if } semi(y) = semi(n) \\ idom(y) & \text{if } semi(y) \neq semi(n) \end{cases}$$

2.3.1. Dominance Frontier

Additionally we need the dominance frontier (DF) in order to find instructions which must be visited so the following code can be executed.

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.

In other words, the DF of a node is a set of nodes, which are just not any more dominated by that certain node. If b is in the DF of a, there is another path to b, so a does not have to be passed.

In figure 2.2 we see in (d) the dominance frontier of the CFG (b). For example the dominance frontier of node 2 is $\{2\}$. This means all following nodes are dominated by 2, obviously. So the dominance frontier is 2 itself. To have it more vivid, look at figure 2.3.

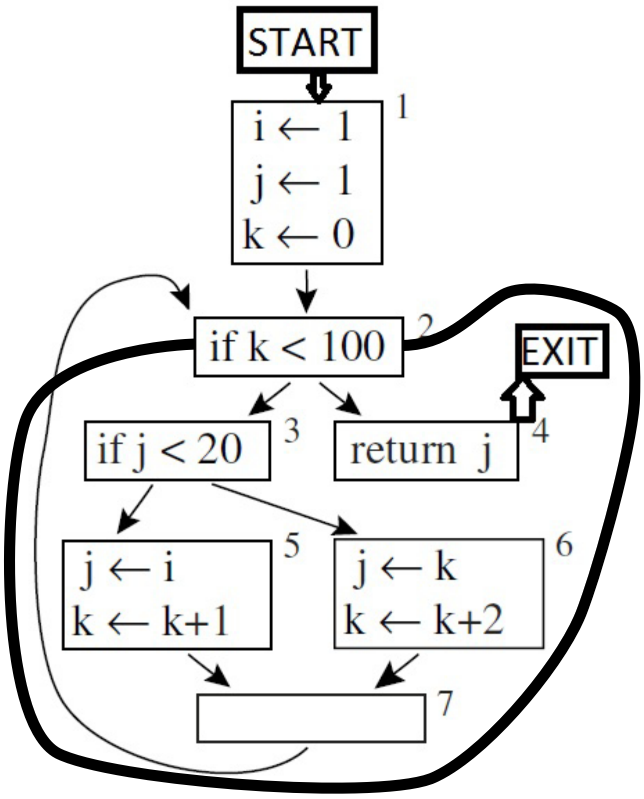


Figure 2.3.: Dominance frontier of node 2 (thick line).

The DF of each node can be calculated with two sets:
$DF_{local}[n]$: The successors of n that are not strictly dominated by n
$DF_{up}[n]$: Nodes in the dominance frontier of n that are not strictly dominated by n's immediate dominator.
We compute the dominance frontier of n from both of them:

$$DF[n] = DF_{local}[n] \cup \bigcup_{c \in children[n]} DF_{up}[c]$$

We will calculate and use the dominators, etc. for computing the static single assignment form in the final solution.

2.4. Static Single Assignment form (SSA-form) [Wikb]

SSA-form is used by programme analysis and optimization tools. Due to its structure it is easier to find dead code and constants and several other optimizations. The major advantage is that each variable may only be defined once and no variable's value is changed, which greatly simplifies the reaching definition analysis.

For a programme to be in **Static Single Assignment form** it requires variables to be assigned exactly once. To achieve this, every variable which is assigned more than once has to be renamed, and at certain joints all versions of a variable are "merged" by so-called ϕ-functions and can be used subsequently again as the origin variable.

But when and where do we have to insert ϕ-functions? Whenever we use a variable which could come from more than one definition site, we insert such a node. In principle in the CFG whenever a node joins two branches we could add a ϕ-node there. Although there are examples where a ϕ-function is not necessary at this point. For instance having two branches which do not change a certain variable, this variable obviously does not need a ϕ-function.

Theorem 3 ***Path-convergence criterion*** *[App98]*
There should be a ϕ-function for variable a at node z of the flow graph exactly when all of the following are true:

1. *There is a block x containing a definition of a.*
2. *There is a block y (with $y \neq x$) containing a definition of a.*
3. *There is a non-empty path P_{xz} of edges from x to z.*
4. *There is a non-empty path P_{yz} of edges from y to z.*
5. *Paths P_{xz} and P_{yz} do not have any node in common other than z.*
6. *The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.*

A ϕ-function itself defines a variable so we have to iterate through blocks:

Iterated path-convergence criterion

```
    while there are nodes $x$, $y$, $z$ satisfying condition 1-5
        and $z$ does not contain a $\phi$-function for $a$
      do insert $a \leftarrow \phi(a, a, \ldots, a)$ at node $Z$
```

Listing 2.4: Iterated path-convergence criterion [App98]

While inserting the function, we need to know how many parameters the ϕ-function has. As mentioned above, we are looking at blocks not the instruction nodes. So we just look how many predecessors a block node has, this is the amount of parameters. At this point we will name each parameter as well as the left side just like the origin name. We will change all names later.

To be able to use the dominance frontier to calculate the ϕ-nodes we need the following property:

Theorem 4 ***Dominance property of SSA form*** *[App98]*
An essential property of static single-assignment form is that definitions dominate uses; more specifically,

1. *If x is the ith argument of a ϕ-function in block n, then the definition of x dominates the ith predecessor of n.*

2. *If x is used in a non-ϕ statement in block n, then the definition of x dominates node n.*

Combining this property with the dominance frontier brings us this criterion:

Theorem 5 ***Dominance Frontier Criterion*** *[App98]*
Whenever node x contains a definition of some variable a, then any node z in the dominance frontier of x needs a ϕ-function for a.

Which leads to:

Theorem 6 ***Iterated Dominance Frontier*** *[App98]*
Since a ϕ-function itself is a kind of definition, we must iterate the dominance frontier criterion until there are no nodes that need ϕ-functions.

So before we can create the SSA form we need a suitable data structure to walk the code, we will later use a control flow graph for that. Now we find all definitions sites of all variables. Having those definition sites, we check the path-convergence criterion and insert the ϕ-nodes. Afterwards we can change all variable names accordingly, so that each name is only assigned once.

Let us look at figure 2.2 again. We see that in the CFG node 2 needs variable `j`, which is defined in 5, 6 and 1. In figure 2.4 we see on the left hand side the result when we insert the ϕ functions. Afterwards all variables are renamed by walking the dominator tree, which we can see on the right hand side.

The steps needed to create SSA-form are

1. Convert code (in this case Dalvik bytecode) to a CFG.

2. Create dominator tree out of CFG.

3. Create dominance frontier.

4. Insert ϕ-functions at the DF of the nodes.

5. Rename all variables with help of the dominator tree.

This can be solved by the already-mentioned **dominance frontier**, which we will see again in chapter 4.

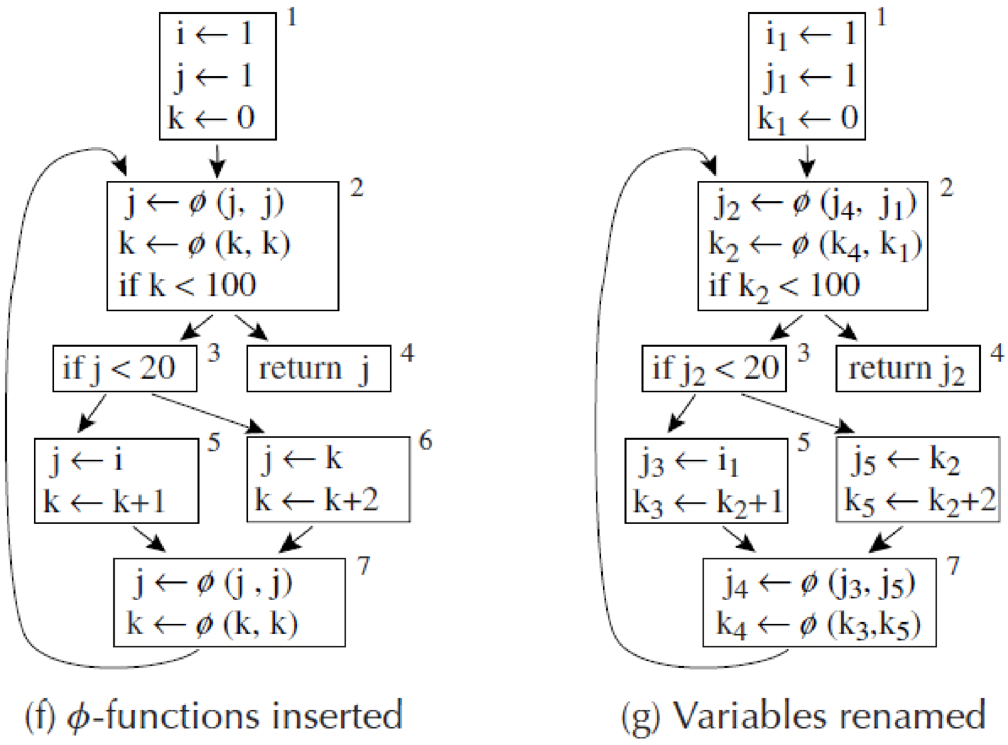


Figure 2.4.: Listing 2.1 with ϕ-nodes and renamed variables.

3. Planning

Our goal was to develop a scientific prototype. Even science prototypes need thorough planning so the final software is easy to understand and maintain. As the data structures are not very complex and the algorithms do not involve large object-orientation, we will not go into UML-design.

3.1. Existing System

Before we will define our requirements we look at the given system.
The chair of programming paradigms at Karlsruher Institute of Technology already has a similar programme to convert normal Java code to SSA form. Then it is analysed with *Joana*.
WALA uses a register-based SSA-form, where instructions are similar to Java bytecode. WALA was invented by IBM for the purpose of programme analysis. Besides many other projects, the chair's own project *Joana*, which provides information flow control, is based on WALA.

3.2. Requirements

There already is a programme which does what we want for Java bytecode. Unfortunately, as you have seen before, Dalvik bytecode is slightly different to normal bytecode. So the main requirement of this programme is to convert Dalvik bytecode to SSA-form, so a framework can be developed and finally be connected to *Joana*.

3.2.1. Specifications

There are these high priority requirements to the programme:

1. Parse `dex`-files to SMALI-code [1]
2. Data structure with the CFG
3. Data structure with the dominators and dominance frontier
4. Data structure with the SSA graph

[1]see section 4.3

Additionally we have these low priority requirements:

1. Dominator computation should be based on Lengauer-Tarjan's algorithm. [Cyt+91]
2. ϕ-function computation should be based on [Cyt+91].
3. All provided test-cases should succeed on testing. [2]
4. It should be possible to reintegrate the code into the SMALI project.

Those criteria should not be met:

1. Creating an industry product rather than a scientific prototype.

3.3. Planned Target

Target group and place is the chair for programming paradigms at Karlsruhe Institute of Technology. Additionally this project shall be provided to the open source community of the SMALI project, which was used for the first part of the solution.

The main usage of the programme will be to convert Android bytecode into SSA-form, to create a basis for development of a frontend for WALA. This will be integrated into a programme analysis software, and shall only represent a scientific prototype.

[2]Additionally to the test-classes in the appendix, the project "MoWiDi" is to be tested, which is an Android application which enables the Android device to be used as a hard disk drive via WLAN. It was created during the summer term of 2010 at the chair for operating systems of the Karlsruher Institute for Technology. `http://os.ibds.kit.edu`

4. Implementation

Knowing all the basics, we can have a look at the solution. As a basis we have the SMALI project. It provides methods to obtain the control flow of Dalvik bytecode. With this information we apply the Lengauer-Tarjan algorithm to calculate the dominators. Eventually the SSA-form is acquired by inserting ϕ-functions and rename the variables according to [Cyt+91].

4.1. Classes and Changes

4.1.1. New Classes

In total there were 23 new classes necessary of which 6 classes were just needed for testing. Without test cases, comments and JavaDoc it took 1719 lines of code to perform the task, SMALI etc. not included.

`org.jf.dexlib.Code.Analysis.graphs`

This package includes all classes needed for graph representation and computation of a CFG. It has 854 lines of code (excluding comments, JavaDoc and tests). See table 4.1.

`edu.kit.pp.ssa.dom`

This package includes all classes needed for calculating the dominators and the dominance frontier, as well as graph representation. It has 609 lines of code (excluding comments, JavaDoc and tests). See table 4.2.

`edu.kit.pp.ssa.ssa`

This package includes all classes needed to calculate the SSA-form from the dominators. It has 256 lines of code (excluding comments, JavaDoc and tests). See table 4.3.

4.1.2. Changed Existing Classes

`org.jf.baksmali.Adaptors`

This package contains adaptors to access classes and their items, i.e. methods and fields. See table 4.4.

Class	Purpose
`AbstractNode`	An abstract node.
`CFG`	Represent and build a CFG graph.
`CFGEdge`	Represents a CFG edge.
`CFGEdgeFactory`	Used to create edges.
`Edge`	An interface for edges.
`EntryNode`	Entry node of the CFG.
`ExitNode`	Exit node of the CFG.
`GraphDumper`	Calculates CFG, DOM, DF and SSA and prints them.
`GraphWalker`	Provides ways to traverse a graph.
`InstructionNode`	A node which has an instruction inside.
`Node`	An interface for nodes.
`WriteGraphToDot`	Writes a graph to a graphviz `dot`-file.
`WriteMapToTxt`	Write a map to a `txt`-file, needed for DF printing.

Table 4.1.: New classes in package `org.jf.dexlib.Code.Analysis.graphs`.

Class	Purpose
`DomEdge`	Represents a dominator tree edge.
`DomTree`	Represents the dominator tree.
`Dominators`	Used to compute the dominators and to get the tree and DF.
`DominatorsTest`	Test cases for the dominators.
`MyEdge`	Generic edge for test cases.
`MyEdgeFactory`	Generic edge factory for test cases.
`Node`	A node for test cases.
`Tree`	A tree for test cases.

Table 4.2.: New classes in package `edu.kit.pp.ssa.dom`.

Class	Purpose
`SSA`	Used to compute and get the SSA CFG.
`SSABlockNode`	A block of SSA CFG nodes.
`SSABlockGraph`	A SSA CFG with blocks.
`SSAEdge`	Represents a SSA CFG edge.
`SSAGraph`	represents a SSA CFG with single nodes.
`PHI`	Represents a ϕ-function.
`PHINode`	A node containing a ϕ-function
`MySSATest`	Unfinished test case for `SSA`

Table 4.3.: New classes in package `edu.kit.pp.ssa.ssa`.

Class	Purpose
`ClassDefinition`	Added method to dump graphs of a class. `dumpGraphs`

Table 4.4.: Changed classes in `org.jf.baksmali.Adaptors`.

`org.jf.baksmali`

This package includes the class which contains the `main`-method, as well as other classes which are related to that. See table 4.5.

Class	Purpose
`baksmali`	Added code so that, if wanted, a class' graph is dumped.

Table 4.5.: Changed classes in `org.jf.baksmali`.

4.2. Workflow of the Programme

In this section we will have a rough look at the workflow of the programme. A flow diagram (figure 4.1) explains the work of the programme and the algorithm.
First thing you will notice when looking at figure 4.1 is that everything is executed by `GraphDumper`. To start with all calculations we need some `AnalyzedInstructions` to calculate the CFG from. We get those by using the SMALI package and parse a DEX file. Now `GraphDumper` creates a CFG, which builds itself by using the instructions. With this CFG we can create a `Dominators` object.
Finally the SSA graph can be calculated. If selected to do so by command line arguments, the graph dumper will put all or some results to .dot files (Graphviz format).

The entire solution is written in Java. For graphs and trees the `jgrapht` library is included, as well as the SMALI packages, of course. The major classes use factory methods to create objects instead of public constructors.

Let us have a look at the interface and usage of the main classes.

CFG:
A `CFG` already is a `DirectedPseudograph` and can be used as such. To build it you must call either
`CFG.build(List<AnalyzedInstruction> instructions, String name)` or
`CFG.build(List<AnalyzedInstruction> instructions, String name, **boolean** includeExc)`
with `instructions` being a list of instructions of a method you get by using SMALI disassembler and `name` being an identifier, usually the methods name. If you want

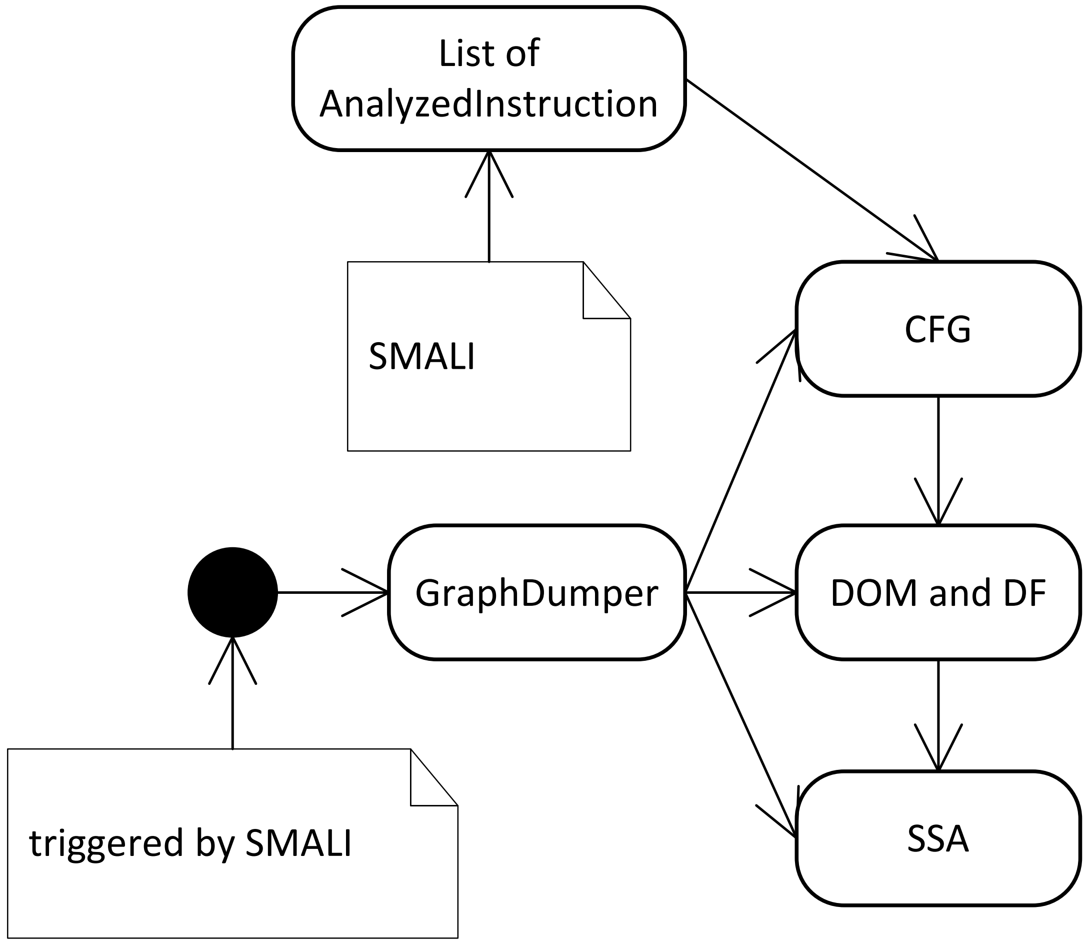


Figure 4.1.: Flow of the programme

to include exceptions in the CFG, you must use the second method call and specify `includeExc == true`.
Although all edges in the CFG will prevail the algorithm, it is important to know that variable names in the nodes **will** change, if you need the origin names you will have to perform a deep copy of the CFG, which is not implemented.

DOM:
With having the CFG we can make a call to `Dominators`. It needs both the `CFG` object and the start node. You will have to call
`Dominators.compute(DirectedGraph<Y, Z> graph, Y entry)`
`graph` should just be the `CFG` object and `entry` the first node of the programme. The dominators and DF are calculated right away.

Getters for obtaining the dominator tree and the dominance frontier are provided.

SSA:
Usage of `SSA` is similar. You just call
`SSA.compute(DirectedGraph<Y, Z> graph, Dominators<Y, Z> dom)`
with the `CFG` object and the `Dominators` object and the CDG is computed, i.e. ϕ-nodes are placed and variables are renamed. Again getter-functions are provided to obtain the SSA graph.

Let us now see how everything works in particular. To do that, let us use a recurring example you have already seen before in figure 2.2 and listing 2.1.
You will notice later, that the programme's result is slightly different to the figure copied from [App98]; the **`static`** keyword makes no difference to neither the algorithm nor the result.

4.3. Introducing SMALI [Sma]

Before we can use an algorithm to calculate the SSA-form, we have to read in bytecode. For that we use the SMALI project, which provides an assembler and a disassembler for Dalvik bytecode.

```
.class public LHelloWorld;

.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System;
        ->out:Ljava/io/PrintStream;

    const-string v1, "Hello World!"

    invoke-virtual {v0, v1}, Ljava/io/PrintStream;
        ->println(Ljava/lang/String;)V

    return-void
.end method
```

Listing 4.1: SMALI-Bytecode

An example for SMALI-bytecode can be seen in listing 4.1 and was already used in section 2.1.2 as well.

Let us have a look at the structure of the SMALI project, especially those classes we use.

Before creating a CFG we need to extract the instructions out of the dex file. Assuming we have initialised an `DexFile` object already, we need to take out the class definitions, which are represented by `ClassDefItem`. Each of those class definitions contain, of course, method definitions. Even "empty" classes have at least a hidden `_init_()V`-Method which is similar to a constructor.
With the `ClassDataItem` it is possible to extract each method and retrieve its instructions. Those will be used by the `CFG` class to build a tree.

```
// walk all class definitions
for (ClassDefItem clsDef : dexFile.ClassDefsSection.getItems()) {
    // walk all methods in a class
    ClassDataItem clsData = clsDef.getClassData();
    for (EncodedMethod em : clsData.getDirectMethods()) {
        MethodAnalyzer analyze = new MethodAnalyzer(em, false);
        analyze.analyze();
        List<AnalyzedInstruction> instructions = analyze.
           getInstructions();
        // here: use instructions...
        // in our case build CFG, create dominators and DF
        // and create SSA graph out of that.
    }
}
```

Listing 4.2: Example use of SMALI

While building the CFG we have to find out successors, predecessors and registers used and set by an instruction. Those are provided by methods of `AnalzyedInstruction`.

Large disadvantages of SMALI, which raised during implementation, are the hardly documented methods and classes, as well as weird method calls or design flaws, e.g. highly encapsulated objects without more functionality.
Especially getting the used registers of an instruction needs finding out which type of instruction it is and then the `AnalyzedInstruction` has to be cast to a `ThreeRegisterInstruction` etc.
Only now you can get the variable names of the registers. A very big error potential is that SMALI gives you just the registers. It has to be obtained whether the first of the registers is assigned, or just used.
To make it more clear: `a = b + c` and `invoke-direct(a, b, c)` return the same three registers, although only the first example really sets a register.
This is required for the first step: To find all definitions of a variable.

4.4. CFG

This part is very easy. After disassembling the dex-file we already have a data structure of instructions making it possible to retrieve successors, being almost an easy-to-use graph. The `CFG`'s build method gets the `AnalyzedInstruction` list and starts building a graph we can work with more easily than with the origin data structure.

First thing which has to be done is adding all vertices to the new, yet empty, tree. To enhance usability the `AnalyzedInstructions` we got from SMALI will be wrapped in a `InstructionNode` to make the, for us important, methods and attributes quicker accessible. By using a depth-first-search algorithm all edges are added to the new tree. However we must keep in mind there could be two edges from one vertex to another as exceptions are having also edges.

After all vertices and flows are put into the graph we strip all unreachable nodes. They bring no information to neither the graph nor the analysis.

The result of listing 2.1 converted to a CFG can be seen in figure 4.2. You can see that Java needs registers also for constants like the 100 in the **`while`**'s condition. As well as the two **`if`** branches are not joined for one **`goto`** rather than two to join back at the **`while`**'s head.
Although no exception can occur in this example, exception-flows would be names `CF_EX` in the graph.

4.5. DOM and DF

With the CFG we can compute the dominators and with that the dominance frontier. Both will later come in handy to compute the ϕ-nodes faster than by just fulfilling the criterion for them.

4.5.1. Dominator Tree

Although there are algorithms for computing the dominators they do not scale very good. As it is the best known algorithm to calculate dominators and is fast, we will use Lengauer-Tarjan algorithm.
This algorithm just follows the depth-first spanning tree. Each node there gets a `dfnum`, which is the first number a node was encountered by depth-first search.
If you have a node n and its immediate dominator d, d must be an ancestor of n in the spanning tree. Therefore we have $dfnum(d) < dfnum(n)$.

In figure 4.3 we see this, as well as the fact that an ancestor x does not dominate n. This means there must be a second path around x to reach n. The $dfnum$s of the second path are higher than the one from n so they are not ancestors.

Lengauer-Tarjan Algorithm

Having the two needed theorems we arrived at the most complex algorithm of this work: the Lengauer-Tarjan algorithm for computing the dominators: (listing 4.3)

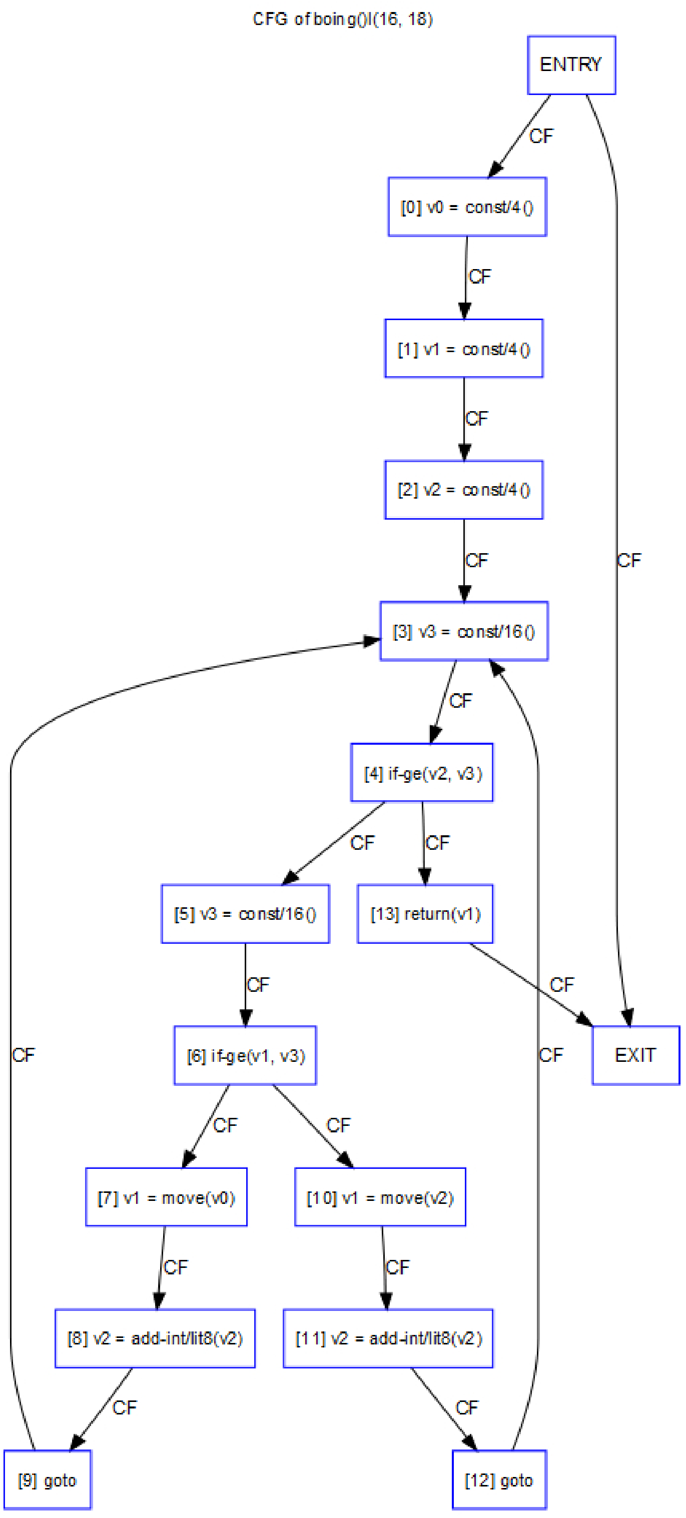


Figure 4.2.: listing 2.1's CFG

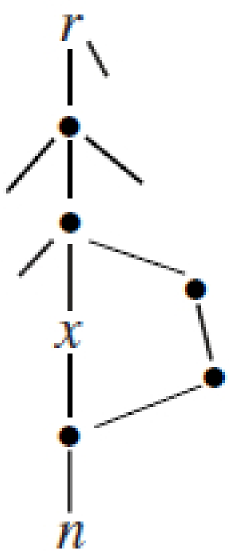


Figure 4.3.: Dominators and spanning-tree paths. [App98]

```
DFS(node $p$, node$n$) =
    if $dfnum[n] = 0$
        $dfnum[n] \leftarrow N; vertex[N] \leftarrow n; parent[n] \leftarrow p$
        $N \leftarrow N + 1$
        for each successor $w$ of $n$
            DFS($n$, $w$)

Link(node $p$, node $n$) =
    add edge $p \rightarrow n$ to spanning forest implied by ancestor array

AncestorWithLowestSemi(node $n$) =
    in the forest, find the nonroot ancestor of $n$ that has the
       lowest-numbered semindominator

Dominators() =
    $N \leftarrow 0; \forall n : bucket[n] \leftarrow \{\}$
    $\forall n : dfnum[n] \leftarrow 0, semi[n] \leftarrow ancestor[n] \leftarrow idom[n] \leftarrow samedom[n] \leftarrow$ none
    DFS(none, $r$)
    for $i \leftarrow N - 1$ downto 1
        $n \leftarrow vertex[i]; p \leftarrow parent[n]; s \leftarrow p$
        // calculate semidominator of n (semidominator theorem)
        for each predecessor $v$ of $n$
            if $dfnum[v] \leq dfnum[n]$
                $s' \leftarrow v$
            else
            $s' \leftarrow semi$[AncestorWithLowestSemi($v$)]
        if $dfnum[s'] < dfnum[s]$
            $s \leftarrow s'$
    $semi[n] \leftarrow s$
    $bucket[s] \leftarrow bucket[s] \cup \{n\}$
    Link($p$, $n$)
    // calculate dominators of v
    for each $v$ in $bucket[p]$
        $y \leftarrow$ AncestorWithLowestSemi($v$)
        if $semi[y] = semi[v]$
            $idom[v] \leftarrow y$
        else
            $samedom[v] \leftarrow y$
    $bucket[p] \leftarrow \{\}$
// calculate deferred dominators
for $i \leftarrow 1$ to $N - 1$
    $n \leftarrow vertex[i]$
    if $samedom[n] \neq$ none
        $idom[n] \leftarrow idom[samedom[n]]$
```

Listing 4.3: Lengauer-Tarjan algorithm for computing dominators. [App98]

The algorithm uses depth-first search to compute all *dfnum*s. We visit the nodes from highest *dfnum* to lowest and compute the semidominators and dominators.

The algorithm consists of two outer loops. The first loop computes all the semidominators of n and link the path from the parent of n to n. After that the dominators according to the first case of the dominator theorem are calculated. If this would not fit, they are deferred to be calculated in the second outer loop. The second loop just calculates the dominators according to the second case in the dominator theorem. (see section 2.3

Looking at the helper methods, the DFS should be clear. While walking the depth-first spanning tree we calculate the *dfnum*s and initialize the needed arrays.
Although `Link` and `AncestorWithLowestSemi` have potential for optimisation and better runtime, in this solution it was kept the $O(N)$ way, however $O(logN)$ is possible by using path compression. These methods might be called several times, we let *ancestor* point to an ancestor which is above the parent.

The dominators calculation was kept closely to the algorithm. Instead of arrays it uses `HashMaps`. Nevertheless the tree has to be rebuild upon the end of the calculations. This is as the algorithm was not implemented in a object oriented way, but we want a nice tree afterwards. We are just walking for that all nodes and adding edges to the dominators. This takes just $O(N)$ time and is less or equal than the other time bounds.

The result of this computation can be seen in figure 4.4.

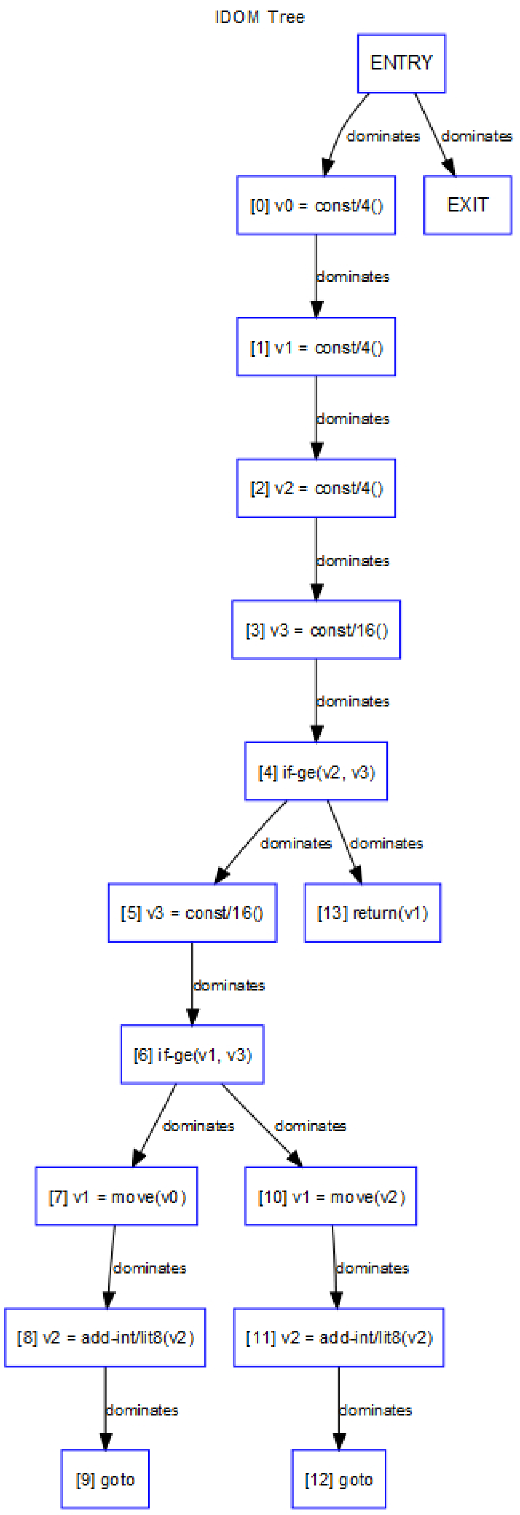


Figure 4.4.: listing 2.1's dominator tree

4.5.2. The Dominance Frontier

We could just iterate through all nodes and find the positions for a ϕ-function by checking the criteria (see theorem 3). However you would need to examine every path from one node to another etc.
An efficient way is to use the dominance frontier instead.

We can compute $DF_{local}[n]$ a little easier than by definition. By just using the immediate dominators we have $DF_{local}[n]$ = the set of those successors of n whose immediate dominator is not n.

Let us look at the algorithm in listing 4.4. We just call it on the root of the dominator tree (which ought to be the start node of the CFG). It computes $DF_{local}[n]$ by examining

```
computeDF[$n$] =
    $S \leftarrow \{\}$
    // this loop computes DF local of n
    for each node $y$ in $succ[n]$
        if $idom(y) \neq n$
            $S \leftarrow S \cup \{y\}$
    // this loop computes DF up of c
    for each child $c$ of $n$ in the dominator tree
        computeDF[$c$]
        for each element $w$ of $DF[c]$
            if $n$ does not dominate $w$, or if $n = w$
                $S \leftarrow S \cup \{w\}$
    $DF[n] \leftarrow S$
```

Listing 4.4: Compute DF of Node n. [App98]

the successors of n, then combines $DF_{local}[n]$ and for each child c: $DF_{up}[c]$. [App98]
This algorithm runs prictically in linear time.

The real implementation in this programme is pretty similar to the pseudo code. The only difference is, we need to have a set to check which nodes were visited, as tests showed that somehow the programme could enter an infinite recursion in certain conditions.

The result of this calculation is just a map with all dominance frontiers of the nodes. In table 4.6 you can see the result of the recurring example. By comparing it with figure 4.4 you can just confirm the criterion of the DF is met.

Node	DF(Node)
[0]	EXIT
[1]	EXIT
[2]	EXIT
[3]	[3], EXIT
[4]	[3], EXIT
[5]	[3]
[6]	[3]
ENTRY	
[7]	[3]
[8]	[3]
[9]	[3]
[10]	[3]
[11]	[3]
[12]	[3]
[13]	EXIT
EXIT	

Table 4.6.: DF of recurring example.

4.6. Placing and Renaming of ϕ-Nodes

To create the SSA-form and its graph we first need to **place** the ϕ-nodes and then **rename** all variables.
By using the dominators and dominance frontier those tasks can be performed more efficient than by just walking all nodes and checking the criteria.

4.6.1. Placing ϕ-Functions

With the given criteria from section 2.4 we get the algorithm in listing 4.5. First thing we have to do before running the algorithm is gathering all variables and populate A_{orig}. It keeps a set of all variables which are defined in a certain node. The $A_\phi[a]$ will finally contain a set of block nodes which have to have a ϕ-function.

After having all variables and a populated A_{orig} we populate the `defsites`-set. It contains the nodes in which a certain variable is defined. Here we use a `HashMap` to perform fast adding and reading.
Eventually we are running a loop for all variables (see theorem 6).

The representation of W must allow quick testing of membership and quick extraction ([App98]) as well, which is provided by using a `HashMap`. Using **`boolean`**-flags instead is suggested, but as we are trying to remain as object oriented as possible a `Map` just seems to fit the requirements without having to implement a lot yourself.

```
Place-$\phi$-Functions =
    for each node $n$
        for each variable $a$ in $A_{orig}[n]$
            $defsites[a] \leftarrow defsites[a] \cup n$
    for each variable $a$
        $W \leftarrow defsites[a]$
        while $W$ not emtpy
            remove some node $n$ from $W$
            for each $y$ in $DF[n]$
                if $a \notin A_{\phi}[y]$
                    insert the statement $a \leftarrow \phi(a, a, \ldots, a)$ at the top of
                        block y, where the $\phi$-function has as many
                       arguments as $y$ has predecessors
                   $A_{\phi}[Y] \leftarrow A_{\phi}[Y] \cup a$
                   if $a \notin A_{orig}[y]$
                       $W \leftarrow W \cup y$
```

Listing 4.5: Place ϕ-functions. [App98]

Each ϕ-function is inserted both in the SSA graph and the SSA block graph. This is necessary as we need to know if a block holds a ϕ-function. Just adding it to the SSA graph would force us to build the SSA block graph again, which so can be avoided.
Just adding it to the SSA block graph would destroy the invariant that tree and block tree shall represent the same SSA graph. Unfortunately to have a easy solution for adding the ϕ-functions are added to the top of a block node, which is just a different sequence than in the tree. Nevertheless this will not destroy the invariant as sequence of execution of unrelated ϕ-functions in sequence is unimportant for the result.

You can see the result of inserting the ϕ-functions in our recurring example in figure 4.5. Notice the just mentioned descending order of the ϕ-functions.
As constants are also placed in registers, and several constants might hit the same register, here register v_3, they get a ϕ-function as well, although it is never needed. This happens because the path-convergence criterion, and so our entire algorithm, does only care about definitions, but no usages. After renaming all variables we probably could delete v_3's ϕ-instruction.
Another important difficulty is the *EXIT* node. As you can see in the figure, the *return* node is not the last one but *EXIT*. Just like the *ENTRY* node it just symbolises the point where the programme starts or ends, and is no instruction at all. Although it makes no difference if the *ENTRY* node stays in, until the algorithm is complete, the *EXIT* node must be removed and can be inserted afterwards, otherwise the algorithm will insert ϕ-functions in the *return* node.

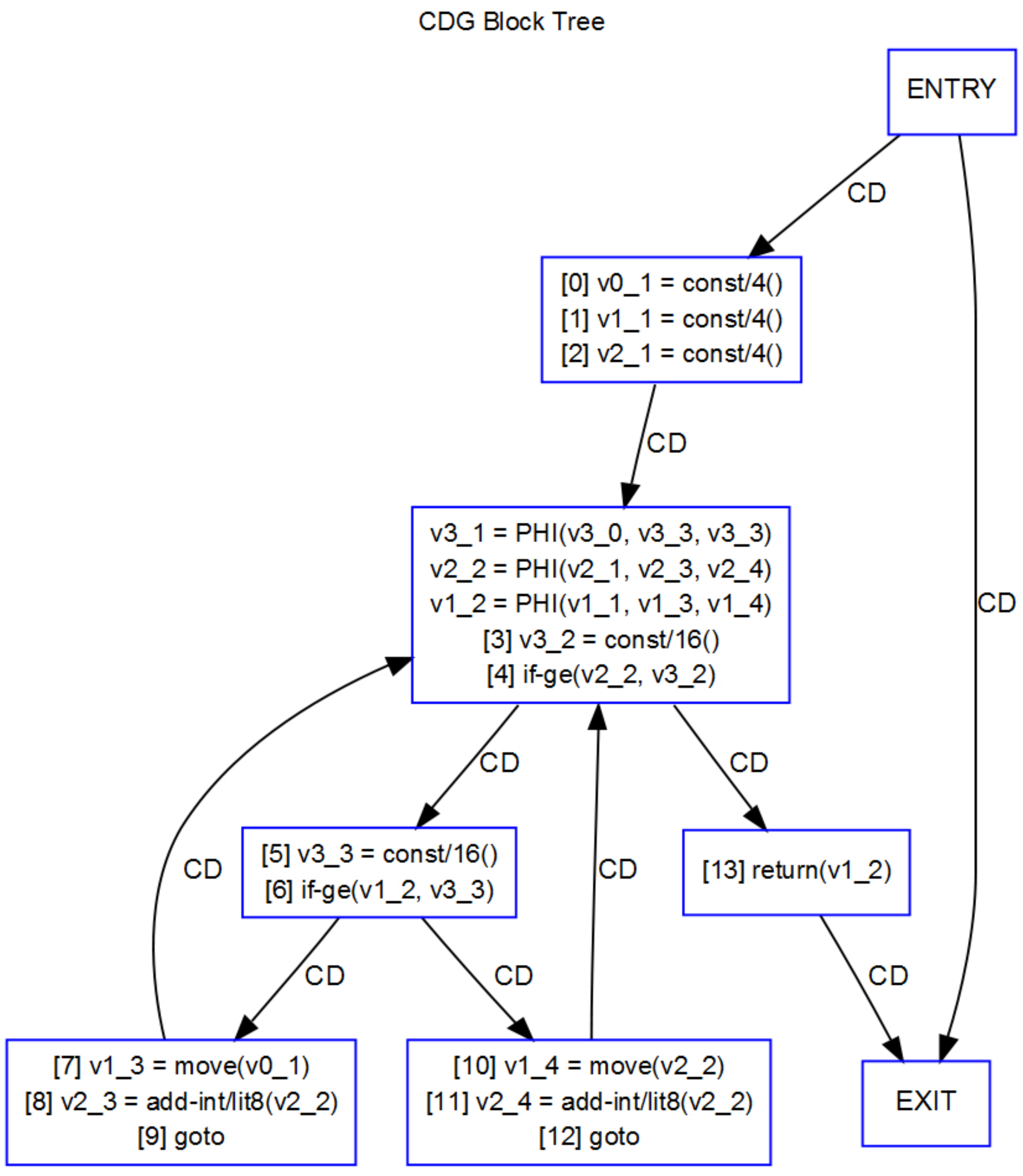


Figure 4.5.: listing 2.1's SSA block graph

4.6.2. Renaming the Variables

Having the ϕ-functions we need to rename all variables, including those in the ϕ-function itself. Given a variable a we will rename all different definitions to $a_1, a_2, \ldots$.
Again the trivial algorithm for renaming the variables would be slow. We would just change each definition's variable name and rename each use after that with the same name.

A better solution can be achieved by using the dominance frontier. (see listing 4.6)

```
Initialization
    for each variable $a$
        $Count[a] \leftarrow 0$
        $Stack[a] \leftarrow$ empty
        push 0 onto $Stack[a]$

Rename($n$) =
    for each statement $S$ in block $n$
        if $S$ is not a $\phi$-function
            for each use of some variable $x$ in $S$
                $i \leftarrow top(Stack[x])$
                replace the use of $x$ with $x_i$ in $S$
        for each definition of some variable a in S
            $Count[a] \leftarrow Count[a] + 1$
            $i \leftarrow Count[a]$
            push $i$ onto $Stack[a]$
            replace definition of $a$ with definition of $a_i$ in $S$

    for each successor $Y$ of block $n$
        suppose $n$ is the $j$th predecessor of $Y$
        for each $\phi$-function in $Y$
            suppose the $j$th operand of the $\phi$-function is $a$
            $i \leftarrow top(Stack[a])$
            replace the $j$th operand with $a_i$

    for each child $X$ of n
        Rename($X$)

    for each statement $S$ in block $n$
        for each definition of some variable $a$ in $S$
            pop $Stack[a]$
```

Listing 4.6: Rename variables. [App98]

Again we just call `rename` on the root node.
Before we have to initialize `Count` and `Stack`. `Count[a]` holds the amount of definitions of a variable a. `Stack[a]` holds the rename suffixes.

First part of the algorithm is to walk all instructions in the block. We need to replace the usages of a variable. All definitions of a variable must be counted as we need that much suffixes, then we can rename all definitions of a variable in this block.

After that all ϕ-functions have to be renamed. We must walk all successors of the current block. Here we suppose that the current block is the jth predecessor of the successor. We just change then the jth operand of the ϕ-function. By doing this it is even possible to later find out which operand of the ϕ-function came from which edge.

This renaming is done with all children of the current node.
Finally in each `rename` call we have to pop one element from the stack as we already renamed all necessary variables with this suffix.

The real implementation looks a little bit larger as there had to be inserted some conditions to ensure the programme will not fail.
Especially the `top(Stack[x])` could lead to `NumberFormatException`. This occurs as an instruction's variables are represented as a `String` to make it easier to rename. Each time the programme reads a variable which is already renamed, parsing it to an **`int`** fails. Omitting this error was not to lead to any error so far.

The final result can be seen in figure 4.5. As already mentioned you can see where each parameter of a ϕ-function comes from.
As constants are used in the **`if`**-instructions, their ϕ-node appears to be unnecessary, at least if every constant was defined at the top with its own name right from the beginning.
Especially here the $EXIT$ node must be omitted from the graph by the same reasons as stated in the subsection above.

5. Evaluation

Having solved the problem converting a CFG to SSA-form, we need to test our results and check whether the given runtime complexity is met.

5.1. Realisation of Planned Targets

Back in chapter 3 we looked at the requirements for this project. We will check all points whether they were met or not.

Certainly parsing the `dex`-files and building a CFG of it could be met.
The dominators and dominance frontiers could be computed with the given algorithm and even all test cases did succeed while testing. Finally the SSA graph was calculated as required and could also stick with the given algorithm.

Whenever applicable, object-oriented programming was used. However still no final industry standard product was created.

The programme is expected to be easily re-integrated into SMALI as the origin files of SMALI were not changed during programming. Only new files were added, which do not change any of the invariants of given classes.
As the programme is just doing computation some more integrative work has to be done to use it in the Joana project. However the most work is expected in migrating the SMALI part to a user-friendly frontend, as the new classes can just be migrated the way they are.

5.2. Testing

To test the implementation, unit tests have shown to be a good way to do so. Unfortunately they appear to be too costly to create, at least for the SSA graph.
We tested the Java-files from Appendix A.2 as well as the already mentioned "MoWiDi"-project. This adds up to 1773 methods. Although many methods may have no branches or loops etc., e.g. initialisations. Here it was important that no error occured, results were only spot checked. Additionally there were automatic JUnit tests carried out for the dominator calculation, its base were two different CFGs with 7 nodes each.

First we test our CFG implementation. This implementation has to be correct per definition, as we only traverse the already given graph of SMALI. Anyhow several tests were made by comparing hand made CFGs of some code example with the computed CFG.

Although there was no issue, it was possible to see that SMALI or Java works differently from the trivial idea. Remember listing 2.1 and its CFG figure 4.2.
There might be only one instruction in Java, like **`if`** `(k < 100)`, but in real you need two instructions: one **`if`** and one to load `100` into a register. Conclusively you need more nodes than anticipated.

The hardest part is the easiest to confirm. `Dominators` only need a normal directed graph, no matter the content of the nodes, so testing can be done very easily by creating some jUnit tests.
It was tested with two graphs. Each of them has 7 nodes and 5 respectively 6 edges. Dominance frontier was only tested with one of them. It is easy to add even more graphs and expected results.
During testing several mistakes could be found and solved. Those mistakes were not possible to "see" as they were caused by insufficient commenting of the SMALI project, which caused wrong usage of certain methods.
Dominator and DF calculation is expected to be correct by testing and visual confirmation of certain graphs.

The last to confirm is the SSA graph. Beginning this project we expected to be able to easily write jUnit tests for the SSA graph. Unfortunately writing tests was put to the end of the implementation work and several design decision conclusively shown to be bad for testing.
To test it we need to put a CFG and the dominators to the SSA graph build call. The nodes have to contain `InstructionNodes`, so the SSA graph algorithm can work. Although those instruction nodes were extended to allow direct creation without SMALI several errors raised, which probably occurred due to hash collisions. As confirming correctness visually seems to be sufficient the error seeking is adjourned for a later version.
Placing the ϕ-nodes was easy to check as it was pretty obvious in small but complex graphs where to put them (Just like our recurring example).
Checking on the renaming was easy as well: each variable name may only occur once and ϕ-nodes have to join all versions of the certain variable.
By checking several graphs manually there were several errors found which were caused by hash collisions and the (then not yet tested) wrong dominance frontier. Building the SSA graph on-the-fly appeared to be a bad idea, too. Some edges might be changed or deleted during the algorithm, which was not covered by the first version. The current version avoids this problem by building the tree just after the algorithm.

Additionally to the SSA graph there is a `SSABlockGraph`. As already mentioned, the algorithm needs a block view of the instructions. To insert ϕ-nodes into the SSA graph's block tree they are just pushed onto the block's top, whilst in the normal SSA graph they are inserted below the preceding, if existing, ϕ-node. So if there are several ϕ-nodes in one block they are descending while being ascending in the normal SSA graph. As this does not change control flow or correctness it was not changed. However the SSA graph and its SSA block graph are then no longer in the same sequence!

According to everything above the programme is working correctly and performs as expected and required.

5.3. Time

Before having a look at the charts we look at the given time bounds the programme should run in:

Computation of	**Time Bound (average)**
CFG	$O(\|N\| + \|E\|)$
DOM and DF	$O(\|E\| + \sum_{X \in N} \|DF(X)\|)$
Place ϕ-Nodes	$O(A_{tot} \times avrgDF)$
Rename Variables	$O(M_{tot})$

with

N and E	nodes and edges
A_{tot}	total number of assignments to variables in resulting programme
$avrgDF$	weighted average of the sizes $DF(X)$
M_{tot}	total number of mentions of variables in resulting programme

Table 5.1.: Time bounds according to [Cyt+91]

Let us have a look at the results of several tests. Especially little amount of nodes or edges can represent a smooth line in each of the figures, whilst high numbers of nodes or edges follow only roughly a linear regression. Tests were done method by method and calculation by calculation. Afterwards for each amount of nodes or edges an average runtime was calculated. Before starting time measurements the same calculation was performed several times (100 at least) to force Java doing all optimizations and garbage collection not during measurements.
You might remember: Every method goes for its own. A usual method will not have so much instructions, at least if you adhere to current programming standards. Hence there were little real tests available for large numbers of nodes. If so we probably could see even high counts follow the required time bounds.
Several own little examples and classes from a large Android project were used to test the time bounds.
Some of the tests, which were used to obtain the timings, can be seen in table 5.2.

In figure 5.1 and 5.2 we see the time needed for calculating the CFG from given instructions. Although in high numbers of vertices time has some peaks, it stays in linear style.

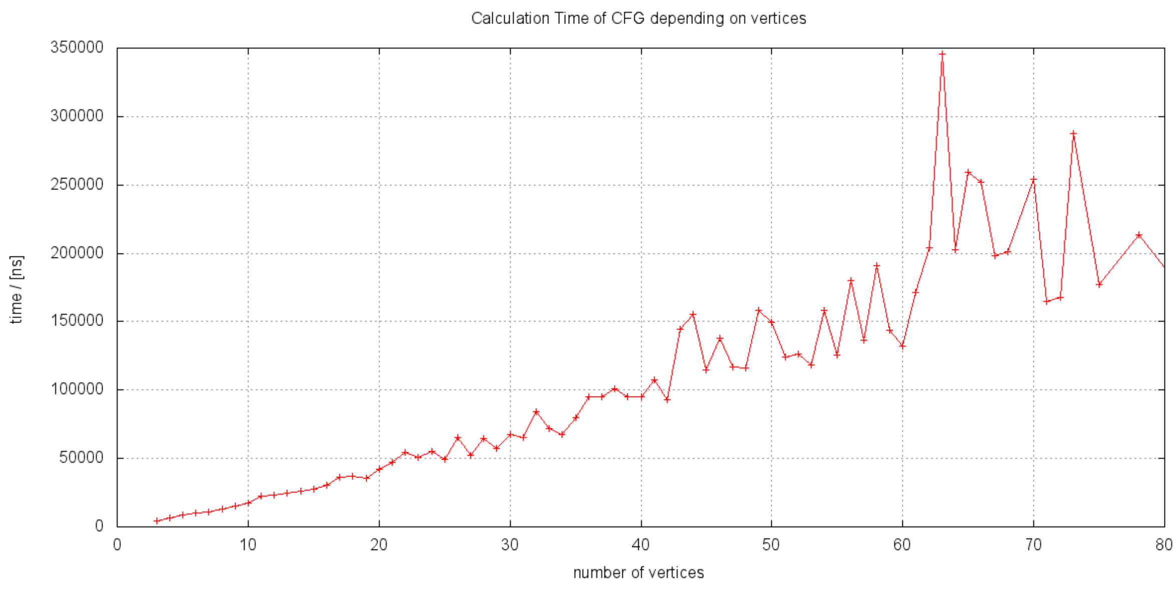


Figure 5.1.: Calculation time solely of CFG depending on vertices.

We can easily confirm the time bound requirement of DFS: $O(N + E)$.

As SMALI already holds some kind of tree information for each instruction, CFG calculation even without a figure would have been easy.
The harder parts begin here, starting with calculation of DOM and DF. Just the DOM tree should calculate in $O(N^2)$. With an optimization even in $O(N \log N)$, this was not implemented, however. The DF is calculated almost in linear time. In most cases the total size of all DFs is approximately linear in the size of the graph [App98]. So DF grows not faster than DOM and can be neglected.

Although large numbers bring again unsmooth values, we see in figure 5.3 and 5.4 that until 40 vertices or edges it appears to follow linear time. Larger amounts can jump higher or lower as expected by $O(N^2)$.
Already mentioned: Usual methods will not have that much vertices and edges, we can expect an average linear runtime here for state-of-the-art object oriented programmes.

Coming now to the calculation of the SSA graph. It has to calculate where to put the ϕ-vertices and has to rename all variables.
In worst case number of inserted ϕ-vertices is N^2 although in the usual case the amount is just proportional to N. In practice we have $O(N)$ for this part. We can confirm that by investigating figure 5.5 and 5.6. Looking just at the curve depending on edges we can see a pretty linear curve. Also it is possible to see that if a programme with little vertices can have many edges and vice versa. Hence by looking at the curve depending on vertices there are more peaks out of the targeted linear curve.
Nevertheless we can expect practical linear time or with little deviations, especially for

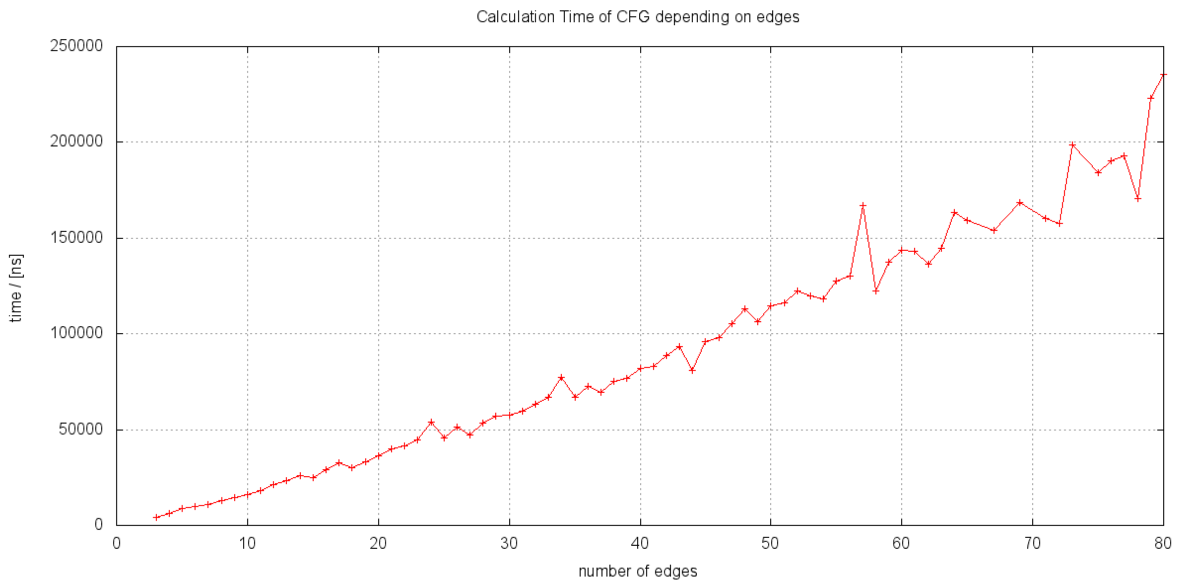


Figure 5.2.: Calculation time solely of CFG depending on edges.

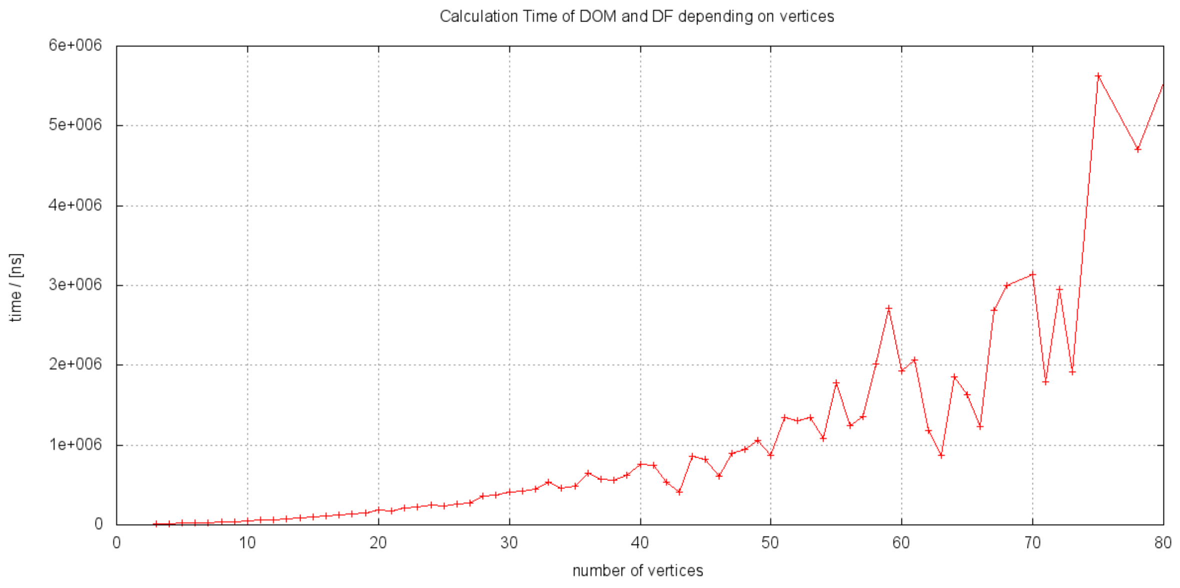


Figure 5.3.: Calculation time solely of DOM and DF depending on vertices.

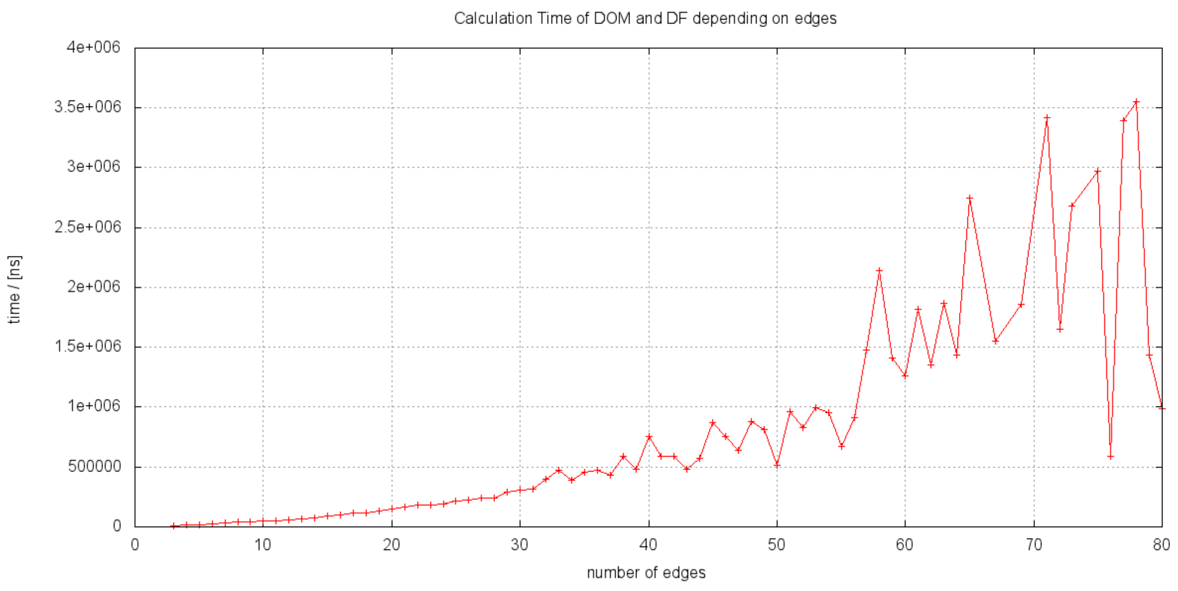


Figure 5.4.: Calculation time solely of DOM and DF depending on edges.

File and Method	$\lvert V \rvert$	$\lvert E \rvert$	T_{CFG}	T_{DOM}	T_{SSA}	T_{Total}	$\lvert \phi \rvert$
`Test191.boing`	16	18	27273	87804	71285	186362	3
`Test.foo`	12	13	17887	55637	38115	111639	0
`Test.bar`	12	13	19627	46601	59854	126082	1
`GraphWalker.dfs`	24	26	40642	223525	83180	347347	3
`GraphWalker.dfsNoRecursion`	54	61	133343	1377160	267250	1777753	18

Table 5.2.: Time of certain methods, all times are nanoseconds. (See appendix section A.2)

little methods.

Except for calculation of the DOM, which takes $O(N^2)$, especially in large amounts of vertices, the entire algorithm works in average in linear time.
Let us confirm this statement in the last chart 5.7 and 5.8. Here all times are accumulated. Apparently calculation depends on the time of DOM calculation, as this is the only non-linear calculation.
Having low numbers of nodes the total time still is almost linear. High numbers again will just rise to $O(N^2)$. But still the overall time is just acceptable and as expected.

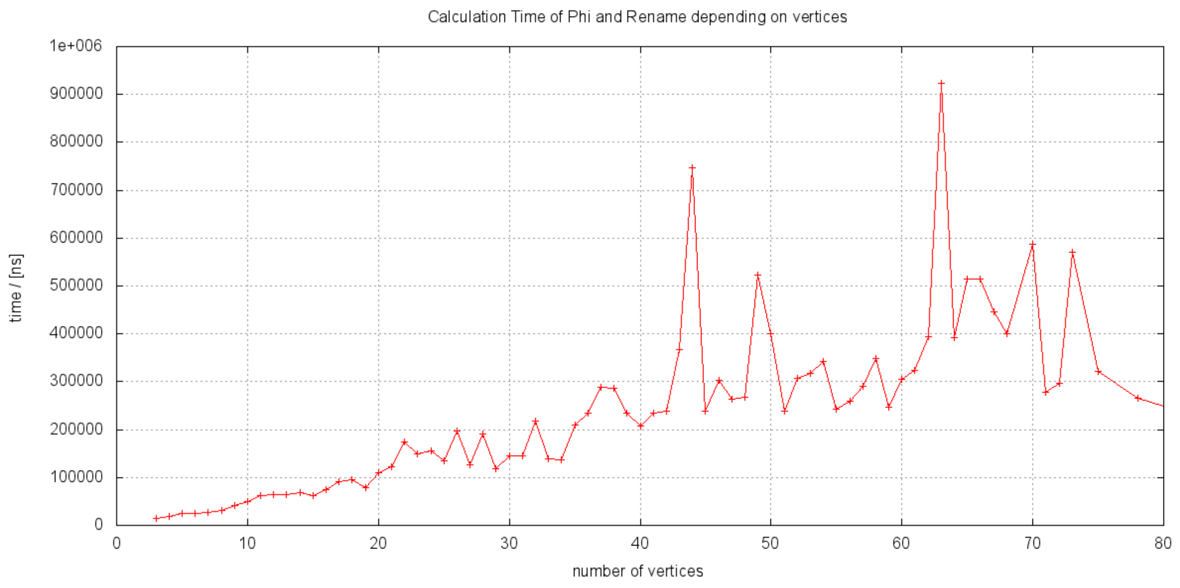


Figure 5.5.: Calculation time solely of SSA graph depending on vertices.

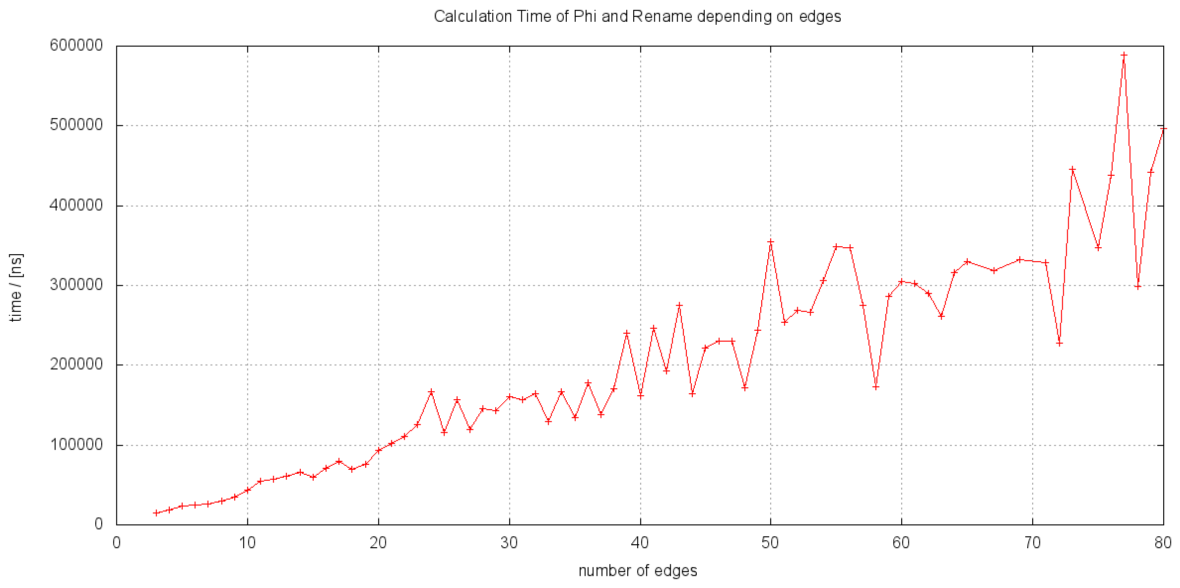


Figure 5.6.: Calculation time solely of SSA graph depending on edges.

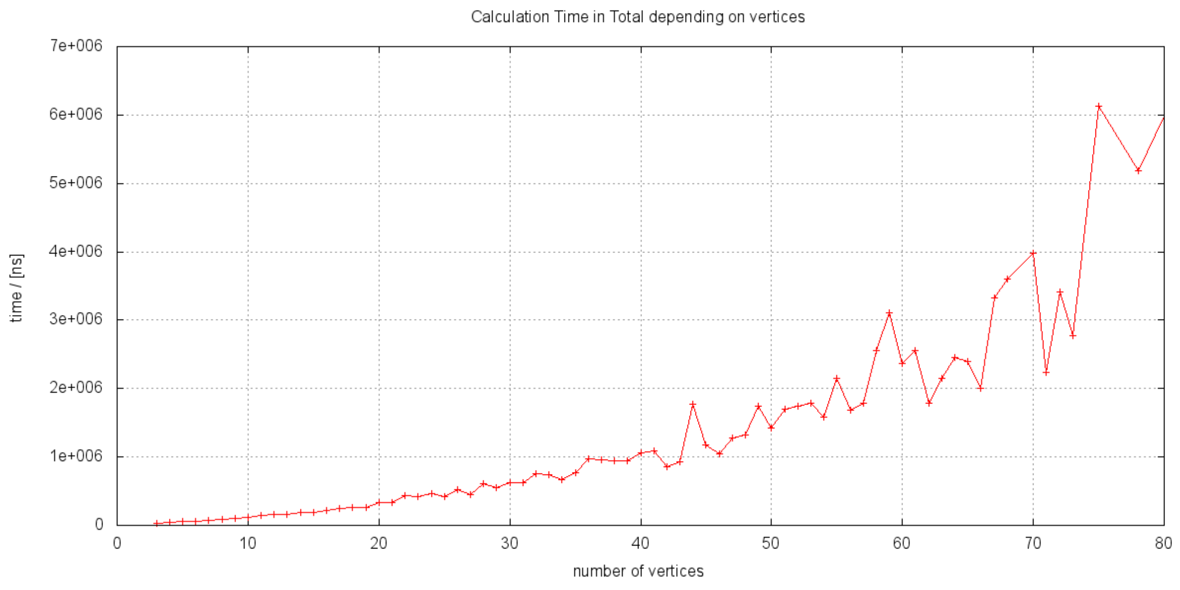


Figure 5.7.: Total calculation time depending on vertices.

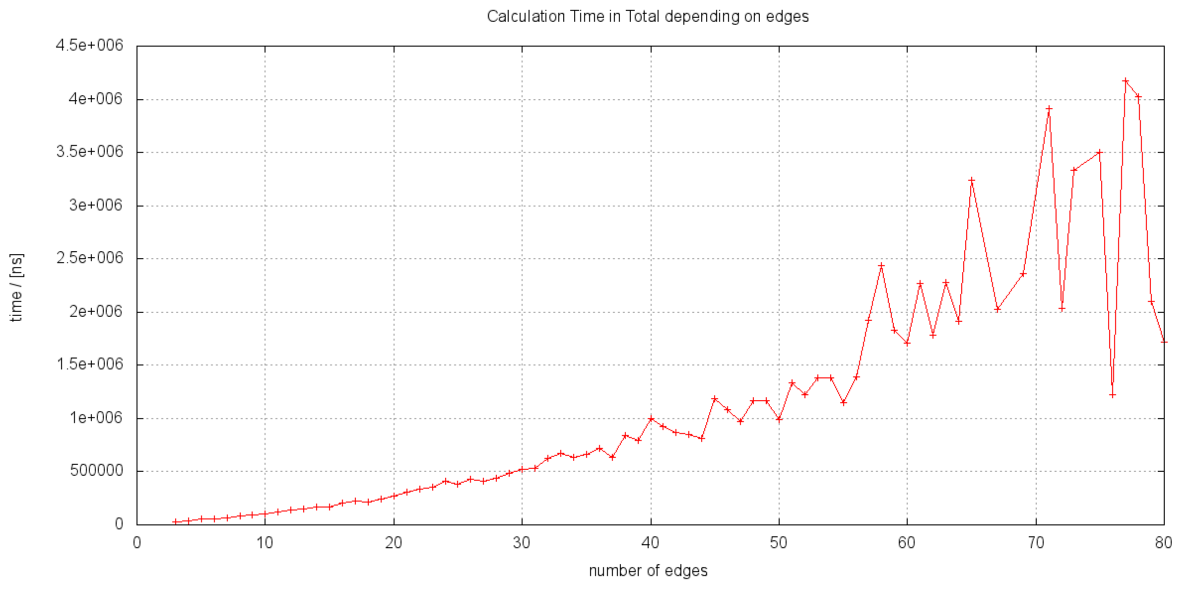


Figure 5.8.: Total calculation time depending on edges.

6. Conclusions

We just computed the SSA-form of Android bytecode and made the first steps to build a frontend for a programme analysis framework, e.g. WALA. The next step would be to translate our Dalvik Bytecode in SSA-form to the intermediate representation of the WALA framework. After that we could analyse Android applications without the need of another intermediate language.

Parts of this project shall be included in the SMALI project, so many developers could create their own frontends or framework. Having tested this prototype with 1773 methods without any errors, the requested standards were met and even broken upwards. However several optimisations to the code itself could be made. The programme could be easily extended with threads, which would make the programme more scalable.

Bibliography

[App98] Andrew W. Appel. “Modern Compiler Implementation in Java”. In: Static Single-Assignment Form. Cambridge University Press, 1998. Chap. 19. ISBN: 0-521-58388-8.

[CF94] Ron Cytron and Jeanne Ferrante. “Efficiently computing ϕ-nodes on-the-fly”. In: *Languages and Compilers for Parallel Computing*. Ed. by Utpal Banerjee et al. Vol. 768. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1994, pp. 461–476. URL: `http://dx.doi.org/10.1007/3-540-57659-2_27`.

[Cyt+91] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS* 13 (1991), pp. 451–490.

[Dal] *Bytecode for the Dalvik VM*. 2011. URL: `http://source.android.com/tech/dalvik/dalvik-bytecode.html`.

[Jav] *Java bytecode: Understanding bytecode makes you a better programmer*. 2011. URL: `http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/`.

[LEJ79] Thomas Lengauer, Robert Endre, and Tar Jan. “A fast algorithm for finding dominators in a flowgraph”. In: *ACM Transactions on Programming Languages and Systems* 1 (1979), pp. 121–141.

[Sma] *smali – An assembler/disassembler for Android’s dex format*. May 2011. URL: `http://code.google.com/p/smali/`.

[Soo] *Soot*. 2011. URL: `http://http://www.sable.mcgill.ca/soot/`.

[Wal] *WALA*. May 2011. URL: `http://wala.sourceforge.net`.

[Wika] *Bytecode for the Dalvik VM*. 2011. URL: `http://en.wikipedia.org/w/index.php?title=Dominator_(graph_theory)&oldid=435620107`.

[Wikb] *Static Single Assignment form*. 2011. URL: `http://en.wikipedia.org/w/index.php?title=Static_single_assignment_form&oldid=457391598`.

List of Figures

List of Tables

Listings

A. Appendix

A.1. Manual of the Programme

The programme is controlled by command line arguments.

Arguments you will need for creating a CDG
Enter needed bootclass path entries with flag `-c` and the entries:
```
-c libandroid/core.jar:libandroid/framework.jar:libandroid/ext.jar:li
```
You will need those `jar`-files to make the programme work!

If you want graphic output you need to signal `-g` and specify which output to create. You may decide between none, some or all:

- `CFG` to output the CFG as graphic and/or
- `DOM` to output the dominator tree as graphic and DF as text file and/or
- `CDG` to output the CDG as tree and block tree and/or
- `EXC` to include exception control flow paths in calculation and output

Additionally the target position for the output files must be specified. This part could look like this:
```
-g CFG,DOM,CDG=./out
```

Eventually you have to put to the end the `classes.dex` file you want to work with. The entire call could look like this:
```
java dalvik-ssa -c libandroid/core.jar:libandroid/framework.jar:liban
-g CFG,DOM,CDG=./out ./../HelloAndroid/bin/classes.dex
```

A.2. Test Files

A.2.1. Test191.java

```java
package edu.kit.ipd;

public class Test191 {

    public static int boing() {
        int i = 1;
        int j = 1;
        int k = 0;

        while (k < 100) {
            if (j < 20) {
                j = i;
                k = k + 1;
            } else {
                j = k;
                k = k + 2;
            }
        }
        return j;
    }
}
```

Listing A.1: Test191.java

A.2.2. Test.java

```java
package edu.kit.ipd;

public class Test {

    private int x = 5;

    public void foo() {
        x++;
        if (x > 234) {
            throw new IllegalStateException();
        }
    }

    public void bar() {
        x++;
        try {
            foo();
        } catch (Throwable t) {
        }
        x++;
    }
}
```

Listing A.2: Test.java

A.2.3. GraphWalker.java

```java
package edu.kit.ipd;

import java.util.HashSet;
import java.util.Set;
import java.util.Stack;

public abstract class GraphWalker<V, E> {

    public static class DirectedGraph<V, E> {

        public Iterable<E> outgoingEdgesOf(V node) {
            return null;
        }

        public V getEdgeTarget(E edge) {
            return null;
        }

        public Set<V> vertexSet() {
            return null;
        }
    }
    private final DirectedGraph<V, E> graph;

    public GraphWalker(DirectedGraph<V, E> graph) {
        this.graph = graph;
    }
    private static final boolean NO_RECURSION = false;

    public final void traverseDFS(final V start) {
        if (NO_RECURSION) {
            dfsNoRecrusion(start);
        } else {
            Set<V> visited = new HashSet<V>();
            dfs(start, visited);
        }
    }

    private void dfs(final V node, final Set<V> visited) {
        visited.add(node);

        discover(node);

        for (final E out : graph.outgoingEdgesOf(node)) {
            final V succ = graph.getEdgeTarget(out);
            if (!visited.contains(succ)) {
                dfs(succ, visited);
            }
        }

        finish(node);
    }

    public abstract void discover(V node);

    public abstract void finish(V node);

    /*
     * Not sure this really works...
     */
    private void dfsNoRecrusion(final V entry) {
        // iterate dfs finish time
        final Set<V> visited = new HashSet<V>(graph.vertexSet().size
           ());
        final Stack<V> stack = new Stack<V>();
        V current = entry;

        while (current != null) {
            boolean foundOne = false;

            if (!visited.contains(current)) {
                visited.add(current);
                stack.push(current);

                discover(current);

                for (E succEdge : graph.outgoingEdgesOf(current)) {
                    // there may be a problem iff the successor can
                       be the same node twice (or more)
                    // think of succ normal flow + succ exception
                       flow. But this never happens in the current
                    // code. Even with empty catch blocks.
                    final V succ = graph.getEdgeTarget(succEdge);
                    if (!visited.contains(succ)) {

                        // this is slow and should be removed
                        stack.remove(succ);

                        if (!foundOne) {
                            foundOne = true;
```

```java
                              current = succ;
                          } else {
                              stack.push(succ);
                          }
                      }
                  }
              } else {
                  // finished current node. act.
                  finish(current);
              }

              if (!foundOne) {
                  current = (stack.isEmpty() ? null : stack.pop());
              }
          }
      }
}
```

Listing A.3: GraphWalker.java