NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Declarative type inference and SSA transformation of Android applications

Ilias M. Tsatiris

Supervisors: **Yannis Smaragdakis,** Professor NKUA
George Fourtounis, Research Associate NKUA

ATHENS

JUNE 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Δηλωτική εξαγωγή τύπων και μετασχηματισμός εφαρμογών Android σε SSA μορφή

Ηλίας Μ. Τσατίρης

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Ερευνητικός συνεργάτης ΕΚΠΑ

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2019

BSc THESIS

Declarative type inference and SSA transformation of Android applications

Ilias M. Tsatiris
R.N.: 1115201500162

SUPERVISORS: **Yannis Smaragdakis,** Professor NKUA
George Fourtounis, Research Associate NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Δηλωτική εξαγωγή τύπων και μετασχηματισμός εφαρμογών Android σε SSA μορφή

Ηλίας Μ. Τσατίρης
Α.Μ.: 1115201500162

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Ερευνητικός συνεργάτης ΕΚΠΑ

ABSTRACT

Android is everywhere: from mobile devices and TVs to more recently automobile infotainment systems. This widespread use of the Android OS makes application analysis ever so important, in order to optimize the application performance and system resource usage.

Static Single Assignment (SSA) is a convenient normalized program representation that can simplify several static analyses. However, the underlying DEX bytecode of an Anrdroid application does not have the SSA property, and further processing of the program must be done in order to obtain it. Moreover, DEX bytecode provides almost no explicit type information for the program's variables, limiting the accuracy of the analyses.

In this thesis, we develop a Datalog program that transforms an Android application into SSA form, effectively extending the functionality of the DEX fact generation front end of the DOOP framework. Additionally, we present a type inference algorithm, to resolve the types of the variables that the SSA transformation produced.

SUBJECT AREA: Static Program Analysis and Program Transformation

KEYWORDS: static program analysis, program transformation, DOOP framework, dex bytecode, type inference

ΠΕΡΙΛΗΨΗ

Το Android είναι παντού: από τις κινητές συσκευές και τις τηλεοράσεις, μέχρι πιο πρόσφατα στις κεντρικές κονσόλες των αυτοκινήτων. Αυτή η ευρεία χρήση του λειτουργικού Android καθιστά την ανάλυση εφαρμογών όλο και πιο σημαντική, προκειμένου να βελτιστοποιήσουμε την απόδοση τους και την κατανάλωση πόρων του συστήματος.

Το Static Single Assignment (SSA) είναι μια βολική κανονικοποιημένη αναπαράσταση προγράμματος η οποία μπορεί να απλοποιήσει αρκετές στατικές αναλύσεις. Ωστόσο, το υποκείμενο DEX bytecode μιας εφαρμογής Android δεν έχει την ιδιότητα SSA, και απαιτείται περαιτέρω επεξεργασία προκειμένου να την αποκτήσει. Ακόμη, το DEX bytecode περέχει μηδαμινή ρητή πληροφορία τύπων για τις μεταβλητές του προγράμματος, περιορίζοντας έτσι την ακρίβεια των αναλύσεων.

Σε αυτή την πτυχιακή εργασία, αναπτύσουμε ένα πρόγραμμα Datalog το οποίο μετασχηματίζει μια εφαρμογή Android σε SSA μορφή, επεκτείνοντας ουσιαστικά την λειτουργικότητα του εμπρόσθιου τμήματος του Doop που παράγει DEX facts. Περαιτέρω, παρουσιάζουμε έναν αλγόριθμο εξαγωγής τύπων, προκειμένου να παράξουμε πληροφορία τύπων για τις μεταβλητές που ο μετασχηματισμός μας παρήγαγε.

ΠΕΡΙΟΧΗ ΑΝΤΙΚΕΙΜΕΝΟΥ: Στατική ανάλυση προγραμμάτων και μετασχηματισμός προγραμμάτων

ΛΕΞΕΙΣ-ΚΛΕΙΔΙΑ: στατική ανάλυση προγραμμάτων, μετασχηματισμός προγραμμάτων, DOOP framework, dex bytecode, εξαγωγή τύπων

To my parents Roxani and Mihalis,
and my brother Manos.

ACKNOWLEDGMENTS

I’d like to thank my supervisors, Yannis Smaragdakis and George Fourtounis, for coming up with the thesis topic, and giving me the opportunity to work on such an interesting project. Their work ethic and attitude is admirable, to say the least.

I’d also like to thank the rest of the PLAST lab, for the insight and help they provided whenever I needed assistance.

June 2019

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Static program analysis is the analysis of computer programs in order to comprehend their structure and extract information about all possible executions. Static analysis typically inspects only the program text (in source or inermediate code). Rice's theorem tells us that all non trivial properties of programs are undecidable, thus static analysis frameworks aim to give an accurate approximation to the problem they are tackling.

Doop is a declarative pointer/taint analysis framework for Java programs, which can also analyse Android applications. A typical Android app consists of a single (.apk) file, that contains, among other files, Dalvik Executables (.dex). Doop's dex frontend transforms the input dex files into relations (sets of tuples), in order to perform the analysis in Datalog: a declarative programming language that has found great success in many fields, including program analysis. Even though Doop's dex front end outputs valid input facts, they neither are in SSA form nor have their types been fully resolved.

Our goal is to extend the Doop dex fact generation pipeline, by implementing an SSA transformation and type inference algorithm in Datalog. Our algorithm will preprocess the input facts before the actual analysis is run, thus solving the problem that was described above.

The rest of the thesis is organized as follows:

1. In chapter 2, we provide an overview of the Android environment, the Dalvik bytecode, SSA Form and the Doop framework.
2. In chapter 3, we introduce the core Datalog rules of the transformation program.
3. In chapter 4, we introduce the core Datalog rules of the type inference section of our algorithm.
4. In chapter 5, we present our experimental results.
5. In chapter 6, we conclude this thesis.

2. BACKGROUND

2.1 Android platform

Android [5] is an open source software stack developed by Google, based on the Linux Kernel. Even though it is predominately used in phones, it was created to support a wide range of mobile devices, all of which are quite limited in terms of performance and resources, like battery life and storage.

The use of the Linux kernel as the basis of the Android stack means that Android is a stable and secure platform, since the Linux kernel is being researched and maintained by thousands of developers.

Since Android version 5.0, every application runs in its own instance of the Android Runtime (ART). ART and its the predecessor Dalvik were created specifically for the Android project. Their role is to manage multiple virtual machines on low-memory devices. Both ART and Dalvik execute DEX files, a bytecode format that is optimized to minimize the application's memory usage. ART [7] offers many improvements over Dalvik, such as improved garbage collection, ahead-of-time compilation and better debugging support.

Android Applications are usually developed in a language that runs on the JVM [6], mainly Java and more recently Kotlin. The sources are compiled to Java bytecode, and with the help of the dx tool, which is part of the official Android SDK build tools, they get compiled to Dalvik bytecode that the ART is able to execute. Android also allows developers to write native code in languages like C and C++, with the help of the Native Development Kit (NDK) that Android provides [15]. The DEX files, native code, libraries and the related metadata are all packed together in an APK file (Application Package), an archive that contains all the contents of an Android Application, and is used for their distribution and installation.

A typical APK file has the following structure [18]:

```
app.apk
├─ META-INF
│  ├─ MANIFEST.MF
│  ├─ CERT.RSA
│  └─ CERT.SF
├─ lib
│  ├─ armeabi
│  ├─ armeabi-v7a
│  ├─ arm64-v8a
│  ├─ x86
│  ├─ x86_64
│  └─ mips
├─ res
├─ assets
├─ AndroidManifest.xml
├─ classes.dex
└─ resources.arsc
```

Figure 1: Structure of an Android Package (APK)

META-INF	the directory that contains application metadata.
lib	the directory containing libraries as well as user native code compiled to architecture-specific binaries.
res	the directory containing resources not compiled into resources.arsc
assets	the directory containing the apps assets
AndroidManifest.xml	file containing additional application metadata
classes.dex	file containing all the classes in the application source compiled to dex bytecode format
resources.arsc	file containing some precompiled resources

2.2 Dalvik bytecode

As mentioned before, every application runs in its own instance of the Android Runtime (ART) or Dalvik VM depending on the Android version. Both of these runtimes execute Dalvik Executable files which contain Dalvik bytecode.

Due to the importance of the DEX format and Dalvik bytecode, we will give a brief overview of the basic characteristics of the Dalvik runtime, with an emphasis on the underlying bytecode.

2.2.1 General design

The machine model of Dalvik bytecode is designed in a way that is similar to a real architecture. [12] Unlike the Java VM, it is a register based VM, thus fewer but more complex commands are typically needed to perform a given task.

When used for primitive type values, like ints and floats, registers are 32-bit. For 64-bit values such as double or long, adjacent registers are paired. When used as object references, registers are sufficiently wide to hold exactly one reference.

Some instructions are type-general, meaning that they are not limited to a specific type. For example a 32-bit move instruction may be used to move any 32-bit value from one register to another, no matter the type that either the source or the destination register holds. This means that a register in Dalvik bytecode may be associated with multiple types that may not be related. Our SSA transformation and type inference, which are discussed later in this thesis, aim to resolve this issue.

Finally, Dalvik does not have a dedicated null constant to represent null references, but uses the value 0 instead, which further complicates the type resolution issues that were mentioned above.

2.2.2 Smali and example code

Dalvik Executables are binary files in a format that the Dalvik VM can easily read and execute. However this means that the format is unreadable by humans. Smali/baksmali is an assembler/disassembler for the dex format. Its syntax is similar to a competing tool, dedexer. Dissasembling a Dalvik file is useful when trying to reverse engineer an application, or, in our case, when debugging program transformers that operate at the bytecode level.

The whole process of disassembling the dex file contained in an apk file, is handled quite nicely by the apktool utility [16].

Lets give a code example:

```
package Demo;

import java.io.*;

public class Main {
  public static void main(String[] args) {
        int i = 0;

        String a = null;

        a = new String("Demo");

        if (i == 0)
            System.out.println(a);

        if (a != null)
            System.out.println(a);
    }
}
```

Figure 2: Simple Java program to demo Java to DEX compilation

We can then compile the program with javac, and use dx on the .class files to create an .apk file. All our Java code has been compiled into Dalvik bytecode and is contained within the classes.dex file inside the .apk.

Finally we can use apktool on the apk, to decompile the Dalvik bytecode into smali, so that we can inspect the result of the compilation.

```
.class public LDemo/Main;
.super Ljava/lang/Object;
.source "Main.java"

# direct methods
.method public constructor <init>()V
    .locals 0

    .prologue
    .line 5
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method
```

Figure 3: Smali code example

```
.method public static main([Ljava/lang/String;)V
    .locals 3
    .param p0, "args" # [Ljava/lang/String;

    .prologue
    .line 7
    const/4 v1, 0x0

    .line 9
    .local v1, "i":I
    const/4 v0, 0x0

    .line 11
    .local v0, "a":Ljava/lang/String;
    new-instance v0, Ljava/lang/String;

    .end local v0  # "a":Ljava/lang/String;
    const-string v2, "Demo"

    invoke-direct {v0, v2}, Ljava/lang/String;-><init>(Ljava/lang/String;)V

    .line 13
    .restart local v0 # "a":Ljava/lang/String;
    if-nez v1, :cond_0

    .line 14
    sget-object v2, Ljava/lang/System;->out:Ljava/io/PrintStream;

    invoke-virtual {v2, v0},
        Ljava/io/PrintStream;->println(Ljava/lang/String;)V

    .line 16
    :cond_0
    if-eqz v0, :cond_1

    .line 17
    sget-object v2, Ljava/lang/System;->out:Ljava/io/PrintStream;

    invoke-virtual {v2, v0},
        Ljava/io/PrintStream;->println(Ljava/lang/String;)V

    .line 18
    :cond_1
    return-void
.end method
```

Figure 4: Smali code example (cont.)

Even though at first glance it might seem like there is explicit type info for the registers through the .local directives, these are just debug symbols and are typically absent from production APKs.

2.3 SSA Form

Static Single Assignment (SSA) is a form of Intermediate Representation code that enables and enhances various code optimizations [1] and static analyses.

The basic property of SSA Form that differentiates it from typical three address code is the fact that the left-hand sides of all assignments are distinct: no variable gets assigned twice. For example:

Listing 1: Three address code snippet

```
x = 1
y = 2
x = x + y
```

Listing 2: Three address code in SSA Form

```
$x_1$ = 1
$y_1$ = 2
$x_2$ = $x_1$ + $y_1$
```

Even though the above property might seem sufficient, a problem arises when we introduce control flow:

Listing 3: Control flow without SSA

```
if (c) then
  x = 1
else
  x = 2

y = x
```

Listing 4: Control flow in SSA Form

```
if (c) then
  $x_1$ = 1
else
  $x_2$ = 2

y = $x_?$
```

The problem here is that we don't know statically (assuming no constant propagation can be done) which control flow path the execution will follow, thus we don't know which definition of x to assign to y. To solve this problem, SSA introduces the ϕ function that combines two (or more) definitions of a variable and returns the correct one, based on the control flow path that was taken to reach the ϕ assignment statement. With the use of this function, or example will now become:

Listing 5: Control flow in SSA Form with Phi function

```
1 if (c) then
2   $x_1$ = 1
3 else
4   $x_2$ = 2
5 $x_3$ = $\phi$($x_1$, $x_2$)
6 y = $x_3$
```

In this example, ϕ returns x_1 if the control flow went through line 2, else it returns x_2.

The key simplifier is that for most static analyses it makes no difference whether the control flow followed the "then" or the "else" branch: both branches need to be covered. Therefore the ϕ function has a very simple semantics in the context of the analysis.

To see how SSA can assist in program optimization lets examine the following code snippet [17]:

Listing 6: Reaching definition

```
y = 1
x = y
y = 2
z = x + y
```

Listing 7: Reaching definition with SSA Form

```
y1 = 1
x1 = y1
y2 = 2
z1 = x1 + y2
```

In the simple three address code version, in order for the optimizer to find which value of x and y is being used in each statement that references these registers, reaching definition analysis would have to be performed.

However, in SSA form this decision is instant. This follows from the fact that use-def chains in SSA contain a single element.

In the context of static analysis, and more specifically pointer analysis, SSA form may be used to increase the precision of the analysis that is performed by making it flow sensitive, without needing to change the prexisting logic of the analysis [3].

2.4 Doop Framework

Static analysis is the analysis of software through examination of its source code, without executing it. This is in contrast to dynamic analysis, which is performed at runtime. There are many types of static analysis; one of them is Pointer Analysis.

Pointer analysis attempts to answer the following question: what is the set of objects that a variable may point to, under all possible executions? The solution to this problem provides us with a static model of the heap that can, in turn, be used in other static analyses, as well as in code optimisation. A simple example of the value of static analysis in code optimisation is the following:

Listing 8: Constant propagation with pointers

```
1   z = 1
2   p = &z
3   *p = 2
4   k = z + 4
```

In order for the optimizer to perform constant propagation on the above snippet, it must first perform pointer analysis on the code, albeit a simple one in our example, through which it will undestand that *p points to z*, and will propagate the updated value of z in the addition expression at line **(4)** [13].

Many Pointer Analysis tools and frameworks have been developed. For the rest of this section, we will specifically discuss Doop, since we aim to extend its functionality. Doop is a static analysis framework for Java programs, emphasising Pointer Analysis.

The various pointer analysis algorithms are expressed in Datalog, a declarative logic programming language, and more specificaly in the Souffle and LogiQL dialects [4]. Datalog is a syntactic subset of Prolog, with one of the main differences being the lack of functional symbols, which has the implication that domains remain finite. This fact, in conjunction with some other important semantic properties of the Datalog programs (stratified negation, total domain ordering) guarantee that all Datalog programs terminate.

It should be noted, however, that extensions that have been made in both the Souffle [8, 11] and LogiQL [10] dialects (e.g., the ability to extend the domain with the introduction of functional predicates), make the language Turing equivalent and thus program termination can no longer be guaranteed [8, 11].

The first step that Doop needs to take is to transform the input program into a set of relations that will be given as input to the Datalog program. For this purpose, Doop utilises the Soot Framework [9] which, among other purposes, can be used to generate facts in Jimple, a typed 3-address intermediate representation language. Doop can also handle Android programs with the use of Dexpler, the Android submodule of Soot.

These facts are then given as input to the analysis algorithm, expressed in Datalog. In the case of Souffle, the Datalog specification is first compiled into an equivalent C++ program. Assuming the relations *Alloc(var, heap, meth)*, *Move(to, from)* that represent heap allocations and moves respectively, as well as the self-descriptive relation *VarPointsTo(var, heap)*, a simple Datalog pointer analysis algorithm would be the following transitive closure computation:

Listing 9: Simple Datalog pointer analysis rules

```
VarPointsTo(to, heap) :- Move(to, from), VarPointsTo(from, heap).
VarPointsTo(to, heap) :- Alloc(to, heap, _).
```

The input facts (e.g. *Move, Alloc*) define the Extensional Database (EDB), whereas the facts that are produced by our rules (e.g. *VarPointsTo*), define the Intentional Database (IDB).

Even though the above example might seem simple, Datalog has been proven very valuable in the field of program analysis and especially pointer analysis. This can be attributed to two factors:

1. Mutual recursion, which is extremely prevalent in pointer analysis, is a lot easier to express in logic programming languages and especially Datalog, as the order of execution of the various rules does not matter, due to its fully declarative nature.
2. Datalog is very close to formal logic, which allows researchers to reason about their programs at almost the same level as the underlying mathematical formulations.

3. SSA TRANSFORMATION

3.1 Introduction

In this section, we will provide an algorithm in Datalog for transforming a set of input facts generated by the DOOP front end of Doop, into SSA Form. First we will overview some basic concepts that are important for the understanding of our algorithm, and then will go over the core Datalog rules of our SSA Transformer.

3.2 Basic concepts

3.2.1 Basic Blocks and Control Flow Graphs

In the context of compiler design [1], basic blocks are maximal code sequences that are contiguous and which are consistent with the following constraints:

- Control flow may only enter at the first instruction of a basic block.
- Control flow may not exit the block by halting or branching, with the exception of its last instruction.

Once our program has been partitioned into basic blocks, we can proceed to the construction of the control flow graph.

Control Flow Graphs are a form of program representation with graph notation. The nodes are the basic blocks of the program and the edges represent the control flow between the blocks. More specificaly, the control flow graph contains an edge from basic block A to basic block B, if, and only if, the last instruction of A can be followed by the first instruction of B. In this case, we say that A is a predecessor of B and that B is a successor of A.

It is common in literature that two additional basic blocks are added, called START and EXIT [1]. These do not contain any executable instructions and are used to represent the starting and exit points of our program respectively. For any control flow graph:

- There is a single edge connecting the START node with the basic block that contains the first instruction of the program.
- If a basic block B contains a possible terminal instruction, then there is an edge from B to EXIT

Let's give an example:

Listing 10: A simple code segment

```
1    x = 15
2    y = 1
3    if (x < 1) goto (8)
4    y = y + 1
5    x = x - 1
6    if (x > 0) goto (3)
7    goto (9)
8    print y
9    end program
```

The Control Flow Graph of the above program can be seen below:

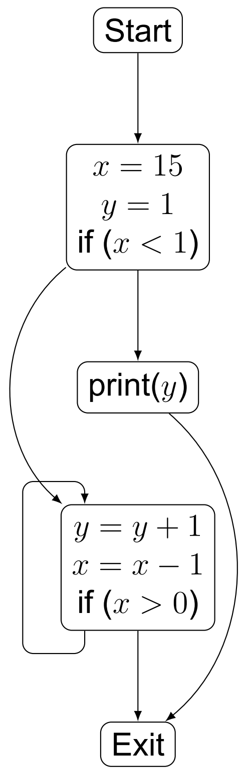

Figure 5: Example of a control flow graph for listing 10

3.2.2 Dominators and related concepts

We will now define a series of relations that are important for the SSA Transformation algorithm.

All the relations defined below are related to Control Flow Graphs:

- We say that n *dominates* m, and we write $n\ dom\ m$, if, and only if, every path from the START node to m goes through n. It is easy to observe that the dominace relation is reflexive and transitive.
- We say that n *strictly dominates m*, and we write $n\ sdom\ m$, if, and only if, n dominates m and n is not equal to m
- Finally, we define the *dominance frontier* of a node x, $DF(x)$, as the set of all nodes n such that x dominates a predecessor of n, but n does not strictly dominate n [2].

The latter can be expressed more formally and compactly using set notation:

$$DF(x) = \{n \mid (\exists m \in Pred(n) : x\ dom\ m)\ \wedge\ \neg(x\ sdom\ n)\}$$

Figure 6: Defintion of *dominance frontier*

As we will see, the dominance frontier will prove to be extremely valuable for our algorithm.

3.3 The SSA transformation algorithm

We are now ready to introduce an algorithm that performs the SSA Transformation. We will give a high level overview of the algorithm and then go over the core Datalog rules of our implementation.

3.3.1 Overview

When translating to SSA Form, one would follow a process that resembles the following steps:

1. Rename the left hand side of every assignment to use unique variable names (e.g. by subindexing).
2. Place ϕ functions wherever they are needed.
3. Update all instructions to use the new variables that were created, either due to the renaming of step 1, or due to the introduction of new assignments due to the ϕ functions of step 2.

Regarding these steps:

- Step 1 is relatively straight forward, as it only involves variable renaming and we only need to make sure that the new identifiers are unique.
- Step 2 is the crux of the transformation. There are multiple algorithms for determining where to place ϕ functions. For example, one could naively introduce ϕ functions at each confluence point of our CFG, even though variable merging might not be actually needed.
- Step 3 requires some analysis to find the reaching definition of each variable (variable versions) at every program point.

Translation of a procedure to an SSA form with a minimal number of ϕ functions can be accomplished with the use of dominace frontiers [2].

We expand the definition of the dominance frontier to sets of control flow graph nodes, as the union of the dominance frontiers of each node in S. Formally:

$$DF(S) = \bigcup_{x \in S} DF(x)$$

We also define the *iterated dominace frontier* $DF^{+}(\ )$ as [2]:

$$DF^{+}(S) = \lim_{i \to \infty} DF^{i}(S)$$

where

$$DF^{1}(S) = DF(S)$$
$$DF^{i+1}(S) = DF(S \cup DF^{i}(S))$$

Figure 7: Definition of *iterated dominance frontier*

If S is the set of nodes that contain assignments to variable x, then $DF^{+}(S)$ is exactly the set of nodes where a ϕ function for x is needed [2].

Applying the SSA translation as defined above on the control flow graph of Figure 5 would yield the following CFG:

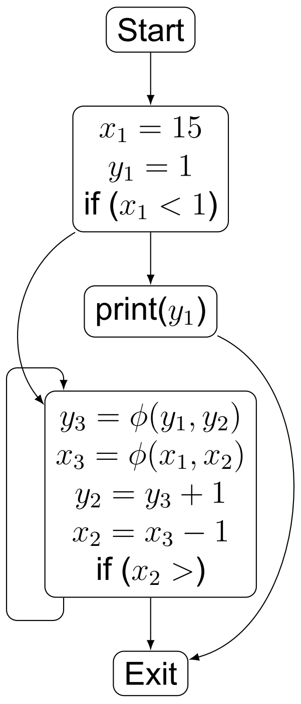

Figure 8: Control Flow Graph of Figure 5 after SSA translation.

3.4 Datalog rules for SSA Transformation

Having covered the basics of the SSA translation, we will now provide a series of Datalog rules that implement the core logic of the transformation. In the following rules, there will be some Datalog predicates that will not be explained in depth. These are either self-explanatory or already implemented in the DOOP framework.

We will split the rules into sections that follow the steps we described above for SSA translation. However, more steps might be needed, since the above process is not extensive and does not cover some technicalities.

3.4.1 Variable renaming

SSA Form requires that the LHS of every assignment is a unique version of the original variable. In the examples we provided above this was accomplished by subindexing the destination variable of each assign statement with a unique index. The Datalog rule that implements the renaming is the following:

Listing 11: Datalog rule for variable renaming

```
SSA_AssignToOriginal(?instr, ?to),
SSA_Alias(?ssa_name, ?to),
SSA_AssignDetails(?instr, ?to, ?index, ?ssa_name, ?method) :-
   Instruction_Index(?instr, ?index),
   AssignInstruction_To(?instr, ?to),
   Instruction_Method(?instr, ?method),
   ?ssa_name = cat(?to, cat("_", to_string(?index))).
```

The SSA_AssignDetails predicate associates an assignment instruction (?instr), relevant info (?index, ?method) and the original target variable (?to) with a new target variable (?ssa_name), that is a unique renaming of the original, by following the scheme:

$$?ssa\_name = ?to\_?index$$

In essence, our new variable name is the original with an underscore and the instruction's index appended to it. This is indeed unique, thus our renaming is valid, and satisfies the SSA property. The reason we choose this renaming strategy over the seemingly simpler increasing counter appoach that is usually found in the literature, is purely a matter of ease of implementation, since the index is immediately available and no further computation is required.

As for the other two predicates in the head of the rule, SSA_Alias keeps track of the various renamings of an original variable (e.g. (x_1, x), (x_2, x)), while SSA_AssignToOriginal, is merely a syntatic convenience. Both of these predicates are quite useful and are used extensively in the rest of our logic.

This rule covers almost all assignment instructions, due to the AssignInstruction_To predicate. However, there are some extra cases that need to be accounted for, more specifically:

- Instance field loads (LoadInstanceField)
- Static field loads (LoadStaticField)
- Array loads (LoadArrayIndex)
- Return value assignment (AssignReturnValue)
- Exception variables of handlers (ExceptionHandler)

However, the handling of these cases is almost identical, so we will not include the relevant rules.

We also rename the method parameters, following the following renaming scheme:

$$?ssa\_param = ?param\_0$$

For completeness, we include the relevant rules without further explaination due to their simplicity:

Listing 12: Datalog rules for this and formal parameter renaming

```
// Method formal params.
SSA_Alias(cat(?original, cat("_", "0")), ?original),
SSA_FormalParam(?index, ?method, cat(?original, cat("_", "0"))) :-
   FormalParam(?index, ?method, ?original).

// Method this var.
SSA_Alias(cat(?original, cat("_", "0")), ?original),
SSA_ThisVar(?method, cat(?original, cat("_", "0"))) :-
   ThisVar(?method, ?original).
```

With the above rules we have produced unique names for each assignment instruction in our program, thus satisfying one of the two basic properties of SSA Form.

3.4.2 Placement of ϕ functions

Since our implementation uses the DF^+ relation (Figure 7) to decide where to place ϕ functions, we will first need to define some basic predicates for *strict dominance*, *dominance frontiers* and *iterated dominance frontiers*. In the following rules, we will use the *PredecessorBB* [1] predicate which provides us with basic block predecessor information and the *Dominates* predicate which represents the dominance relation between basic blocks, both of which are already implemeneted in the Doop framework.

For the *strict dominance* relation, we have the following straightforward implementation:

Listing 13: Datalog rule for strict dominance relation

```
// Simple implementation of the mathematical definition of strict
// domination.
StrictlyDominates(?dominator, ?block) :-
   Dominates(?dominator, ?block),
   ?dominator != ?block.
```

For our dominance frontier rule, we just translate our mathematical definition of Figure 6 in Datalog. The transition is seamless, since our definition was given as a first order logic formula:

Listing 14: Datalog rule for the dominance frontier

```
DominanceFrontier(?dBlock, ?block) :-
   PredecessorBB(?pred, ?block),
   Dominates(?dBlock, ?pred),
   !StrictlyDominates(?dBlock, ?block).
```

Having defined the above rules, we are ready to provide the Datalog rule that computes the *iterated dominance frontier*. This calculation is performed on sets of variables, where each set, e.g. $S_x \subseteq V$, contains all the nodes that contain an assignment to variable x in the original program. The rules are as follows:

Listing 15: Datalog rule for the iterated dominance frontier

```
// Base case
DFPlus(?dBlock, ?block, ?var) :-
   ContainsAssignment(?dBlock, ?var),
   DominanceFrontier(?dBlock, ?block).

// Recurse
DFPlus(?block, ?dfblock, ?var) :-
   DFPlus(?block, ?block_1, ?var),
   DominanceFrontier(?block_1, ?dfblock).
```

Following our definition of the *iterated dominance frontier* (Figure 7), the first rule implements the base case (DF^1) while the second rule implements the recursive definition of DF^{i+1}.

These two Datalog rules provide a succinct solution to this fixpoint computation. Imperative approaches that are provided in the literature [2] are harder to follow, since the implementation language is much different than the mathematical language in which the algorithm was described.

[1]In the actual implementation, the predicate is named MayPredecessorBBModuloThrow

Now that we know where functions are needed, we can start producing the ϕ assignment instructions. The first step in this process is to create a unique LHS for the assignment, which is accomplished with the help of the PhiAssign rule:

Listing 16: Datalog rule for phi assignments

```
SSA_Alias(?phi_var, ?var),
PhiAssign(?phi_var, ?block) :-
   DFPlus(_, ?block, ?var),
   Instruction_Index(?block, ?index),
   ?phi_var = cat(cat(?var, "_phi_"), to_string(?index)).
```

PhiAssign tells us that a ϕ assignment for the original variable $?var$ with LHS $?phi\_var$ will be placed at the start of $?block$. The naming scheme for the new variable that is introduced is:

$$?phi\_var = ?var\text{\_phi\_}?index$$

However, in order to produce the actual ϕ assignment instructions, more information is required. Specifically, we need to know the *reaching definitions* at the start of the basic block where the ϕ assigment will be placed. First we introduce the ReachingDef rules, on a case by case basis:

Listing 17: Datalog rule for ReachinDef: Case 1

```
// Case 1
ReachingDef(?insn, ?ssa_var, ?origin) :-
   BasicBlockBegin(?insn),
   ExistsDefInPhiHeader(?insn, ?ssa_var, ?origin).
```

If $?insn$ is the first instruction in a basic block and a ϕ assignment for $?origin$ exists in the Phi Header, then that is the reaching definition for $?origin$ at $insn$. A *Phi Header* is the set of ϕ assignments at the start of a basic block.

Listing 18: Datalog rule for ReachinDef: Case 2

```
// Case 2
ReachingDef(?insn, ?ssa_var, ?origin) :-
   BasicBlockOutDefs(?pred, ?ssa_var, ?origin),
   PredecessorBB(?pred, ?insn),
   BasicBlockBegin(?insn),
   !ExistsDefInPhiHeader(?insn, _, ?origin).
```

If $?insn$ is the first instruction in a basic block, there is an out def $?ssa\_var$ for $?origin$ in a predecessor and there is no ϕ assignment for $origin$ in the Phi Header, then $?ssa\_var$ is the reaching definition for $origin$.

Note that $?ssa\_var$ must be the same for all predecessor blocks. Otherwise a ϕ instruction would be present and case 1 would hold instead.

Listing 19: Datalog rule for ReachinDef: Case 3

```
// Case 3
ReachingDef(?insn, ?ssa_var, ?origin) :-
   ReachingDef(?prev, ?ssa_var, ?origin),
   PrevInSameBasicBlock(?insn, ?prev),
   !SSA_AssignToOriginal(?prev, ?origin).
```

If $?ssa\_var$ is the reaching defintion for $origin$ at the previous instruction and the previous instruction is not an assignment to $?origin$, then $?ssa\_var$ is the reaching definition at $?insn$ for $origin$.

Listing 20: Datalog rule for ReachinDef: Case 4

```
// Case 4
ReachingDef(?insn, ?ssa_var, ?origin) :-
   PrevInSameBasicBlock(?insn, ?prev),
   SSA_AssignDetails(?prev, _, _, ?ssa_var, _),
   SSA_Alias(?ssa_var, ?origin).
```

If the previous instruction is an assignment to $?origin$ with LHS $?ssa\_var$, then $?ssa\_var$ is the reaching definition at $?insn$ for $origin$.

Listing 21: Datalog rule for ReachinDef: Case 5

```
// Case 5
ReachingDef(?insn, ?ssa_var, ?origin) :-
   Instruction_Index(?insn, 1),
   Instruction_Method(?insn, ?method),
   (SSA_FormalParam(_, ?method, ?ssa_var);
   SSA_ThisVar(?method, ?ssa_var)),
   SSA_Alias(?ssa_var, ?origin),
   !ExistsDefInPhiHeader(?insn, _, ?origin).
```

Finally, if $?insn$ is the first instruction in $?method$, there is a formal parameter or "this" var $?ssa\_var$ for $origin$ and there is no ϕ assignment for $origin$ in the Phi Header, then $?ssa\_var$ is the reaching definition for $origin$.

With the above rules, we have computed the reaching definitions at every instruction.

For completeness we include the definition of some helper predicates:

Listing 22: Datalog rule for ExistsDefInPhiHeader

```
ExistsDefInPhiHeader(?insn, ?ssa_var, ?origin) :-
   BasicBlockHead(?insn, ?head),
   PhiAssign(?ssa_var, ?head),
   SSA_Alias(?ssa_var, ?origin).
```

Listing 23: Datalog rules for BasicBlockOutDefs

```
BasicBlockOutDefs(?block, ?ssa_var, ?origin) :-
   ReachingDef(?end, ?ssa_var, ?origin),
   BasicBlockTail(?block, ?end),
   !SSA_AssignToOriginal(?end, ?origin).

BasicBlockOutDefs(?block, ?ssa_var, ?origin) :-
   BasicBlockTail(?block, ?end),
   SSA_AssignDetails(?end, _, _, ?ssa_var, _),
   SSA_Alias(?ssa_var, ?origin).
```

Finally, we include the definition of InDefs, which tells us which definitions are reaching the start of a basic block, above the PhiHeader. This is the union of the out defs of all predecessors and the formal parameters in the special case of the first instruction of a method. The following rules implement exactly that:

Listing 24: Datalog rules for InDefs

```
InDefs(?insn, ?ssa_var, ?origin) :-
   BasicBlockOutDefs(?pred, ?ssa_var, ?origin),
   PredecessorBB(?pred, ?insn),
   BasicBlockBegin(?insn).

InDefs(?insn, ?ssa_var, ?origin) :-
   Instruction_Index(?insn, 1),
   Instruction_Method(?insn, ?method),
   (SSA_FormalParam(_, ?method, ?ssa_var);
   SSA_ThisVar(?method, ?ssa_var)),
   SSA_Alias(?ssa_var, ?origin).
```

We are now ready to provide the Datalog rule that introduces the actual ϕ assignments. It should be noted however, that since our intermediate representation does not have an instruction for ϕ assignments, we will emulate them by producing multiple virtual assignments. These are refered to as PhiPseudoAssign instructions in our program. For example:

```

   $x_3 = \phi(x_1, x_2)$
```

```
   $x_3 = x_1$
   $x_3 = x_2$
```

This is semantically incorrect from an execution standpoint. However, since our end goal is performing pointer analysis, this is a simple solution that does not require the modification of our intermediate language. Alternatively, one could introduce a special ϕ instruction for this purpose, similar to a Move instruction [3].

Having made this clarification, here is the rule for calculating the PhiPseudoAssign instructions:

Listing 25: Datalog rule for virtual phi assignments

```
PhiPseudoAssign(?phi_var, ?phi_arg, ?dst_block, ?pseudoname) :-
   PhiAssign(?phi_var, _, ?dst_block),
   InDefs(?dst_block, ?phi_arg, ?origin_var),
   SSA_Alias(?phi_var, ?origin_var),
   ?pseudoname = cat(?phi_var, cat("_", ?phi_arg)).
```

The logic is straightforward: If we have decided that a PhiAssign will be placed at the start of $?dst\_block$ with LHS $?phi\_var$, and there is an InDef $?phi\_arg$ such that $?phi\_var$ and $?phi\_arg$ are versions of the same original variable $?origin\_var$, then create a virtual assignment of the form:

$$?phi\_var = ?phi\_arg$$

Pseudoname is just a unique identifier for the virtual ϕ assignments and does not have any special meaning.

3.4.3 Instruction ordering

Having created all virtual assignments for our ϕ instructions, the next step is to "connect" them with the rest of the instuctions. More specifically, we need to create a new ordering on our instructions, since the introduction of the new instructions has invalidated any previous ordering, and any instruction indexes that are used are no longer correct.

First we define some concepts used in our algorithm:

- A PhiChunk consists of a set of contiguous virtual ϕ assigments (PhiPseudoAssign) to the same LHS variable.
- A PhiHeader of a block B consists of a set of contiguous PhiChunks that will be placed at the start of B.

To order the instructions, we will calculate a new predicate SSANext(?prev, ?next) which states the obvious: instruction $?next$ is the sucessor of $?prev$. After this, the calculation of the new instruction indexes becomes a trivial task.

In our calculation of SSANext, we will follow a bottom-up approach:

1. We order and connect the virtual ϕ assignments, to form PhiChunks.
2. We order and connect the PhiChunks, to form PhiHeaders.
3. We connect the PhiHeaders with the rest of the instructions.

Let us define the following ordering predicates for our own convenience:

```
ExistsPseudoAssignBetween(?prev, ?next) :-
   PhiPseudoAssign(?phi, _, ?block, ?prev),
   PhiPseudoAssign(?phi, _, ?block, ?next),
   PhiPseudoAssign(?phi, _, ?block, ?mid),
   ord(?prev) < ord(?next),
   ord(?prev) < ord(?mid),
   ord(?mid) < ord(?next).
```

A virtual assignment $?mid$ exists between $?prev$ and $?next$, if all of them are virtual assignments to the same variable $?phi$, are in the same block and it holds that:

$$ord(?prev) < ord(?mid) < ord(?next)$$

Ord [8] is a built-in ordering functional symbol for strings that is included in Souffle. The details of this ordering are unimportant, other than the fact that

$$s_1 \neq s_2 \Rightarrow ord(s_1) \neq ord(s_2)$$

We also define a similar predicate for PhiChunks:

```
ExistsPhiChunkBetween(?prev, ?next) :-
   PhiPseudoAssign(?prev, _, ?block, _),
   PhiPseudoAssign(?next, _, ?block, _),
   PhiPseudoAssign(?mid, _, ?block, _),
   ord(?prev) < ord(?next),
   ord(?prev) < ord(?mid),
   ord(?mid) < ord(?next).
```

We are now ready to connect our virtual assignments and form PhiChunks. The follwing rule implements exactly that:

Listing 26: Datalog rule for ordering the virtual phi assignments in a PhiChunk

```
SSANext(?prev, ?next),
PhiChunkNext(?prev, ?next) :-
   PhiPseudoAssign(?chunk, _, ?block, ?prev),
   PhiPseudoAssign(?chunk, _, ?block, ?next),
   ord(?prev) < ord(?next),
   !ExistsPseudoAssignBetween(?prev, ?next).
```

If $?prev$ and $?next$ are both virtual ϕ assignments, it holds that $ord(?prev) < ord(?next)$ and there is no virtual assignment between them then $?next$ follows $?prev$ inside the PhiChunk (PhiChunkNext) as well as in the global ordering of the instructions (SSANext).

Next we provide two rules that are related to the structure of the PhiChunks, specifically their start and end instructions:

```
PhiChunkStart(?phi_chunk, ?start) :-
   PhiPseudoAssign(?phi_chunk, _, _, ?start),
   !PhiChunkNext(_, ?start).
```

A virtual assignment $?start$ is the first instruction of it's $?phi\_chunk$, if there is no previous instruction in the PhiChunk.

```
PhiChunkEnd(?phi_chunk, ?end) :-
   PhiPseudoAssign(?phi_chunk, _, _, ?end),
   !PhiChunkNext(?end, _).
```

A virtual assignment $?end$ is the last instruction of its $?phi\_chunk$, if there is no instruction following it in the PhiChunk.

Now we will provide the rule that orders the PhiChunks and forms PhiHeaders:

Listing 27: Datalog rule for ordering the PhiChunks in a PhiHeader

```
SSANext(?prev_end, ?next_start),
PhiHeaderNext(?prev, ?next) :-
   PhiPseudoAssign(?prev, _, ?phi_header, _),
   PhiPseudoAssign(?next, _, ?phi_header, _),
   PhiChunkEnd(?prev, ?prev_end),
   PhiChunkStart(?next, ?next_start),
   ord(?prev) < ord(?next),
   !ExistsPhiChunkBetween(?prev, ?next).
```

The logic is quite similar to the rule that handles virtual assignments and forms PhiChunks. The main differnece is in the way SSANext is handled. More specifically, if we decided that PhiChunks $?prev$ and $?next$ are ordered in their containing PhiHeader as their name suggests, then the end instruction of $?prev$ ($?prev\_end$) is followed by the start instruction of $?next$ ($?next\_start$) in the global ordering of the instructions.

Building on top of the PhiHeaderNext predicate we define the following rules:

Listing 28: Datalog rules for finding the start and end instructions of PhiHeaders

```
PhiHeaderStart(?phi_header, ?start) :-
   PhiPseudoAssign(?start_chunk, _, ?phi_header, _),
   PhiChunkStart(?start_chunk, ?start),
   !PhiHeaderNext(_, ?start_chunk).

SSANext(?end, ?phi_header),
PhiHeaderEnd(?phi_header, ?end) :-
   PhiPseudoAssign(?end_chunk, _, ?phi_header, _),
   PhiChunkEnd(?end_chunk, ?end),
   !PhiHeaderNext(?end_chunk, _).
```

These rules calculate the first and last instructions of a PhiHeader respectively. In the calculation of PhiHeaderEnd, we additionally connect the $?end$ instruction of the PhiHeader with the first instruction of the block, which is $?phi\_header$. This is due to the fact that in our implementation a PhiHeader construct is uniquely identified by the head instruction of the basic block on top of which it is placed.

So far we have linked all instructions locally inside the PhiHeaders, as well as the last instructions of the PhiHeaders with the head of their block. All that is left to do is take care of the rest of the instructions inside the basic blocks. This is handled by the following two rules:

```
SSANext(?prev, ?next) :-
   Instruction_Next(?prev, ?old_next),
   PhiHeaderStart(?old_next, ?next).

SSANext(?prev, ?next) :-
   Instruction_Next(?prev, ?next),
   !PhiHeaderStart(?next, _).
```

The cases they cover are the following:

1. If $?prev$ is followed by $?old\_next$ in the original ordering and $?old\_next$ was the head instruction of the block for which we calculated a PhiHeader with first instruction $?next$, then link $prev$ and $next$.
2. If the $?prev$ is followed by $?next$ in the original ordering and either $?next$ is the head of a block that does not have a PhiHeader (thus was not modified locally with virtual assignments) or is not the head of a block, then the original ordering still holds.

Our ordering of the instructions is now complete. We can now proceed with the calculation of the new instruction indexes:

Listing 29: Datalog rules for instruction indexing

```
SSA_InstructionIndex(?insn, 1) :-
   isInstruction(?insn),
   !SSANext(_, ?insn).

SSA_InstructionIndex(?insn, ?index + 1) :-
   SSA_InstructionIndex(?prev, ?index),
   SSANext(?prev, ?insn).
```

This calculation is trivial; we start from the first instructions of each method, giving them index 1, and we recursively calculate the rest of the indexes by following SSANext.

We also provide two useful rules, NewIndexMapping and NewBBStart. The first one provides us with a mapping between old and new instruction indexes, while the second one gives us the index of the new basic block head instruction, since ϕ nodes may have changed it.

```
NewIndexMapping(?old, ?new, ?method) :-
   Instruction_Index(?insn, ?old),
   Instruction_Method(?insn, ?method),
   SSA_InstructionIndex(?insn, ?new).
```

Listing 30: Datalog rules to calculate new basic block start index

```
NewBBStart(?old, ?new, ?method) :-
   NewIndexMapping(?old, ?new, ?method),
   Instruction_Index(?insn, ?old),
   Instruction_Method(?insn, ?method),
   BasicBlockBegin(?insn),
   !PhiHeaderStart(?insn, _).

NewBBStart(?old, ?new, ?method) :-
   Instruction_Method(?insn_old, ?method),
   Instruction_Index(?insn_old, ?old),
   PhiHeaderStart(?insn_old, ?insn_new),
   SSA_InstructionIndex(?insn_new, ?new).
```

3.4.4 Output fact generation

For the final step of our transformation algorithm, we need a rule per input relation. These rules will take care of merging all the information that was calculated in the previous steps, such as reaching variable definitions and new instruction indexes and produce the output facts.

Let's review the rule that is responsible for rewriting local assignments:

Listing 31: Datalog rules AssignLocal fact rewritting

```
RewriteAssignLocal(?insn, ?index, ?to, ?from, ?inmethod) :-
   isAssignLocal_Insn(?instr),
   SSA_AssignDetails(?insn, _, _, ?to, ?inmethod),
   SSA_InstructionIndex(?insn, ?index),
   AssignLocal_From(?insn, ?from_origin),
   ReachingDef(?insn, ?from, ?from_origin).

RewriteAssignLocal(?insnid, ?index, ?to, ?from, ?inmethod) :-
   PhiPseudoAssign(?to, ?from, ?block, ?insn),
   Instruction_Method(?block, ?inmethod),
   SSA_InstructionIndex(?insn, ?index),
   ?insnid = cat(?inmethod, cat("/phiassign/instruction",
      to_string(?index))).
```

The basic idea is the following: If $?insn$ is a local assign, and we decided that $?to$ is the new SSA name for the LHS, we have a new $?index$, and the RHS $?from$ is the reaching definition at $?insn$ for the original RHS $?from\_origin$, then generate an output fact for local assignments (RewriteLocalAssign) that combines all this information.

The second rule takes care of the virtual ϕ assignments that we introduced, which are local assignments in our IR.

Finally, lets review the rewrite rule for a branch instruction and specifically for an *If* instruction:

Listing 32: Datalog rules for If fact rewritting

```
RewriteIf(?insn, ?index, ?to, ?inmethod) :-
   isIf_Insn(?insn),
   SSA_InstructionIndex(?insn, ?index),
   Instruction_Method(?insn, ?inmethod),
   If_Target(?insn, ?old_to),
   NewBBStart(?old_to, ?to, ?inmethod).
```

The core logic is the same. However, since *If* instructions have the index of the jump target, we need to take the updated index, which is done with the $NewBBStart$ predicate. Note that using NewIndexMapping would not be correct in the general case, since the jump targets are essentially the first instruction in a basic block and these may have changed due to the introduction of ϕ nodes.

This concludes our SSA transformation algorithm. The rest of the fact rewrite rules will be elided, however the two cases we covered should be more than enough to provide the reader with the general pattern these rules follow.

4. TYPE INFERENCE

4.1 Introduction

In this section, we will provide a type inference algorithm in Datalog. The motivation behind the development of such an algorithm is the lack of explicit type information in the Dalvik bytecode format.

We start by going over some fundamental concepts, followed by a high level overview of our algorithm. Finally, we will provide the core Datalog rules of our implementation.

4.2 Fundamental concepts

In this section we will provide an overview of the core mathematical structures that are related to our algorithm and type inference in general. Understanding these concepts, at least intuitively, is important for the understanding of the main ideas in our algorithm.

4.2.1 Types and related structures

Let T be the sets of all types. In the scope of our algorithm one may think of this set as the set containing types that are found in the input program.

Let $\leq$ be a binary relation over T, such that:

$$t_1 \leq t_2 \Leftrightarrow t_1 \text{ is a subtype of } t_2$$

The $\leq$ binary relation has the following properties:

1. $\forall t \in T : \ t \leq t$ **(reflexivity)**
2. $\forall t_1, t_2 \in T : \ t_1 \leq t_2 \wedge t_2 \leq t_1 \Rightarrow t_1 = t_2$ **(antisymmetry)**
3. $\forall t_1, t_2, t_3 \in T : \ t_1 \leq t_2 \wedge t_2 \leq t_3 \Rightarrow t_1 \leq t_3$ **(transitivity)**

Thus, $\leq$ over T forms a partial order, called the *Partial Order of Types*.

Finally T has a least element $\bot$ and greatest element $\top$. More formally these two elements have the following properties:

1. $\forall t \in T : \ t \leq \top$
2. $\forall t \in T : \ \bot \leq t$

In the context of Java programs, and more specifically reference types, $java.lang.Object$ is the $\top$ element and the null type (type of the $null$ contant) is the $\bot$ element.

A useful visualization for partially ordered sets are *Hasse diagrams*. For example, the following class hierarchy

```java
class A {}

interface I1 {}
interface I2 {}

class B implements I1, I2 {}
class C implements I1, I2 {}
```

can be visualized with the following Hasse diagram:

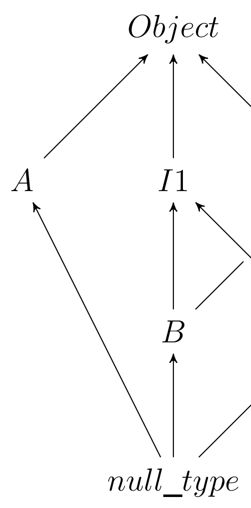

Figure 9: The Hasse diagram of a simple type hierarchy

A *typing* t is a function $t : V \rightarrow T$, that maps variables to types. Any type inference for local variables aims to compute a *valid typing*.

A *valid typing* is a typing that is consistent with a set of constraints that the program instructions impose on the variables they use and define. In our alogrithm, these constraints take the form of subtype/supertype constraints.

4.2.2 Common poset operations

Two operations that are useful when dealing with posets are the *Least Common Ancestor* and the *Greatest Common Predecessor*.
Let $(P, \leq)$ be a poset. The *Least Common Ancestor* is a function $lca : P \times P \rightarrow 2^P$, such that:

$$lca(x, y) = minimal(z|x \leq z \wedge y \leq z)$$

where $minimal(\cdot)$ are the minimal elements of a poset.
More formally, let $(P, \leq)$ be a poset. The minimal elements of $S \leq P$ is the set:

$$\{x| \nexists y \in S : x \leq y\}$$

Similarly, we define the $GreatestCommonDecendant$ to be a function $gcd : P \times P \rightarrow 2^P$, such that:

$$gcd(x, y) = maximal(z|z \leq x \wedge z \leq y)$$

where $maximal(\cdot)$ are the maximal elements of a poset:

$$\{x| \nexists y \in S : y \leq x\}$$

4.2.3 Program instructions and constraints

Program instructions impose upper and lower bounds to the static types of variables that are involved. Lets consider the following code segment, written in a Java-like language:

```
<untyped> x;
x = new A();
f(x); // void f(B)
```

From the snippet above, we can infer that the static type of x must be a supertype of A. Thus, if t is the typing that our type inference algorithm calculcates, then it must hold that $t(x) \geq A$. Additionally, the static type of b must be a subtype of B, since f takes an argument of type B: $B \leq t(x)$.

Similar constraints are imposed by any instruction that uses or defines variables. These constraints can be expressed compactly in the form of inference rules:

$$\frac{x := new\ T();}{T \leq t(x)}\ \text{HeapAlloc} \qquad \frac{[f(\ldots) : T_0] \quad x := f(\ldots);}{T_0 \leq t(x)}\ \text{AssignReturn}$$

$$\frac{[f(T_1, T_2, \ldots, T_n) : T_0] \quad f(x_1, x_2, \ldots, x_n);}{t(x_1) \leq T_1,\ t(x_2) \leq T_2,\ \ldots,\ t(x_n) \leq T_n}\ \text{ActualParam}$$

$$\frac{[f(T_1, T_2, \ldots, T_n) : T_0] \quad f(x_1, x_2, \ldots, x_n) : T_0\{\ldots\}}{T_1 \leq t(x_1) \leq T_1,\ T_2 \leq t(x_2) \leq T_2,\ \ldots,\ T_n \leq t(x_n) \leq T_n}\ \text{FormalParam}$$

$$\frac{catch(T\ x)}{T \leq t(x) \leq T}\ \text{Catch} \qquad \frac{[f(\ldots) : T_0] \quad f(\ldots) : T_0\{\ldots return\ x; \ldots\}}{t(x) \leq T_0}\ \text{AssignReturn}$$

$$\frac{x := y \rightarrow T.f : T_f}{T_f \leq t(x),\ t(y) \leq T}\ \text{InstanceLoad} \qquad \frac{x := T.f : T_f}{T_f \leq t(x)}\ \text{StaticLoad}$$

$$\frac{y \rightarrow T.f : T_f := x}{t(x) \leq T_f,\ t(y) \leq T}\ \text{InstanceStore} \qquad \frac{T.f : T_f := x}{t(x) \leq T_f}\ \text{StaticStore}$$

Figure 10: Subytype constraint inference rules

The above inference rules capture constraints where the lower or upper bound of the constraint is fixed. Instructions where this is not true, like local assignments, could be modeled in a similar fashion:

$$\frac{x := y}{t(y) \leq t(x)}\ \text{LocalAssign}$$

Note that both sides of the constraints are ”unresolved” - they are both a function of the typing t we are trying to compute. This might be fine from a mathematical standpoint however, for reasons related to the implementation of our algorithm, it is convienient that we handle such cases differently.

For this reason, we introduce a new binary relation $\sqsubseteq$, such that:

$$\forall x, y \in V\ x := y \Rightarrow y \sqsubseteq x$$

The inference rules for local assignments are the following:

$$\frac{x \sqsubseteq y \quad t(y) \leq T}{t(x) \leq T}\ \text{LocalAssign Upper} \qquad \frac{y \sqsubseteq x \quad T \leq t(y)}{T \leq t(x)}\ \text{LocalAssign Lower}$$

Figure 11: Infernce rules for local assignments

Intuitively, assignments transfer constraints between the variables: upper bounds of the LHS become upper bounds of the RHS and lower bounds of the RHS become lower bounds of the LHS.

Array loads and stores are handled similarly, with a bit more complicated logic similar to that of instance loads and stores:

$$\frac{x := y[\cdot] \quad t(x) \leq T}{t(y) \leq T[\cdot]}\ \text{ArrayLoad Upper} \qquad \frac{x := y[\cdot] \quad T[\cdot] \leq t(y)}{T \leq t(x)}\ \text{ArrayLoad Lower}$$

$$\frac{y[\cdot] := x \quad t(y) \leq T[\cdot]}{t(x) \leq T}\ \text{ArrayStore Upper} \qquad \frac{y[\cdot] := x \quad t(x) \leq T}{T[\cdot] \leq t(y)}\ \text{ArrayStore Lower}$$

4.3 The type inference algorithm

We will now go over our type inference algorithm. The algorithm could be roughly split into the following steps:

1. For each variable, gather the subtype constraints that the program instructions impose on them.
2. For each variable, combine its constraints to find a set of compatible types.
3. For each variable, output a type for it, by choosing a type from the set that was calculated during step 2.

At the end of this chapter, we will discuss the challenges that primitive types introduce to a subtype-based algorithm like ours (especially for Dalvik bytecode) and the solution we provided to this problem.

4.3.1 Gathering the constraints

The first step of our algorithm consists of finding and encoding the various constraints that the instructions place on the program's variables. As we discussed in the previous section, instructions impose subtyping constraints on the variables involved in them. More specifically, they impose upper and lower bounds on $t(v)$, where t is the typing our algorithm aims to compute.

These constraints have already been expressed in the form of inference rules at Figure 10. Since our algorithm is written in Datalog, the inference rules are almost identical to their

Datalog counterparts, modulo some implementation and intermediate language details.

In order to encode these constraints we will need two predicates:

- UpperBoundTypeForConstraint($?var, ?type, ?instr$): Instruction $?instr$ imposes the constraint: $t(?var) \leq ?type$.
- LowerBoundTypeForConstraint($?var, ?type, ?instr$): Instruction $?instr$ imposes the constraint: $?type \leq t(?var)$.

We will also need two additional predicates:

- UpperConstraintForVar($?var, ?instr$): Instruction $?instr$ imposes an upper bound constraint on variable $?var$.
- LowerConstraintForVar($?var, ?instr$): Instruction $?instr$ imposes an lower bound constraint on variable $?var$.

The usefulness of the the last two predicates will become clear in the next section.

We will provide the implementation of some inference rules from Figure 10. In the following snippets we will use the same intermediate language as the one in our SSA Transformation algorithm. Since the objective of this thesis is both SSA transformation and type inference, we will use the output facts of our SSA algorithm as input, instead of the original ones.

As our first example, lets examine the rule that handles heap allocations:

Listing 33: Datalog rule for HeapAlloc type constraint

```
LowerConstraintForVar(?to, ?insn),
LowerBoundTypeForConstraint(?to, ?type, ?insn) :-
   RewriteAssignHeapAllocation(?insn, _, ?heap, ?to, _, _),
   HeapAllocation_Type(?heap, ?type).
```

Similarly for formal method parameters:

Listing 34: Datalog rule for FormalParam type constraint

```
// Static type of formal params is known.
HasKnownType(cat(?var, "_init"), ?var, ?type) :-
   SSA_FormalParam(_, _, ?var),
   SSA_Alias(?var, ?var_origin),
   Var_Type(?var_origin, ?type).
```

For FormalParams and ThisVars, we use the predicate HasKnownType. This is because we know the type of a method's formal parameters from its signature, thus we use this special predicate to signify this knowledge to the inference engine of our algorithm and avoid unneeded computation.

Note the use of the Var_Type predicate; currently the dex frontend outputs type information for formal parameters but not for other variables. We take advantage of this fact to simplify our rule.

Finally, we will show how local assignments are handled. As we mentioned in a previous segment, we need to introduce the notion of an abstract constraint:

```
Assign(?stmt, ?var1, ?var2) :-
   RewriteAssignLocal(?stmt, _, ?var1, ?var2, _).

AbstractConstraint(?var_sup, ?var_sub) :-
   Assign(_, ?var_sup, ?var_sub).
```

Having defined the rules for abstract constraints, we can now provide the Datalog rules that implement the inference rules specified at Figure 11:

Listing 35: Datalog rule for abstract type constraint

```
LowerConstraintForVar(?var, ?stmt),
LowerBoundTypeForConstraint(?var, ?type, ?stmt) :-
   AbstractConstraint(?var, ?var_other),
   LowerBoundTypeForConstraint(?var_other, ?type, ?stmt).

UpperConstraintForVar(?var, ?stmt),
UpperBoundTypeForConstraint(?var, ?type, ?stmt) :-
   AbstractConstraint(?var_other, ?var),
   UpperBoundTypeForConstraint(?var_other, ?type, ?stmt).
```

The Datalog rules for the rest of the inference rules of Figure 10 follow a similar pattern with the ones above, and are elided for simplicity.

4.3.2 Combining the constraints

Now that all constraints have been encoded, we need to process them in order to compute a set of valid types T_v for each variable v.

Recall that constraints are split into upper $(v \leq \cdot)$ and lower $(\cdot \leq v)$ bound constraints. The way our algorithm works is by computing an Upper Bound Set (U_v) and a Lower Bound Set (L_v) for every variable v. We can then compute T_v as:

$$T_v = \{t | u \in U_v \wedge l \in L_v \wedge l \leq t \leq u\}$$

Intuitively, U_v and L_v give us an upper and a lower bound, such that every type that is ”in between” respects all upper bound constraints on the type of v.

Viewing the computation of T_v from a satisfiability standpoint provides us with a better insight on how to go about combining the constraints. More specifically, let $I_0^v, I_1^v, \ldots I_n^v$ be the instructions that impose some upper bound constraint on v under some arbitrary ordering. Then the upper constraints on a variable v form a logical formula:

$$C_{upper}^v = C_0^v \wedge C_1^v \wedge \ldots \wedge C_n^v$$

where C_i is a disjunction of the upper bound constraints $(v \leq \cdot)$ that I_i imposes on v. In most cases these disjunctions will contain exactly one term. However, our handling of primitives may require up to three terms in the disjunction in some cases. We will expand more on this in the relevant section.

A natural way to construct the upper bound set of v, U_v, is to iterate over the constraints and use the gcd function. Taking into account the CNF formulation above, our apporach is to iterate over the C_i, ”lowering” the upper bound set on each step.

More specifically, let $I_0^v, I_1^v, \ldots I_n^v$ be an ordering of the instructions that impose some upper bound constraint on v. With $const(I_i, v)$ we will denote the set of upper bound constraints that I_i imposes on variable v. For example, if we have that $C_i^v = v \leq t_1 \vee v \leq t_2$ then $const(I_i, v) = \{t_1, t_2\}$. We define the iterative process:

$$U_v^0 = const(I_0, v)$$
$$U_v^{i+1} = \bigcup gcd(x, y), \text{where } x \in U_i, y \in const(I_{i+1}, v)$$

Figure 12: Iterative computation of the Upper Bound Set

Then the set U_v is exactly U_v^n.

To implement this in Datalog, one first needs to define an ordering similar to the one defined above. This is achieved with the following rules:

Listing 36: Datalog rules for constraint ordering

```
NotFirstUpperConstraintForVar(?var, ?stmt) :-
   UpperConstraintForVar(?var, ?stmt),
   UpperConstraintForVar(?var, ?stmt2),
   ord(?stmt2) < ord(?stmt).

NotLastUpperConstraintForVar(?var, ?stmt) :-
   UpperConstraintForVar(?var, ?stmt),
   UpperConstraintForVar(?var, ?stmt2),
   ord(?stmt2) > ord(?stmt).

LaterUpperConstraintForVar(?var, ?stmt, ?stmtLater) :-
   UpperConstraintForVar(?var, ?stmt),
   UpperConstraintForVar(?var, ?stmtLater),
   ord(?stmtLater) > ord(?stmt).

NextUpperConstraintForVar(?var, ?stmt, ?stmtNext) :-
   LaterUpperConstraintForVar(?var, ?stmt, ?stmtNext),
   ?stmtNextOrd = min ord(?stmtLater) : LaterUpperConstraintForVar(?var,
      ?stmt, ?stmtLater),
   ord(?stmtNext) = ?stmtNextOrd.

FirstUpperConstraintForVar(?var, ?stmt) :-
   UpperConstraintForVar(?var, ?stmt),
   !NotFirstUpperConstraintForVar(?var, ?stmt).
```

The first two rules are there to make our definition easier:

- A constraint from $?stmt$ for $?var$ is not the first upper constraint if there exists a constraint from some $?stmt2$, such that $ord(?stmt2) < ord(?stmt)$.
- Similarly for NotLastUpperConstraintForVar.

Having defined these, the rest is straightforward. In the following explanation, we will assume that $?var$ is fixed:

- A constraint $?stmt$ is the first, if there is a constraint and it is not not the first (making use of our auxiliary relation).
- A constraint $?stmtLater$ follows $?stmt$ in the ordering, if it holds that $ord(?stmt) < ord(?stmtLater)$.
- A constraint $?stmtNext$ follows $?stmt$ immediately, if it is after $?stmt$ and has the minimum ord out of all constraints that are after $?stmt$.

After the implementation of the constraint ordering has been provided, we can move to the crux of the algorithm: the implementation of the iterative computation that we described above.

Listing 37: Calculation of upper bound set

```
UpperBoundTypeForAllConstraints(?var, ?type) :-
   HasKnownType(_, ?var, ?type).

UpperBoundTypeUpToConstraint(?var, ?type, ?stmt) :-
   !HasKnownType(_, ?var, _),
   UpperBoundTypeForConstraint(?var, ?type, ?stmt),
   FirstUpperConstraintForVar(?var, ?stmt).

UpperBoundTypeUpToConstraint(?var, ?type, ?stmt) :-
   UpperBoundTypeUpToConstraint(?var, ?typeBefore, ?stmtPrev),
   NextUpperConstraintForVar(?var, ?stmtPrev, ?stmt),
   UpperBoundTypeForConstraint(?var, ?typeCur, ?stmt),
   GreatestCommonDescendant(?typeBefore, ?typeCur, ?type).

UpperBoundTypeForAllConstraints(?var, ?type) :-
   UpperBoundTypeUpToConstraint(?var, ?type, ?stmt),
   !NextUpperConstraintForVar(?var, ?stmt, _).
```

The first rule is just a shortcut: if we are sure about the static type of some variable, like formal parameters, then there is no reason to go through this process. Note how the rest of the rules correspond to the different cases of the process defined at Figure 12. The last rule is the output of the computation: $U_v = U_v^n$.

The rules for the Lower Bound Set L_v follow the same pattern. First the constraints are ordered, then iterated over. The main difference is the use of the lca function instead of the gcd, since we are trying to ”raise” the lower bound.

Having computed both L_v and U_v we may now calculate T_v. This is straightforward, since we have already expressed the relation between T_V and the lower and upper bound sets above:

Listing 38: Calculation of upper bound set

```
TypeCompatibleWithAllConstraints(?var, ?type) :-
   LowerBoundTypeForAllConstraints(?var, ?lowerType),
   UpperBoundTypeForAllConstraints(?var, ?upperType),
   TypeBetween(?upperType, ?lowerType, ?type).
```

We are now ready to output our typing $t(\cdot)$. All that we need to do is choose some type t_v from T_v for each variable v and set $t(v) = t_v$.

The choice method could be an arbitrary (even random) function that selects an element from a set. In our implementation, our choice function selects the type that minimizes $ord(\cdot)$. In Datalog:

Listing 39: Output rule of our type inference rule

```
SSA_TypeOrd(?var, ?minTypeOrd) :-
   ExistsNonGuardType(?var),
   ?minTypeOrd = min ord(?t) : NonGuardCompatibleTypes(?var, ?t).

SSA_Type(?var, ?minType) :-
   isType(?minType),
   ?minTypeOrd = ord(?minType),
   SSA_TypeOrd(?var, ?minTypeOrd).
```

The rule above is a bit more involved due to the special handling of primitives as will be explained in the next section.

The advantage of such an approach, and any deterministic choice function in general, is that it has the following property:

Lemma 4.1 *Let $v_0, v_1, \ldots, v_n$ be variables that are involved in an assignment cycle C. For example:*

$$v_1 = v_2;\ v_2 = v_3;\ v_3 = v_1;$$

If the choice function is deterministic, then $t(v_i) = t(v_j),\ v_i, v_j \in C$.

Intuitively what the above lemma says is that all variables involved in an assignment cycle will get mapped to the same type.

4.3.3 Primitives

So far we have only discussed how our type inference algorithm handles reference types. This is because reference types have clear semantics about subtyping, unlike primitives where such a relation is not well defined.

One could model the subtype relation between primitive types based on which implicit conversions and promotions can be performed. For example, in Java, `int` can be implicitly converted to `long`, thus one may assume that $int \leq long$. This might be a good approach for Java bytecode, however this is not entirely true for Dalvik.

The main problem arises from the fact that the Dalvik VM uses 32-bit registers and 64-bit are represented as register pairs. Thus assigning an `int` to a `long` (to the first register of the pair to be exact), would not pass bytecode verification.

Moreover, 32-bit integer types (boolean, short, int etc) seem to be interchangeable, as many binary and unary operations can be performed to all of them. Finally, the lack of a dedicated *null* constant and the use of 0 instead, causes ambiguity when encoutering assignments such as $v = 0$: under no further context, $t(v)$ could either be a reference type $(t(v) \leq Object)$ or any 32-bit primitive type (assuming that 64-bit primitives have special constant assignment instructions).

It is clear from the previous examples that additional logic needs to be added in order to handle primitives. More specifically, we need to expand our type poset in order to introduce some kind of subtyping relation between the primitives. In our implementation we added two additional posets, ending up with a ”forest”:

$$\begin{tikzcd}
 & & & & prim32_\top & \\
 & & int32_\top \arrow[rru] & & & \\
int \arrow[rru] & short \arrow[ru] & char \arrow[u] & byte \arrow[lu] & boolean \arrow[llu] & float \arrow[luu] \\
 & & int32_\bot \arrow[llu] \arrow[lu] \arrow[u] \arrow[ru] \arrow[rru] & & & \\
 & & & & prim32_\bot \arrow[llu] \arrow[ruu] &
\end{tikzcd}$$

Figure 13: The poset for the 32-bit primitives

$$\begin{tikzcd}
 & prim64_\top & \\
long \arrow[ru] & & double \arrow[lu] \\
 & prim64_\bot \arrow[lu] \arrow[ru] &
\end{tikzcd}$$

Figure 14: The poset for the 64-bit primitives

Each variable v will now be kickstarted with an initial set of constraints (called **global** in our implementation):

$$C^v_{global} =$$

$$null\_type \le t(v) \le Object \;\vee\; prim32_\bot \le t(v) \le prim32_\top \;\vee\; prim64_\bot \le t(v) \le prim64_\top$$

In simpler words, at the beginning of our algorithm all we can assert about the type of each variable is that it’s either a reference type, a 32-bit primitive or a 64-bit primitive.

Similarly, whenever we encounter an assignment $I_i : v = 0$, we will assert that

$$Upper\ bound : C^v_i = t(v) \le Object \;\vee\; t(v) \le prim32_\top$$

$$Lower\ bound : C^v_i = null\_type \le t(v) \;\vee\; prim32_\bot \le t(v)$$

This is expressed quite easily in Datalog:

Listing 40: Constant assignment constraint rule

```
ConstraintForVar(?to, ?insn),
LowerConstraintForVar(?to, ?insn),
LowerBoundTypeForConstraint(?to, ?type, ?insn) :-
   SSA_AssignNumConstant(?insn, ?to, _, _),
   StatementType(?insn, _, "32bit"),
   isPrimitive32Bottom(?type).
```

```
ConstraintForVar(?to, ?insn),
LowerConstraintForVar(?to, ?insn),
LowerBoundTypeForConstraint(?to, "null_type", ?insn) :-
   SSA_AssignNumConstant(?insn, ?to, "0", _),
   StatementType(?insn, _, "32bit").
```

The first rule states that if we have a 32-bit constant assignment to $?to$, then

$$prim32_\perp \leq t(?to)$$

The second rule takes care of the special case where the constant is 0, asserting that $null\_type \leq t(?to)$. Note that, whenever the second rule is activated, the first rule is also activated. This means that, in the special case where the constant is 0, we will get the desired disjunction. Otherwise, only the first rule will be activated.

The StatementType predicate is a helper predicate that we output during the fact-generation step, to help us determine the type of the various binary and unary operations. The first argument indicates the instruction, the second the input type, and the third the output type of the instruction. For example, the unary operation `int-to-long` has input type `int` and output type `long`. In the snippet above, 32bit means that $?insn$ deals with 32-bit quantities.

Observe how we only give a lower bound on the variables, and not both upper and lower, as the formulas above suggest. This is fine, since the upper bound ”counterpart” already exists as a result of the global constraints, and introducing it again would be redundant.

All that is left to do regarding the primitive constraints, is to encode the upper and lower bound constraints that the various binary and unary operations impose. First we define some helper predicates for our own convenience:

```
PrimOp(?insn, ?to, ?from) :-
   RewriteAssignUnop(?insn, _, ?to, _),
   RewriteAssignOperFrom(?insn, _, ?from).

PrimOp(?insn, ?to, ?from) :-
   RewriteAssignBinop(?insn, _, ?to, _),
   RewriteAssignOperFrom(?insn, _, ?from).
```

PrimOp basically unifies the input and output arguments of the binary and unary operators into a single predicate.

We will go over the case when either the input or the output argument of PrimOp has type ”int32”:

```
Integer32Constraint(?to, ?insn) :-
   PrimOp(?insn, ?to, _),
   StatementType(?insn, _, "int32").

Integer32Constraint(?from, ?insn) :-
   PrimOp(?insn, _, ?from),
   StatementType(?insn, "int32", _).
```

```
ConstraintForVar(?var, ?insn),
UpperConstraintForVar(?var, ?insn),
UpperBoundTypeForConstraint(?var, ?type, ?insn) :-
   Integer32Constraint(?var, ?insn),
   isInt32Top(?type).

ConstraintForVar(?var, ?insn),
LowerConstraintForVar(?var, ?insn),
LowerBoundTypeForConstraint(?var, ?type, ?insn) :-
   Integer32Constraint(?var, ?insn),
   isInt32Bottom(?type).
```

The fist two rules state that a variable will be subect to integer32 constraints, if it is either the input or the output of a PrimOp whose corresponding argument type is ”int32”. It should be clarified that in our implementation, ”int32” means all 32-bit integer types, as they are defined in our prim32 poset.

Having cleared that up, the meaning of the last two rules should be evident, as they basically define what integer32 constraints are: if a variable is subject to integer32 constraints, then it’s type will reside between $int32_\top$ and $int32_\bot$.

More formally:

$$int32_\bot \leq t(?var) \leq int32_\top$$

Other PrimOps are implemented similarly.

Finally, we must modify our choice function. This is necessary, since we introduced virtual types, which we don’t want to include in the output of our algorithm.

First, let us introduce some helper predicates:

```
ExistsNonGuardType(?var),
NonGuardCompatibleTypes(?var, ?type) :-
   TypeCompatibleWithAllConstraints(?var, ?type),
   !isGuardPrimitive(?type).
```

In our implementation, virtual primitives are refered to as GuardPrimitives.

ExistsNonGuardType encodes the information that for variable v, it holds that $T_v \backslash Guard \neq \emptyset$, or more intuitively that there is a non-guard type in T_v, while NonGuardCompatibleTypes computes $T_v \backslash Guard$.

Finally, we use these rules to implement the new SSA_Type rules:

Listing 41: SSA_Type rules for primitives

```
SSA_TypeOrd(?var, ?minTypeOrd) :-
   ExistsNonGuardType(?var),
   ?minTypeOrd = min ord(?t) : NonGuardCompatibleTypes(?var, ?t).

SSA_Type(?var, ?minType) :-
   NonGuardCompatibleTypes(?var, ?minType),
   SSA_TypeOrd(?var, ?minTypeOrd),
   ?minTypeOrd = ord(?minType).

SSA_Type(?var, "int") :-
   TypeCompatibleWithAllConstraints(?var, _),
   !ExistsNonGuardType(?var).
```

SSA_TypeOrd selects for each variable v the type $t \in T_v \backslash Guard$ that minimizes $ord(\cdot)$.

The computation of SSA_Type can be described as follows:

- If $T_v \backslash Guard \neq \emptyset$, then choose the type with the minimum ord from $T_v \backslash Guard$.
- Otherwise, if $T_v \neq \emptyset$, then T_v contains only virtual types. In this case, default to `int`.

Defaulting to `int` is just a heuristic to avoid outputing no type in the case that all candidate types are virtual.

This concludes our type inference algorithm.

Observe how the core of our type inference algorithm is language agnostic. As long as the language supports subtype polymorphism, the iteration and combination of the constraints will remain the same. The only parts that one would need to change is the encoding of the subtype constraints, and of any language specific heuristics that can aid the type inference algorithm, just like we did with primitives.

5. EXPERIMENTAL EVALUATION

In this section, we will evaluate the performance of our SSA transformation and type inference algorithm. All our measurements here are performed only on application facts, without android platform facts. This is due to the fact that the DEX front end is currently under development, and doesn't support platform fact generation yet.

The machine we run these experiments on features an Intel Xeon E5-2667 with 256 GB of RAM.

First we will compare the amount of core input and output facts, that is all the facts excluding LocalAssign, Var-Type and Var-Declaring, as these are obviously a lot more post SSA transformation and type inference.

Table 1: Input vs output facts of our SSA Transformation

APK	Core input facts	Core output facts	Core in / Core out (%)	Input AssignLocal	Output AssignLocal
androidterm-1.0.70	70678	69947	99%	1260	6632
facebook_lite-85.0.0	1123546	1110702	99%	6771	109072
instagram-10.5.1	3168133	3138016	99%	32408	239512
signal-4.12.3	5441749	5392720	99%	62697	391715

From the above table we make the following observations:

- Most input facts are preserved after the SSA transformation. The small loss of facts (1%) is due to some inaccuracies in the modeling of the CFG that disables variable liveness information from reaching instructions in exception handlers, causing the relevant fact rewrite rules to fail.
- Our SSA transformation introduces a large number of new local assignments. This is to be expected, but the increase is still a bit high. In our transformation we didn't attempt to produce minimal SSA form, which could possibly avoid a lot of unnecessary assignments.

Next we will observe how our type inference algorithm performs, by comparing the amount of variables mapped to some type vs the number of variables declared in the program:

Table 2: Percentage of variables assigned to some type

APK	Declared variables	Type-assigned variables	%
androidterm-1.0.70	14817	10847	73%
facebook_lite-85.0.0	243900	171251	70%
instagram-10.5.1	660725	463590	70%
signal-4.12.3	1111922	821686	73%

The performance of our type inference algorithm is significantly hindered by the fact that the DEX front end does not currently output platform facts. Many methods take arguments whose type is defined in the platform (e.g `android.os.*`), return such types or call platform methods. However, as the platform facts are not present, the subtype relation does not contain any records regarding the hierarchy of platform types. This prevents our algorithm from outputing a type for a variable that is constrained by a platform type.

The addition of platform facts should provide a significant boost in the performance of our type inference algorithm. For example, we estimate that the facebook_lite percentage will increase from 70% to at least 87%. Moreover, our primitive heuristics might be inaccurate and thus negatively affect our algorithm.

Finally, we include a table with some execution times:

Table 3: Execution times of our Datalog program

APK	SSA	SSA + Type Inference
androidterm-1.0.70	0s	0s
facebook_lite-85.0.0	11s	17s
instagram-10.5.1	36s	70s
signal-4.12.3	49s	107s

It is evident that for smaller programs, the SSA transformation dominates the running time of our implementation, while the opposite is true for larger workloads.

6. CONCLUSIONS

SSA form and type information are important for the accuracy of many static analyses. Doop's in-house DEX front end produces facts that are not in SSA form and doesn't perform any type inference to resolve the types of the input program's variables. In this thesis, we were able to provide a succint Datalog program that transforms the front end facts into SSA form and perform type inference, effectively extending the functionality of Doop's DEX front end.

There are several ways this work could be extended further:

1. More accurately modeling the CFG and more specifically exception handlers, for reasons discussed in the Evaluation Section 5.
2. Improving and re-examining the heuristics that we introduced in our type inference algorithm.
3. A formal proof of correctness for the SSA transformation.
4. A formal investigation of the completeness and soundness characteristics of our type inference algorithm. What are the shortcomings of our approach with respect to these two properties?
5. Further experimentation when full facts (app and platform) are available.

ACRONYMS AND ABBREVIATIONS

SDK	Software Development Kit
NDK	Native Development Kit
APK	Application Package
JVM	Java Virtual Machine
VM	Virtual Machine
ART	Android Runtime
DEX	Dalvik Executable
EDB	Extensional Database
IDB	Intensional Database
CFG	Control Flow Graph
SSA	Static Single Assignment
GCD	Greatest Common Decendant
LCA	Least Common Ancestor

REFERENCES

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

[2] Steven S. Muchnick (1998), Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[3] Yannis Smaragdakis and George Balatsouras (2015), ”Pointer Analysis”, Foundations and Trends® in Programming Languages: Vol. 2: No. 1, pp 1-69

[4] Doop Project Repository [Online]
`https://bitbucket.org/yanniss/doop`

[5] Android Platform Architecture [Online]
`https://developer.android.com/guide/platform/`

[6] Android Application Fundamentals [Online]
`https://developer.android.com/guide/components/fundamentals`

[7] Android Art and Dalvik [Online]
`https://source.android.com/devices/tech/dalvik`

[8] Souffle Home [Online]
`https://souffle-lang.github.io/`

[9] Soot Home [Online]
`https://github.com/Sable/soot`

[10] LogicBlox Home [Online]
`https://developer.logicblox.com/technology/`

[11] Souffle Datalog [Online]
`https://souffle-lang.github.io/docs/datalog/`

[12] DEX Format [Online]
`https://source.android.com/devices/tech/dalvik/dalvik-bytecode`

[13] Pointer Analysis Lecture Notes [Online]
`https://www.cs.cmu.edu/~aldrich/courses/15-819O-13sp/resources/pointer.pdf`

[14] Doop Tutorial, PLDI 2015 [Online]
`https://plast-lab.github.io/doop-pldi15-tutorial/`

[15] Android NDK [Online]
`https://developer.android.com/ndk/guides/`

[16] Apktool [Online]
`https://ibotpeaches.github.io/Apktool/`

[17] SSA Form Wikipedia [Online]
`https://en.wikipedia.org/wiki/Static_single_assignment_form`

[18] Android APK [Online]
`https://en.wikipedia.org/wiki/Android_application_package`