Extracting Functions from Mobile Apps

Harmony Singh

Master of Science
Computer Science
School of Informatics
University of Edinburgh

2018

Abstract

Mobile applications, or apps, are quickly becoming ubiquitous. The spread of malicious software is also on the rise, particularly in Android apps. Various malware analysis tools exist to cope with the rising numbers, however their efficiency and robustness are lacking. Therefore, there still exists a dire need for research in the field of mobile security. In this project, we attempt to address this problem with the help of formal methods of verification. We implement a functional language to represent the behaviour of Android apps, which is useful for malware analysis, to check against security-related properties.

Acknowledgements

I'd like to express my gratitude to my supervisors – David Aspinall and Wei Chen, who provided me with guidance and support, tirelessly. I would also like to acknowledge the existing work done by Wei Chen on this topic; work done here benefits from discussions about the same. I'd also like to thank them for giving me the opportunity to work on this project, which has indeed been a major learning experience.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Harmony Singh*)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With mobile phones and tablets becoming more prevalent, there is also a simultaneous rise in the number of mobile applications (more commonly known as apps), which provide a wide range of functional capabilities. Android and iOS being the most widely used mobile operating systems (in that order), their respective app markets – Google Play and Apple's App Store – have both garnered more than 2 million apps since their launch. As of March 2017, Google Play has on offer 2.8 millions apps to be downloaded and Apple's App Store, 2.2 million. [33]

Although the abundance and ease of availability of apps is said to have benefitted its users, there is a downside in terms of app security, or rather, the lack thereof. This is due to apps becoming an increasingly popular medium for attackers to inflict with various forms of malicious software (in short, malware), which have the potential to cause grave damage. [24, 39] As presented in [44], Intel Security detected malware from over 190 countries and found more than 6000 instances per hour in some cases.

Android is the mobile operating system most targeted with malware, due to its popularity and open source structure, as shown in [34]. To cope with the rising numbers, various anti-malware tools are available on Google Play. Consequently, the goal of these tools is to efficiently detect malicious apps, with the help of automated (or semi-automated) program analysis techniques. However, their robustness and accuracy is questionable.

As shown in a study of ten prevalent anti-malware tools for Android in [41], even common obfuscation techniques – the purpose of which is to avoid detection by employing obscurity – in malware could not be handled by these tools.

Along with the rapid increase in numbers, the complexity of mobile malware is also on the rise. The results presented in [40] demonstrate the various challenges that malware analysis techniques face, while tackling current malware[1], and why further research is needed in this area.

The complexity of mobile malware results from its intricate structure; involving components that interact with each other in complicated ways. [40] Therefore, it is imperative to identify the malicious components through models that accurately depict their behaviour. These models are generally constructed manually in state-of-the-art static analysis tools [1, 47], and are therefore rigid due to it being hard-wired in code. [13] Resultantly, malicious apps can evade detection through different execution paths that are not covered by these models. The research proposed in [13], and this project, seek to address these points, which we will briefly discuss in the next section.

1.1 Objective

This project is constituent of a broader work in progress [13], which aims to construct a resilient security analysis framework for mobile apps, specifically for Android apps. The goal is to enable complex features – such as callbacks, inter-component communication, and lifecycle – to be modelled, while allowing flexibility for making changes. A specification language is to be designed, to formalise these platform features. As an end product, an automated lightweight tool is proposed for checking simple security-related properties, such as, whether an application has access to your SMS messages in the background and can send arbitrary messages or not.

First off, mobile apps and their assembly code are reduced to light assembly code. The focus of this project is to simplify the light assembly code into functional expressions, while retaining their semantics. The formalisation of the platform features, via a specification language, is followed by their conversion into functional expressions as well. Although, in this project, we will focus on the functional expressions resulting from the app only. Both of these representations can then be verified against security properties that we want to check, e.g. to deny unwanted behavioural patterns. Figure 1.1 provides a brief look at the procedure, highlighting in colour the steps that we aim to cover in this project.

[1]In our project, malware and apps will specifically refer to Android malware and Android apps, respectively.

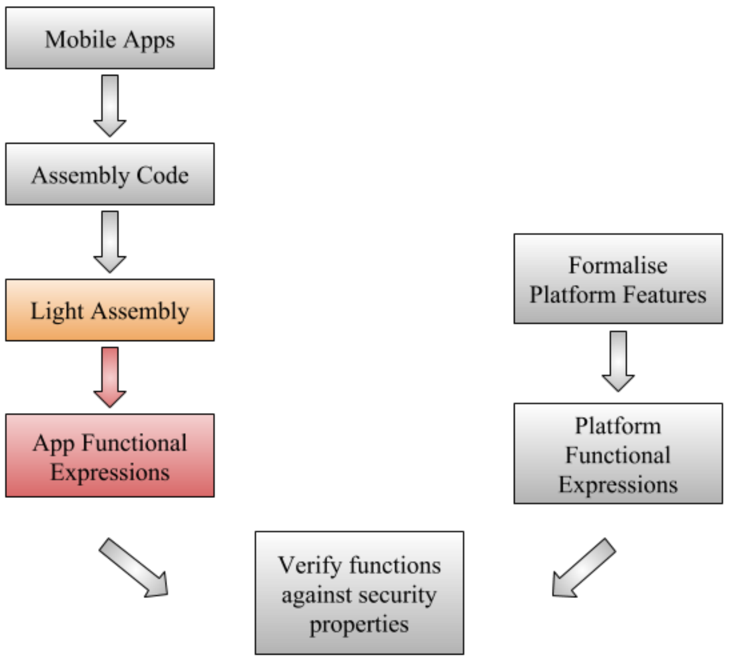

Mobile Apps
Assembly Code
Light Assembly
App Functional Expressions
Formalise Platform Features
Platform Functional Expressions
Verify functions against security properties

Figure 1.1: Illustration of the analysis procedure presented in [13].

Therefore, in this project, our objective is to extract functions from the light assembly code that we receive as input. The corresponding implementation is based on the formal design of a simple functional language. We also delve into the previous steps of the analysis, to better understand our input, and to make modifications.

Additionally, we attempt to demonstrate the retention of semantics between translations, by comparing their respective translated programs. With the help of evaluators, and test cases, we aim to show the correspondence in their evaluation, i.e. the results they compute.

1.2 Outline

We started with the motivation and objective of this work, followed by an overview of related work. The next chapter presents the preliminaries reviewed to carry out the project. This is followed by the formal design chapters, 3 and 4, and their implementation details in Chapter 5. After that, we present details on the evaluators and the evaluation results, followed by a general discussion and reflection. We end by briefly examining future work concepts.

1.3 Related Work

Static analysis techniques of malware analysis have shown to be powerful, as they explore all possible execution paths of a program. This helps in addressing the questions of what data has been leaked, or what private information has been stolen. [40]

We also explore the benefits of formalisation of programs. In [38], a programming logic for bytecode programs has been developed, which aims to optimise the running speeds of Java programs. Some of the checks that are necessary at run-time can be formally verified prior to running the program, thereby reducing delays.

With the help of the theorem-prover, Isabelle/HOL, major parts of the Java Virtual Machine have been formalised in [37]. This formalisation helps to avoid inconsistencies and ambiguity that are present in informal specifications.

Function extraction:

We see a brief introduction to function extraction, in the context of software testing, in [36]. The paper points out that even the most exhaustive form of software testing, along with thorough inspections, cannot capture the complete functionality of a program. In addition to that, the process of software testing can be rather expensive in terms of cost, time and effort. Therefore, what is needed is an all-encompassing view of what the software does.

Function extraction applies mathematical foundations for the automation of representing software behaviour, up to the possible maximum. It is seen to be much faster and accurate as compared to manual testing. Therefore, making use of function extraction to formulate a program's behaviour with utmost mathematical precision is seen to be promising. [46]

The primary objective of software testing mainly pertains to its functionality. However, if automated tools were to accomplish that meticulously, more attention can be diverted to issues such as security testing, reliability, as well as performance. [36] Function extraction also aids in detecting malware in programs, as seen in [29], the focus of which is on understanding malware behaviour, attacker methods and also its countermeasures.

[32] employs function extraction in the domain of low-level imperative programming languages. It makes use of Hoare Logic [23] for machine code behaviour abstraction,

which is used for further program verification and for implementing reliable compilation. They have implemented this method for function extraction in HOL4 and also applied it to ARM, x86 and PowerPC machine code.

Type systems used for verification:

Type theory is seen to be a useful specification language, especially for the proof of purely functional programs. In [21], type theory is used to study the certification of programs that have both functional and imperative components. The paper describes the formal method of providing software correctness as consisting of three steps, namely, the specification; a method to generate proof obligations; and a framework to establish their identity.

The paper also uses the concept of typing with effects, which is inspired by the work presented in [45]. The work mentioned has been implemented in the Coq proof assistant. It takes an annotated program as its argument and generates a set of proof obligations. Their definition of annotated type consists of the usual notion of type, an effect and a specification as a pre- and a post-condition.

[22] presents type-based enforcement of secure programming guidelines, particularly for code injection prevention at SAP, which is similar to what we want to achieve but in the context of Android apps.

Learning unwanted behaviours:

In [16, 15, 14, 17], we see the notion of unwanted behaviours in apps, found in malware samples, such as sending a text message to premium numbers without the user's knowledge. These unwanted behaviours can be isolated through malware classifiers. To improve the robustness of the classifiers, the unwanted behaviours can be abstracted as automata in order to help in learning and verifying them. It has also been shown that semantics improves robustness of such classifiers.

Chapter 2

Preliminaries

2.1 Theoretical Background

A theoretical overview is presented for the reader to appreciate the underlying theoretical concepts employed in this project, especially for those unacquainted with the field of formal languages and verification. It has been assumed that the reader has a basic understanding of automata theory and its languages, therefore a primer on that topic has been omitted. All examples and definitions used in the following two sub-sections are taken or inspired from [35], unless otherwise stated.

We start with the basics of type theory, which forms the basis of verification of our functional representation, followed by a primer on lambda calculus, which we use for the translation.

2.1.1 Type Systems

A wide spectrum of formal methods are commonly used in the field of software engineering, to make sure that a system abides by its specifications by behaving correctly. From a range of powerful frameworks to lightweight formal methods, type systems from the latter category have emerged as the most preferred and extensively used. Type systems or type theory generally refers to a more expansive area of study in logic, mathematics and philosophy, but for our purposes they shall pertain to reasoning about programs.

Properties of a type system:

"A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute."

Type systems are static, in the context of statically typed programming languages. As a result, they are also conservative: they cannot prove the presence of bad behaviours but can only prove their absence, in contrast to dynamic or latent checking. In essence, a type and effect system is used to analyse programs, which depends on a specified policy as input [22].

A program is well-typed if its behaviour corresponds to the one described by the type system. Otherwise, the program is ill-typed and is rejected, when the type system no longer recognises its behaviour. However, that may also be due to the type system not being expressive enough; it fails to capture the behaviour of a program that does actually fit the given specification. Expressiveness and conservativity are thus relevant aspects of research in this field. Another considerable detail is that a type system can provide guarantees that a well-typed program is free from only certain unwanted features. It does not ensure prohibiting any arbitrary undesired behaviour.

In order to ensure the safety or soundness of well-typed programs, also known as type safety, the progress and preservation theorems are introduced. It must be guaranteed that a program does not malfunction or get stuck[1]. The progress theorem stipulates that a well-typed term does not get stuck; either it is in the final state and has an acceptable value or it can move to another state in accordance with the evaluation rules[2]. The preservation theorem states that a consequent term, reached after an evaluation step of a well-typed term, is also well-typed.

To speculate about the behaviour of a program, its syntax and semantics are formalised with the help of a language (metalanguage). An example is given below: -

Syntax:

`t::=`	*terms* :
`    true`	*constant true*
`    | false`	*constant false*

[1]That is to say that it does not get stuck at a state that is undefined by the type system; where the evaluation rules are inapplicable and the value reached is not final. This is better illustrated in the next subsection.

[2]Introduced in the next subsection.

```
        |if t then t else t                                   conditional

v::=                                                             values :
        true                                                  true value
        |false                                               false value
```

`t` is a placeholder for the terms, such as true and false, that are declared below it. `t ::=` represents the declaration. This implies that when `t` is encountered, it can be replaced by any of the specified terms. Below it are values, which are possible final outcomes of the system, reduced to its simplest form.

Semantics:

After the syntax of the language has been formalised, its semantics must also be considered. There are three main styles to formalise the semantics of a language, which determines how values are evaluated; namely operational, denotational and axiomatic semantics. In our project, we will principally adopt operational semantics.

Operational semantics makes use of an abstract machine to represent the behaviour of a program. It is abstract because it does not use a low-level instruction set as its machine code. Instead, it makes use of the language terms as its mode of representation.

More specifically, terms of the language are used to depict the machine's state, and transition functions determine its behaviour. Depending on the current state, a transition function either defines the next step that must be taken by simplifying a term or shows that the machine has reached its final step. To be more precise, this refers to the small-step style of operational semantics, also known as structural semantics, which we illustrate with an example next.

Evaluation:

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \texttt{(E-IFTRUE)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \texttt{(E-IFFALSE)}$$

$$\frac{t_1 \longrightarrow t_1'}{\begin{array}{l}\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \\ \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3\end{array}} \qquad \texttt{(E-IF)}$$

An evaluation relation/reduction is given in the form of $t \rightarrow t'$, which means that in one step, `t` evaluates to `t'`. Three inference or evaluation rules are given to define this relation. More specifically, when the machine is in state `t`, it can move to the next state `t'` in one computational step.

```
if true then true else false
▶ true
```

This is a sample program in the given language. Here, the first (E-IFTRUE) rule is applied and the state of the machine moves to true, which is a value and therefore is the final result of evaluation.

The preceding paragraphs have been taken and modified from [43].

Types:

In order to ensure type safety, we'd like to guarantee that a term does not get stuck, without evaluating it. For example, we want to ensure that indeed the guard of a conditional is a boolean value and not a numerical value. The language given above is untyped; it does not have types assigned to its terms. We shall now add types to it, in order to ensure type safety.

The typing relation is written as `t:T` and is defined by a set of inference rules assigning terms their types. The typing rules for this typed language are given below. The rules, `T-True` and `T-False`, assign the type `Bool` to the boolean constants, `true` and `false`.

`T::=`	*types* :
`Bool`	*type of booleans*

$$\texttt{true : Bool} \qquad \texttt{(T-TRUE)}$$

$$\texttt{false : Bool} \qquad \texttt{(T-FALSE)}$$

$$\frac{\texttt{t}_1\texttt{:Bool} \quad \texttt{t}_2\texttt{:T} \quad \texttt{t}_3\texttt{:T}}{\texttt{if t}_1\texttt{ then t}_2\texttt{ else t}_3\texttt{ : T}} \qquad \texttt{(T-IF)}$$

2.1.2 Lambda Calculus

It has been found that a complicated programming language can be studied by transforming its fundamental structure into a smaller core language or calculus. This can be accomplished with the help of derived forms, which can be understood by evaluating them to the core. [26, 28, 27] A formal system known as the lambda calculus [19, 20] is a core language used, in which all operations are reduced to a series of merely function definition and application.

Subsequently, lambda calculus has been applied in various fields, such as language design, programming language specifications, and in the study of type systems. Its significance emerges from the fact that it is both a simple language to write in, as well as a language that can be reasoned about mathematically and proved rigorously. The concepts and techniques described for lambda calculus are transferable to other core calculi that are akin to it, such as pi-calculus [31, 42] and object calculus [30].

Every programming language seeks to abstract its procedures (or functions) by avoiding repetitiveness. Rather than doing the same calculation repeatedly, we can create a function that accomplishes the same, but in a generalised manner by taking a parameter (or parameters). This will be made clearer by the example below.

Consider the following calculation:

```
(6*5*4*3*2*1) * (5*4*3*2*1) + (4*3*2*1) - (3*2*1),
```

which can be rewritten as:

```
fact(6) * fact(5) + fact(4) - fact(3)
```

where:

```
fact(n) = if n=0 then 1 else n * fact(n-1).
```

For every positive number `n`, the function `fact`, with its parameter `n`, calculates the factorial of `n` and returns it as a result. Suppose we write "λ`n`" to signify "a function that, for each `n`, yields...", then the function definition of `fact` can be rephrased as:

```
fact = λn.  if n=0 then 1 else n * fact(n-1)
```

Then, `fact(0)` would mean the function `(λn.  if n = 0 then 1 else...)` applied to the argument `0`. The result is the value computed by substituting every `n` in the function body with `0 (if 0=0 then 1 else...)`, which is `1`. Lambda calculus demonstrates this form of function definition and application in the "purest possible form". Everything is a function in lambda calculus, including the arguments given to the functions as well as the results returned by them.

Syntax:

The syntax of lambda calculus consists of three types of terms[3]: a variable, an abstraction of a variable from a term, and the application of a term to another term. These can be captured in the following grammar:

`t::=`		*terms* :
	`x`	*variable*
	`λx.t`	*abstraction*
	`t t`	*application*
`v::=`		*values* :
	`λx.t`	*abstraction value*

However, a slight ambiguity must be pointed out. The syntax of a programming language can refer to either its concrete syntax or its abstract syntax. Concrete (or surface) syntax is the one encountered directly: the character sequences that are written and read. Abstract syntax refers to the representation of the program as labeled trees called abstract syntax trees (ASTs). The simple structure of ASTs make it easier to perform complex operations and to conduct proofs about them, and are especially relevant for the workings of compilers and interpreters.

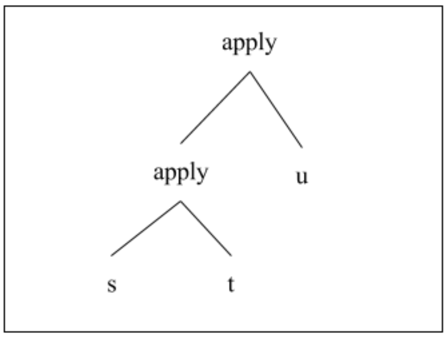


Figure 2.1: The abstract syntax tree for `s t u` and `(s t) u`.

Typically, when viewing the grammar of the terms presented above, the abstract syntax is the one referred to. Even if concrete terms are used in a linear fashion in examples,

[3]Terms specifically refer to lambda-terms in our context; hence `t` represents lambda-terms too.

their tree structures are kept in mind. Additionally, two conventions are adopted while writing linearly. The first is that application associates to the left, such that `s t u` and `(s t) u` have the same AST, as shown in Figure 2.1. Secondly, abstractions are extended to the right as much as possible, such that λ`x. `λ`y. x y x` and λ`x.(`λ`y. ((x y) x))` have the same tree, as seen in Figure 2.2.

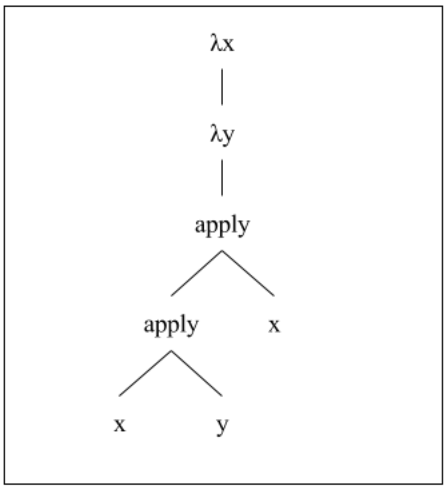


Figure 2.2: The abstract syntax tree for λ`x. `λ`y. x y x` and λ`x.(`λ`y. ((x y) x))`.

Another detail regarding the syntax of the lambda-calculus is the scope of its variables. If a variable `x` appears in the body `t` of an abstraction λ`x.t`, then its occurrence is said to be bound. More specifically, λ`x` is the binder, the scope of which is `t`; `x` is bound by this abstraction. Otherwise an occurrence of a variable not bound by an abstraction is said to be free. Examples of free and bound occurrences are given below:

- Bound – the occurrences of `x` in λ`x.x`, λ`x.`λ`y.x y`, and the first occurrence of `x` in `(`λ`x.x)x`.
- Free variables – the occurrences of `x` in `x y`, λ`y.x y`, and the second occurrence of `x` in `(`λ`x.x)x`.

Evaluation:

The basic form of the lambda calculus consists of abstraction and application only. Therefore, the way to compute values is through applying arguments to functions, where arguments themselves are functions. The left-hand side of an abstraction is replaced by the argument, in the right side or the body of the abstraction.

$$(\lambda x.t_1)\, t_2 \longrightarrow [x \mapsto t_2]\, t_1$$

The occurrences of the variable $\mathtt{x}$ are replaced by $\mathtt{t}_2$ in $\mathtt{t}_1$, and $[\mathtt{x} \mapsto \mathtt{t}_2]\ \mathtt{t}_1$ denotes the substitution. A term in the form of $(\lambda\mathtt{x}.\mathtt{t}_1)\ \mathtt{t}_2$ is known as a reducible expression, or redex, in short, and the operation is known as reduction.

Different strategies exist for the reduction of terms, but we will stick to the call-by-value strategy as it is the most widely used, and allows for easy addition of features. In this strategy, only the outermost or the left-hand side redexes are reduced, and is only done so when the right-hand side is reduced to a value. Therefore, if $\mathtt{t}_2$ is not a value, $(\lambda\mathtt{x}.\mathtt{t}_1)\ \mathtt{t}_2$ is not reduced, and remains as it is.

That is why values consist of arbitrary lambda-terms in our syntax, as call-by-value evaluation stops on encountering a lambda. The small-step operational semantics for the lambda terms we have described is as follows.

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \texttt{(E-APP1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1'\ \mathtt{t}_2} \qquad \texttt{(E-APP1)}$$

$$(\lambda\mathtt{x}.\mathtt{t}_1)\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\ \mathtt{t}_1 \qquad \texttt{(E-APPABS)}$$

2.2 Technical Background

Without delving too much into the minutiae, a brief technical overview is presented to give the reader an insight into the workings of Android apps and how they will be analysed. Some knowledge of computer systems and architectures is assumed here.

Android apps are mostly written in Java. The code, along with any additional files and data, is compiled by Android SDK (software development kit) tools into an Android Package (APK), which is an archive file ending in `.apk`. An APK file contains all of an Android app's content and is the file used by Android devices to download the app. [3]

Android runtime (ART) is the runtime used on Android applications, which are compiled to Dalvik bytecode prior to that. ART and its antecedent, Dalvik, are meant to be compatible: apps developed in Dalvik should also work when running with ART, apart from a few exceptions [11]. The runtime executes the Dalvik Executable (Dex) format and Dex bytecode specification. [4] [6]

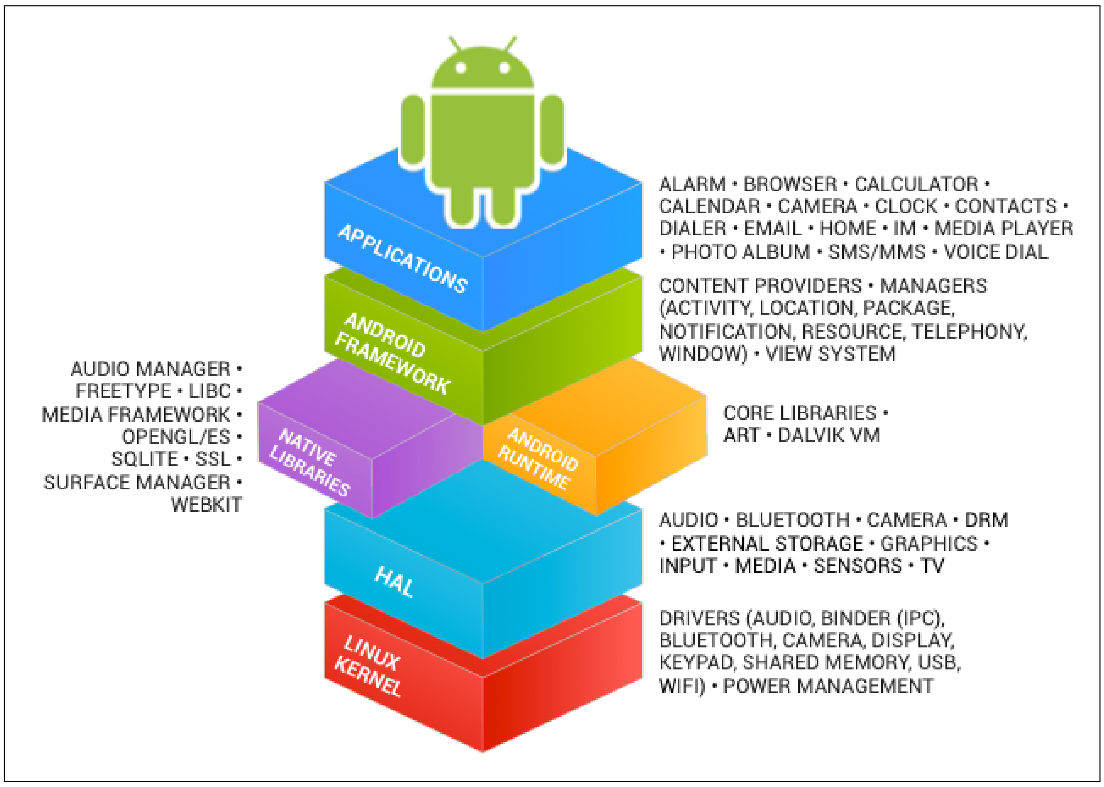

ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
APPLICATIONS
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
ANDROID FRAMEWORK
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
NATIVE LIBRARIES
ANDROID RUNTIME
CORE LIBRARIES • ART • DALVIK VM
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
HAL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT
LINUX KERNEL

Figure 2.3: Android Stack [10].

The official IDE for developing Android apps, Android Studio [2], can be used to build the APK of an app, as demonstrated in Figure 2.4. Unzipping the APK file reveals the following contents, as shown in Figure 2.5. The most noteworthy is the `classes.dex` file, which is the Dalvik Executable (hence the `.dex` extension) that runs on a device. It contains Dalvik bytecode, which is not easily readable.

However, by disassembling it, one can see information about the Java classes used in the app, e.g. by using an Android platform tool called `dexdump` [9], which converts bytecode into assembly code. It returns a text file, which is more coherent, for the `classes.dex`.

Reverse engineering Android applications helps gain an understanding of its workings. Dissecting an Android application, or more specifically its APK, gives valuable insight for security analyses, especially for the purpose of malware detection.

Dalvik Bytecode Format:

All of the details presented below, about the Dalvik bytecode format, have been taken from [5, 7].

In general, the machine configuration is meant to resemble existing commonplace architectures; and its calling conventions are analogous to those in C. The machine model makes use of registers and fixed-sized frames. Register names are of the form `vX`, where `X` is a number; e.g. `v0`, `v1`, `v2`.

Every frame comprises a specific number of registers determined by the method. A frame also contains auxiliary data needed for the method execution, like the program counter and a reference to the `.dex` file, which contains the method.

Bit values like integers and floating point numbers use registers that are 32 bits wide, whereas 64-bit values make use of adjacent register pairs. A 16-bit unsigned quantity is the storage unit for instructions.

A lot of instructions are allowed to address only the first 16 registers, due to the fact that methods commonly do not need more than 16 registers. However, in some cases, a lot more registers can be allocated for reference. For instance, for a pair of catch-all `move` instructions, registers in the range of `v0` – `v65535` can be addressed.

There are also a few "pseudo-instructions" that carry data payloads of variable length referred to by normal instructions (for instance, `fill-array-data`). These instructions must never be come across during the execution flow. The locations of these instructions must be bytecode offsets that are even-numbered, which is 4-byte aligned. To comply with this, an extra `nop` instruction is used for spacing when required.

A few opcodes[4] have a name suffix to distinguish between the type(s) they operate on. 32-bit opcodes that are type-general are not marked. Type-general 64-bit opcodes contain the `-wide` suffix. Type-specific opcodes consist of their type (or an abbreviation of it), such as `-boolean`, `-char`, `-byte`, `-short`, `-int`, `-long`, `-float`, `-double`, `-string`, `-class`, `-void`, `-object`.

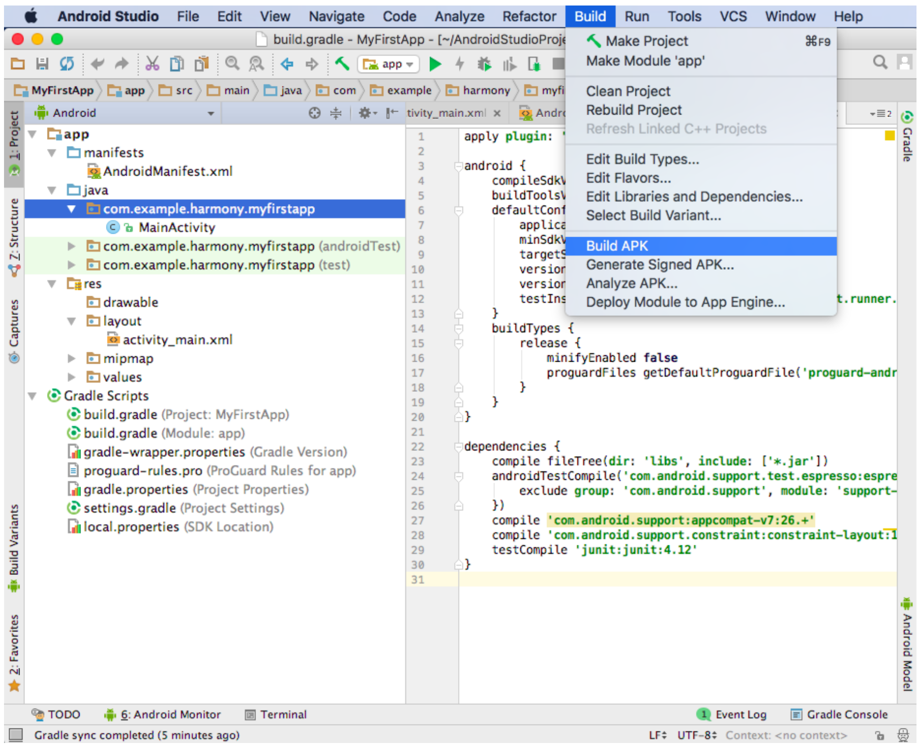
Opcodes with dissimilar instruction layouts or options are differentiated by an opcode suffix, which is distinct from the rest of the name and specified after a "`/`".

The order of instruction arguments is destination followed by source, respectively. An example of an instruction is given below, with its components explained.

In the instruction "`move-wide/from16 vAA, vBBBB`":

[4]Short for "operation code", it specifies the operation that needs to be performed by the (in this context, virtual) machine.

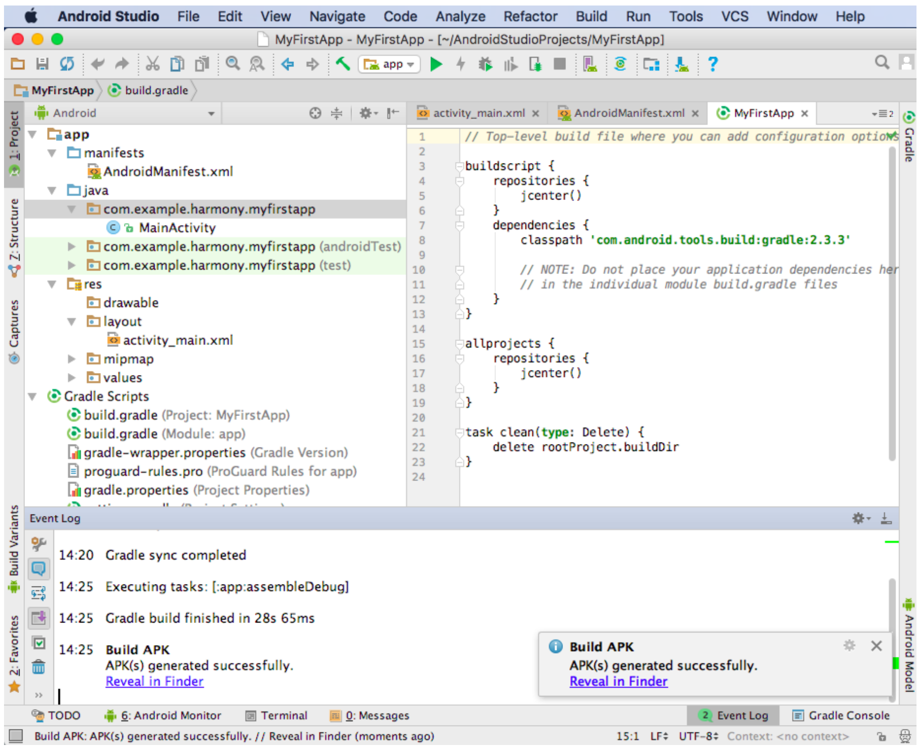
- "`move`" is the base opcode, which specifies the operation of moving a register's value.
- "`wide`" is the name suffix signifying that it operates on wide (64 bit) data.
- "`from16`" is the opcode suffix showing a variant, the source of which is a 16-bit register reference.
- "`vAA`" is the destination register and must be in the range of `v0` – `v255`.
- "`vBBBB`" is the source register and must be in the range of `v0` – `v65535`.


Android Studio File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
build.gradle - MyFirstApp - [~/AndroidStudioProje
MyFirstApp app src main java com example harmony myfi
Android
Make Project ⌘F9
Make Module 'app'
Clean Project
Rebuild Project
Refresh Linked C++ Projects
Edit Build Types...
Edit Flavors...
Edit Libraries and Dependencies...
Select Build Variant...
Build APK
Generate Signed APK...
Analyze APK...
Deploy Module to App Engine...
app
manifests
AndroidManifest.xml
java
com.example.harmony.myfirstapp
MainActivity
com.example.harmony.myfirstapp (androidTest)
com.example.harmony.myfirstapp (test)
res
drawable
layout
activity_main.xml
mipmap
values
Gradle Scripts
build.gradle (Project: MyFirstApp)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
TODO
6: Android Monitor
Terminal
Event Log
Gradle Console

```
apply plugin: '

android {
    compileSdkV
    buildToolsV
    defaultConf
        applica
        minSdkV
        targetS
        version
        version
        testIns                          t.runner.
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-andr
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espre
        exclude group: 'com.android.support', module: 'support-
    })
    compile 'com.android.support:appcompat-v7:26.+'
    compile 'com.android.support.constraint:constraint-layout:1
    testCompile 'junit:junit:4.12'
}
```

Gradle sync completed (5 minutes ago)   LF÷ UTF-8÷ Context: <no context>

(a)


Android Studio File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MyFirstApp - MyFirstApp - [~/AndroidStudioProjects/MyFirstApp]
MyFirstApp build.gradle
Android
activity_main.xml
AndroidManifest.xml
MyFirstApp
app
manifests
AndroidManifest.xml
java
com.example.harmony.myfirstapp
MainActivity
com.example.harmony.myfirstapp (androidTest)
com.example.harmony.myfirstapp (test)
res
drawable
layout
activity_main.xml
mipmap
values
Gradle Scripts
build.gradle (Project: MyFirstApp)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
Event Log
TODO
6: Android Monitor
Terminal
0: Messages
Event Log
Gradle Console

```
// Top-level build file where you can add configuration options

buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.3'

        // NOTE: Do not place your application dependencies her
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

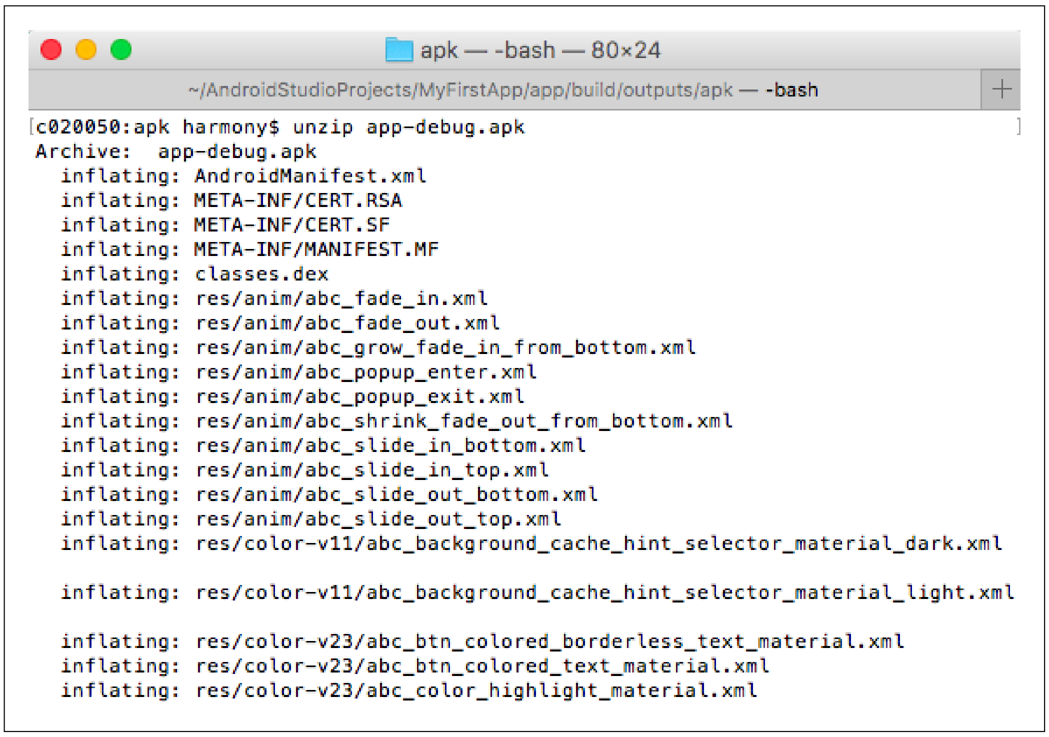
```
14:20  Gradle sync completed

14:25  Executing tasks: [:app:assembleDebug]

14:25  Gradle build finished in 28s 65ms

14:25  Build APK
       APK(s) generated successfully.
       Reveal in Finder
```

Build APK
APK(s) generated successfully.
Reveal in Finder

Build APK: APK(s) generated successfully. // Reveal in Finder (moments ago)   15:1 LF÷ UTF-8÷ Context: <no context>

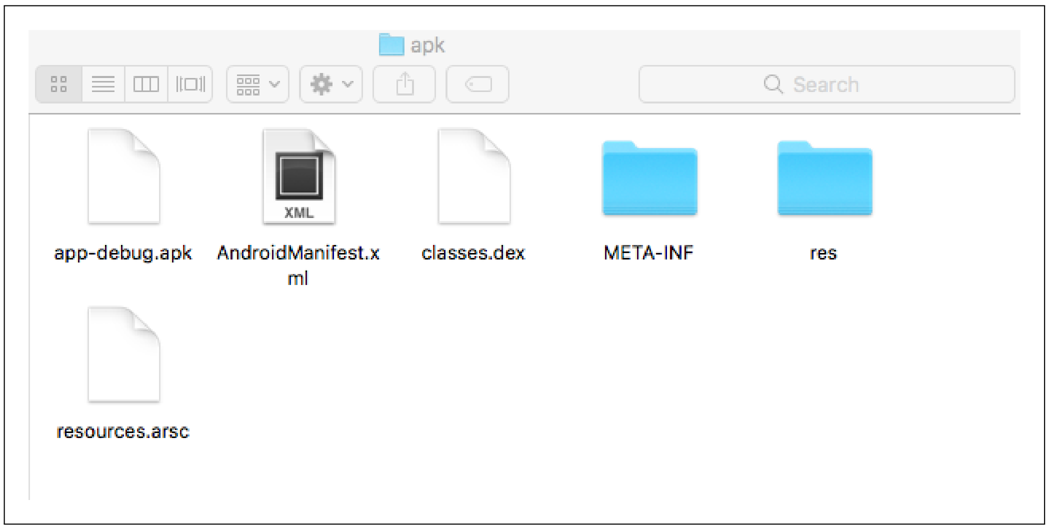
(b)

Figure 2.4: (a) & (b) illustrate building an app's APK in Android Studio.


apk — -bash — 80×24
~/AndroidStudioProjects/MyFirstApp/app/build/outputs/apk — -bash

```
c020050:apk harmony$ unzip app-debug.apk
Archive:  app-debug.apk
  inflating: AndroidManifest.xml
  inflating: META-INF/CERT.RSA
  inflating: META-INF/CERT.SF
  inflating: META-INF/MANIFEST.MF
  inflating: classes.dex
  inflating: res/anim/abc_fade_in.xml
  inflating: res/anim/abc_fade_out.xml
  inflating: res/anim/abc_grow_fade_in_from_bottom.xml
  inflating: res/anim/abc_popup_enter.xml
  inflating: res/anim/abc_popup_exit.xml
  inflating: res/anim/abc_shrink_fade_out_from_bottom.xml
  inflating: res/anim/abc_slide_in_bottom.xml
  inflating: res/anim/abc_slide_in_top.xml
  inflating: res/anim/abc_slide_out_bottom.xml
  inflating: res/anim/abc_slide_out_top.xml
  inflating: res/color-v11/abc_background_cache_hint_selector_material_dark.xml

  inflating: res/color-v11/abc_background_cache_hint_selector_material_light.xml

  inflating: res/color-v23/abc_btn_colored_borderless_text_material.xml
  inflating: res/color-v23/abc_btn_colored_text_material.xml
  inflating: res/color-v23/abc_color_highlight_material.xml
```

(a)



(b)

Figure 2.5: (a) Unzipping an APK file & (b) the file's contents.

Chapter 3

Light Android

This project builds on existing work done from [18] that includes a working implementation of the Light Android program, and a partially complete formal definition[1]. The Light Android program converts Android apps in Dalvik code into a lightweight program. It does this by using an abstract analysis language called Light Android. All Dalvik opcodes are converted into simpler instructions, thereby also reducing the number of different categories; more than 200 Dalvik opcodes are represented by merely 6 Light Android instructions. This makes it easier for further analysis.

This chapter deals with the design and formal definition of the Light Android program, including its syntax and operational semantics. The implementation details, specifying the procedure for the conversion of opcodes into Light Android instructions, are given in Chapter 5.

3.1 Syntax

The terms used to describe the operational semantics of the Light Android program are given in this section, along with their explanation.

s, t, e, *ret_result* $\in$ **Reg**, where Reg is the set of register names taken from Dalvik. Register names are of the form `vX`, where `X` is a number; e.g. `v0`, `v1`, `v2`. Further details are described in Section 2.2.

[1]The syntax and operational semantics presented in the next sections have been modified and elaborated upon, from the work done in [18].

- s: source register
- t: target or destination register
- e: exception-bearing register
- ret_result: the return value register, which stores the result a method returns, if any.

$m \in$ **Mtd** (*method*), where Mtd is the set of method names defined in a class, and can be an instance method or a static method.

$l \in$ **Lab** (*label*), where Lab is a finite set of labels, which are numbers that are used to denote program positions. If m is a method, l_m denotes the label l inside m.

$o \in$ **Loc** (*location*), where Loc is a set of locations, which are pointers in the stack to an object in the heap.

$f \in$ **Fld** (*field*), where Fld consists of static and instance fields.

$C \in$ **Cls** (*class*), where Cls is the set of classes, each of which define instance and static fields and methods, its superclass, and indicate the interfaces the class implements.

$T \in$ **Typ** (*method type*), where Typ is a method type used to disambiguate overloaded methods.

$v \in$ **Val** (*values*), which consist of String and Int values.

The instruction format is given below, specifying the type and number of operands of each:

$i \in$ **Ins** ::= *instructions* :

mov $t\ v$
| mov $t\ s$
| mov $t\ s.f$
| mov $t.f\ s$
| mov $t\ C.f$
| mov $C.f\ s$ *assignment*
| op
| op $t\ s$
| op $t\ s\ s'$
| op $t\ s\ v$

$\mid$ op $t\ s\ C$	*operation*
$\mid$ jmp l	
$\mid$ jmp $l\ s$	
$\mid$ jmp $l\ s\ s'$	
$\mid$ jmp $ls\ s$	*jump*
$\mid$ new $t\ C$	*allocation*
$\mid$ inv $ret\_result_{C.m:T}\ C.m : T\ xs$	*call*
$\mid$ ret	
$\mid$ ret s	*return*

A code block consists of labels with instructions (a partial mapping):

B: Lab $\rightharpoonup$ Ins *code block*

If an instruction has label l, the next instruction in a block is obtained at $l+1$. An example of a code block in a method's body can be seen below.

```
|0000:  const/4 v2, #int 1 // #1
|0001:  const/4 v1, #int 0 // #0
|0002:  goto 000e // -0003
...
```

In order to visualise the process of programs being converted into Dalvik, the following, simplistic examples are given. On the left is a Java code snippet and on the right is its Dalvik equivalent, the kind of instructions that it generates[2]. The first example is that of an infinite loop[3].

```
int x = 0;
while(true){
   x = x+2;
   }
```

```
0028:  const/4 v5, #int 0
0029:  add-int/lit8 v5, v5, #int 2
002b:  goto 0029
```

The second example calculates the factorial of a number, in this case, 6.

[2]The instructions presented here correspond to real Dalvik code generated; the code snippets are from an Android app written in Java.

[3]This example will not have a result in our semantics as it only handles terminating programs, as explained in Section 3.3.

```
int number = 6;
int fact = 1 ;
while (number > 0) {
   fact *= number;
   number -= 1;
    }
...
```

```
0014:  const/4 v3, #int 6 // #6
0015:  const/4 v0, #int 1 // #1
0016:  if-lez v3, 001c // +0006
0018:  mul-int/2addr v0, v3
0019:  add-int/lit8 v3, v3, #int -1 // #ff
001b:  goto 0016 // -0005
001c:  ...
```

There exists a mapping of a method's class, method name and type to its method definition, which consists of a tuple of arguments, body and registers. The method body consists of labels and instructions, as mentioned above. This is captured in the following configuration:

Π: Cls $\times$ Mtd $\times$ Typ $\rightarrow$ Args $\times$ (Lab $\rightharpoonup$ Ins) $\times$ Regs

The environment consists of the stack and heap. The stack contains a list of registers mapped to values and locations. The heap consists of mappings from locations and classes to fields, which in turn are mapped to values and locations.

S: Stk[Reg $\rightharpoonup$ Val $\uplus$ Loc] *stack*

H: Loc $\uplus$ Cls $\rightharpoonup$ Fld $\rightharpoonup$ Val $\uplus$ Loc *heap*

3.2 Translation – from Dalvik opcodes to Light Android instructions

This section provides an insight into the translation procedure; of Dalvik opcodes being converted to Light Android instructions. It also provides the reader with an intuition behind the translation. A fairly lax approach is adopted in this explanation to avoid being pedantic about low-level details. This includes omitting exact syntax or variants of some Dalvik opcodes as specified in the Dalvik bytecode format in [5].

For example, the disambiguating suffixes of some opcodes, as shown in Section 2.2, will not be delved into. Additionally, arguments here, such as register names, correspond to our syntax described in the above section, rather than their exact descriptions, e.g. according to their sizes (vAAAA – a destination register of 16 bits will simply be denoted as t).

Each Dalvik opcode's syntax and description is given below, along with its Light Android translation. However, we omit the explanation of opcodes dealing with arrays, for simplicity. Table 3.1 gives a bird's-eye view of the mapping from Dalvik opcodes to abstract instructions in Light Android.

All explanations below are derived from [5].

Dalvik	Light Android
move t s	mov t s
const t v	mov t v
iget t s f	mov t s.f
iput s t f	mov t.f s
sget t C f	mov t C.f
sput s C f	mov C.f s
new-instance t C	new t C
l: invoke args C.m:T	inv ret_result C.m:T args
l+1: move-result t	mov t ret_result
goto l	jmp l
if-* s l	jmp_h l s
if-* s s' l	jmp_h l s s'

Dalvik	Light Android
nop, monitor s, move-exception t, check-cast s C	op
neg t s, not t s, *-to-* t s	op_h t s
cmp, add, sub, rsub, mul, div, rem, and, or, xor, shl, shr, ushr, instance-of t s C	op_h t s s', op_h t s v, op_h t s C
*-switch s l	jmp l
l: *-switch-payload ident size key(s) ls	jmp_h ls s
return-void, throw e	ret
return ret_result	ret s

Table 3.1: Dalvik to Light Android. * refers to a wildcard character, to match all of the variations of an opcode format e.g., *-switch s l denotes both packed-switch as well as sparse-switch. Refer to specific opcode translations for more details.

Opcodes translated to `mov`:

- All opcodes of the form `move t s`, except the `move-result` and `move-exception` variants, consist of a destination and source register as arguments. They describe the operation of moving the contents of one register (source) into another (destination). Some examples include: `move`, `move/from16`, `move-wide`, etc. These opcodes are converted to "**mov *t s***" in Light Android.
- `move-result t` variants move the result of the most recent method invocation into the given register, *t*. Therefore, this must be the instruction right after an `invoke-` instruction. Otherwise, it is not valid. These opcodes include: `move-result` (for a single-word non-object result), `move-result-wide` (for a double-word result), and `move-result-object` (for an object result).

These opcodes are represented by "**mov *t ret_result***" in Light Android, where *ret_result*[4] is the return value register of the method called. The contents of *ret_result* are moved to the target register, t. Therefore, this then specifies the same operation as the above `move` opcodes, moving the contents of the source register into the target register.

- All `const t v` opcodes move a given literal value or a reference to the string/-class specified by the given index, into a register. Its first argument is the destination register, and a value or a reference is its second. Some examples include: `const-string`, `const-class`, `const/16`, etc. These opcodes are converted to "**mov *t v***" in Light Android.

- `iget t s f` performs the object instance field operation of storing an instance field, f, into the value/destination register, t, where the instance is referenced by the object register, *s*. Its arguments consist of a value register or pair (which is the destination register), an object register, and an instance field reference index. Some examples include: `iget-wide`, `iget-object`, `iget-boolean`, `iget-byte`, etc.

  These opcodes are converted to "**mov *t s.f***" in Light Android.

- `iput s t f` performs the object instance field operation of putting the source register's, *s*, contents into an instance field, f, the reference of which is stored in t. Some examples include: `iput-wide`, `iput-object`, `iput-boolean`, `iput-byte`, etc.

  The Light Android equivalent of these opcodes is "**mov *t.f s***".

- `sget t C f`, is similar to the `iget` opcodes, except it refers to the given static field operation with the static field, rather than the instance field. Some examples include `sget-object`, `sget-boolean`, `sget-byte`, `sget-char`, `sget-short`.

  These opcodes are converted to "**mov *t C.f***" in Light Android.

- `sput s C f` is similar to the `iput` opcodes, except it refers to the given static field operation with the static field, rather than the instance field. Some examples include `sput-object`, `sput-boolean`, `sput-byte`, `sput-char`, `sput-short`.

  These opcodes are represented by "**mov *C.f s***" in Light Android.

[4]Further details on this register are presented in Section 5.1.4, detailing its use and purpose.

Opcodes translated to op:

- `nop` is an opcode that does not perform any operation. It has no argument. Pseudo-instructions that carry data, such as `packed-switch-payload` and `sparse-switch-payload`, are tagged with this opcode.

  `monitor-enter s` and `monitor-exit s` obtain and release the monitor[5] of the indicated object, respectively. Its argument is a reference-bearing register.

  `move-exception t` stores an exception that is just caught into the specified register, which is the only argument of this opcode. It is only valid when it occurs as the first instruction of an exception handler and is invalid otherwise.

  `check-cast s C` consists of a register containing a reference, and a type index, as its arguments. It throws a *ClassCastException* if the reference in the given register cannot be cast to the given type.

  All of the above opcodes are mapped to the "**op**" instruction in Light Android.

- Opcodes defining unary operations such as negation and type conversions consist of a destination register, and a source register as its arguments. The identified operation is performed on the contents of the source register, the result of which is stored in the destination register. Some examples include: `not-int` (unary ones-complement), `neg-long` (unary twos-complement), `int-to-float`, `long-to-int`, etc.

  These opcodes are mapped to "$\mathbf{op}_h\ \boldsymbol{t}\ \boldsymbol{s}$" in Light Android, where h denotes the unary operation performed.

- The opcodes that compare floating point or long values consist of a destination register (t), and two source registers (s, s') containing the values to be compared, as arguments. A result of 0 is stored in t if $s == s'$, 1 is stored if $s > s'$ and -1 if $s < s'$. For example, `cmp-long`, to compare long values.

  Opcodes defining binary operations are of different kinds, mainly differing in the number and type of arguments.

  An opcode of the form `*binop* t s s'` performs a binary operation with three arguments. It consists of two source registers as operands and stores its re-

[5]A monitor is a construct for multi-threading synchronisation.

sult in a destination register. *binop* signifies an arbitrary binary operation, e.g. `add-float` (floating point addition), `sub-double` (floating point subtraction), `and-int` (bitwise AND), `or-long` (bitwise OR), `shl-int` (bitwise shift left), `shr-long` (bitwise signed shift right).

An opcode of the form *binop*`/2addr s s'` performs a binary operation with two arguments. It consists of a first source and destination register, and the second source register, as arguments. This implies that after performing the binary operation on the two source registers, the result is stored in the first. Some examples of this type include: `rem-int/2addr` (twos-complement remainder after division), `or-long/2addr` (bitwise OR), `sub-float/2addr` (floating point subtraction), `mul-double/2addr` (floating point multiplication).

All of the above opcodes are converted to "$\mathbf{op}_h$ $\boldsymbol{t}$ $\boldsymbol{s}$ $\boldsymbol{s'}$ " in the Light Android program, where *h* denotes the binary operation performed. In the Light Android design, *binop*`/2addr s s'` opcodes are also converted to this format, despite having two registers, instead of three. The first source register is duplicated and saved as the target register, *t*, which is perhaps better visualised as "$\mathbf{op}_h$ $\boldsymbol{s}$ $\boldsymbol{s}$ $\boldsymbol{s'}$".

- *binop*`/lit16 t s v` and *binop*`/lit8 t s v` perform the specified binary operation on a source register, *s*, and a literal value, *v*, of 16 bits and 8 bits, respectively, storing the result in the destination register, *t*. Some examples include: `mul-int/lit16`, `div-int/lit16`, `shl-int/lit8`, `shr-int/lit8`.

  These opcodes are converted to "$\mathbf{op}_h$ $\boldsymbol{t}$ $\boldsymbol{s}$ $\boldsymbol{v}$" in Light Android, where *h* denotes the binary operation performed.

- `instance-of t s C` consists of a destination register, a source register containing a reference, and a type index, as its arguments. 1 is stored in the destination register if the given reference is an instance of the given type, and 0 otherwise.

  This opcode is converted to "$\mathbf{op}_h$ $\boldsymbol{t}$ $\boldsymbol{s}$ $\boldsymbol{C}$", where *h* defines the operation of checking whether the given reference in the source register is of the specified type, and stores the result as mentioned above.

Opcodes translated to `jmp`:

- `goto l` specifies an unconditional jump to the indicated instruction, given in the label, *l*, which is a signed branch offset (of varying sizes). These include `goto` `l`, `goto/16` and `goto/32`.

These are converted to "**jmp *l***" in Light Android.

- Opcodes containing `if-`*`test`*`..` are similar to if-then statements in high-level programming languages. The guard of the control flow statement is a comparison of values here. If the comparison succeeds as given, the program branches to the given label.

  Opcodes of the form `if-`*`testz`*` s l` compare the contents of a register with zero. They consist of a register, *s*, for comparison, and the label, *l*, to jump to, as arguments. The *test* signifies relational comparisons, e.g. `if-eqz` (equal to zero), `if-ltz` (less than zero), `if-gez` (greater than or equal to zero), etc.

  These are mapped to the "$\mathbf{jmp}_h\ \boldsymbol{l}\ \boldsymbol{s}$" instruction, where h denotes the comparison performed with zero.

- Opcodes of the form `if-`*`test`*` s s' l` specify a comparison of two registers' values and a label to branch to if the comparison succeeds as given. The *test* signifies relational comparisons between the two source registers' values, e.g. `if-eq` (the two values are equal), `if-lt` (the first value is less than the second), `if-ge` (the first value is greater than or equal to the second), etc.

  These opcodes are converted to "$\mathbf{jmp}_h\ \boldsymbol{l}\ \boldsymbol{s}\ \boldsymbol{s'}$", where h denotes the relational comparison performed.

- Two forms of opcodes handle switch cases: `packed-switch s l` and `sparse-switch s l`. They specify a register to test, *s*, and *l*, a label of its data payload: `packed-switch-payload` and `sparse-switch-payload`, respectively. If the value of *s* is matched with an index of a table entry in the payload, it makes a jump to an instruction based on the label specified. Otherwise, the jump is made to the next label, after the original switch instruction and not after the data payload pseudo-instruction.

  The format of the payloads consists of *ident*, the identifying pseudo-opcode to determine the kind of switch; *size*, the number of table entries; *key*, the first key (the first and lowest switch case value) in the case of `packed-switch-payload`, and a list of all the keys for `sparse-switch-payload`; followed by *ls*, the list of target labels (the offset is relative to the original switch opcode and not this table).

`packed-switch s l` and `sparse-switch s l` are converted to "**jmp *l***" in Light Android, specifying the jump to the data payload table. `packed-switch-payload` and `sparse-switch-payload` are converted to "$\mathbf{jmp}_h$ ***ls s***", where *h* is the equality check of *s* and the indexes of *ls*.

Opcodes translated to `new`:

- `new-instance t C` creates a new instance of the specified type (must be a non-array class) and stores its reference in the destination register. Its arguments are the destination register and the type index.

  This is represented as "**new *t C***" in Light Android.

Opcodes translated to `inv`:

`invoke-`*`kind`*` args C.m:T` is a method call. It consists of the arguments, *args*, to be passed to the method, and the method reference index (which consists of the method's class name, method name and its type, $C.m : T$). The *kind* of this invocation depends on the type of method to be called, e.g. `invoke-virtual` (for a normal virtual method), `invoke-static` (for a static method), `invoke-interface` (for an interface method).

Another variant of this opcode specifies a range of registers as arguments. For example, `invoke-virtual/range`, `invoke-static/range`.

A result-specific `move-result` variant may follow an `invoke` instruction, to store the result of the invocation, if any.

These opcodes are represented as "**inv *ret_result C.m:T args***" in Light Android, where *ret_result* is the return value register of the callee method.

Opcodes translated to `ret`:

- `throw e` throws an exception, indicated in the register, *e*.
  `return-void` returns from a void method.

  These two opcodes are mapped to "**ret**" in Light Android.

- Other variants of the `return` instruction return from a non-void method, e.g. from a value-returning or object-returning method. These are `return`, `return-`

`wide` and `return-object`, which consist of a return value register (or register-pair, in the case of `return-wide`) as its argument.

These opcodes are represented by "**ret *s***" in Light Android.

3.3 Operational Semantics

Operational semantics of a language can be presented in different ways. Some are more abstract, containing only the terms used in the program as machine states and their evaluation. However, some are more representative of the structures manipulated by a compiler or interpreter and contain details regarding its behaviour. [35] This is demonstrated in the semantics below, which describes the operations that are carried out in the stack and heap as a result of evaluating an instruction.

The operational semantics here are given in the big-step style, also known as natural semantics, which show a term being evaluated to its final result, as opposed to a step by step transition in the small-step style[6]. This means that the semantics can only account for terminating computations. It was designed to model *abstract effects* rather than full computations, which are always terminating. Therefore, the endless loop is not modelled in this semantics.

The corresponding inference rule for each variant of the instructions are given, with their explanation. In the judgement, $S_1, H_1 \vdash l_m : \text{mov} \;\; t.f \;\; s \Downarrow S_2, H_2$ (as an example), our context or environment consists of the stack and the heap. The turnstile symbol, $\vdash$, separates the context or environment on its left from the executed instruction on its right. Overall, it can be read as – the term $l_m : \text{mov} \; t.f \; s$, in the context S_1, H_1, evaluates to S_2, H_2. Of course, this proposition only holds when the premises are satisfied.

The most common premise in the instructions is the following:

$$S_1, H_1 \vdash (l+1)_m : \_ \Downarrow S_2, H_2.$$

This shows the state of the machine moving on to the next label, after carrying out specific operations evaluating an instruction. Any changes made to the stack and heap are reflected in the updated versions. The instruction belonging to the next label can be of any form and evaluates to S_2, H_2, a different stack and heap configuration. The

[6]As introduced in Section 2.1.1.

subscript in $(l+1)_m$, or l_m in general, associates the label with its respective method, m.

$$\frac{}{S_1, H_1 \vdash l_m :\ \_\ \Downarrow S_1, H_1}\ \text{(Default)}$$

Given above is the default case that matches anything, which does nothing to change the state of the heap or stack. The program does not move to a new state or label.

3.3.1 `mov` instruction

The `mov` instruction specifies an assignment and consists of two operands. It assigns the second operand (the source) to the first (the destination or target). The following instructions are similar in operation – mapping the first operand to the second – but vary in the nature of the operands, or rather, their type.

$$\frac{\begin{array}{c} S_2 = S_1\ [t \mapsto v] \\ S_2, H_1 \vdash (l+1)_m :\ \_\ \Downarrow S_3, H_2 \end{array}}{S_1, H_1 \vdash l_m :\ \text{mov}\ \ t\ \ v \Downarrow S_3, H_2}\ \text{(a)}$$

(a) This `mov` instruction consists of a target register, t, and a value, v, as the source. The stack is updated by mapping the target register to the value. Moving on to the next label, this updated stack is used.

$$\frac{\begin{array}{c} S_2 = S_1\ [t \mapsto S_1(s)] \\ S_2, H_1 \vdash (l+1)_m :\ \_\ \Downarrow S_3, H_2 \end{array}}{S_1, H_1 \vdash l_m :\ \text{mov}\ \ t\ \ s \Downarrow S_3, H_2}\ \text{(b)}$$

(b) Similar to (a), except the source is a register and its contents must be retrieved from the stack, to which the target register, t, is mapped to.

$$\frac{\begin{array}{c} S_2 = S_1\ [t \mapsto H_1(S_1(s))\ (f)] \\ S_2, H_1 \vdash (l+1)_m :\ \_\ \Downarrow S_3, H_2 \end{array}}{S_1, H_1 \vdash l_m :\ \text{mov}\ \ t\ \ s.f \Downarrow S_3, H_2}\ \text{(c)}$$

(c) The source here consists of a field of an object in the heap, that is pointed to by a location in the stack and held in a register. So, $H(S(s))$ is the object and $H(S(s))(f)$ is the value of the contents of field, f. t is mapped to this field, and the stack is updated accordingly, moving to the next label. To elucidate this, an example is given below:

`mov v5 v7.x` – assigns the value in the field, *x*, to the target register, *v*5. The location of the field *x* is found in *v*7. This is illustrated in Figure 3.1.

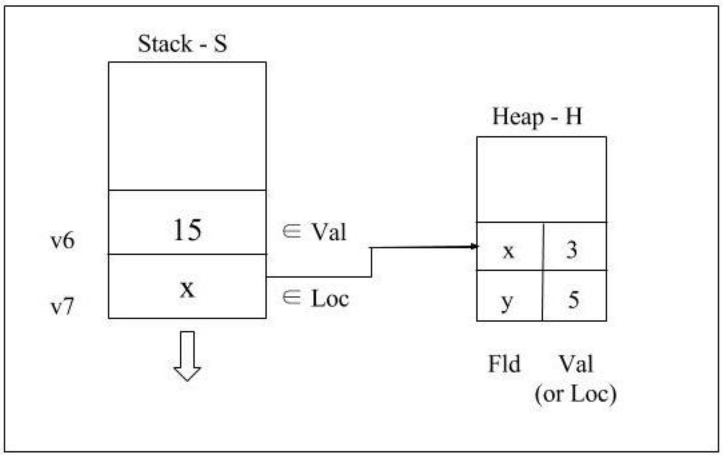


Figure 3.1: The stack and heap configuration for the example given: look up the stack for *v*7 then go to the location it points to and retrieve the value from the heap.

$$\frac{\begin{array}{c} fs \;=\; H_1(S_1(t))\;[f \mapsto S_1(s)] \\ S_1, H_1\;[S(t) \mapsto fs] \vdash (l+1)_m :\; \_ \;\Downarrow S_2, H_2 \end{array}}{S_1, H_1 \vdash l_m :\; \text{mov}\;\; t.f\;\; s \;\Downarrow S_2, H_2} \quad \text{(d)}$$

(d) This is the reverse of the the above; the location of the field, f, is pointed to by t instead of s. An updated object, fs, is created to map f to the contents of the register, s, stored in the stack. In the rest of the code, the heap is updated with this new information.

$$\frac{\begin{array}{c} S_2 \;=\; S_1\;[t \mapsto H_1(C)\;(f)] \\ S_2, H_1 \vdash (l+1)_m :\; \_ \;\Downarrow S_3, H_2 \end{array}}{S_1, H_1 \vdash l_m :\; \text{mov}\;\; t\;\; C.f \;\Downarrow S_3, H_2} \quad \text{(e)}$$

(e) The target register, t, is mapped to the (static) field pointed to by a class in the heap. The stack is updated to reflect this change.

$$\frac{fs\ =\ H_1(C)\ [f \mapsto S_1(s)] \qquad S_1, H_1\ [C \mapsto fs] \vdash (l+1)_m :\ \_\ \Downarrow S_2, H_2}{S_1, H_1 \vdash l_m :\ \text{mov}\ \ C.f\ \ s\ \Downarrow S_2, H_2}\ \text{(f)}$$

(f) This is the opposite of the above; here, the field of a class, stored in the heap, is updated to the contents of s in the stack, by the updated object, fs. The heap is updated to show the class being mapped to fs.

3.3.2 `op` instruction

The `op` instruction specifies an operation being carried out, in terms of applying a function, h, to one or more operands, and storing the result in a target register, t. For each case, the operations defined by h vary according to the opcodes they represent, as explained in Section 3.2.

$$\frac{S_1, H_1 \vdash (l+1)_m :\ \_\ \Downarrow S_2, H_2}{S_1, H_1 \vdash l_m :\ \text{op}\ \Downarrow S_2, H_2}\ \text{(a)}$$

(a) This denotes the absence of operands. Therefore, no changes are made, except moving the state of computation to the next label. It signifies an operation being carried out according to the opcode it represents but does not store its operands, if any. Opcodes being mapped to this instruction include `monitor-enter`, `monitor-exit`, `move-result`, `check-cast`, and `nop`.

The following inference rules are similar, except for the number of source operands as well as their type. Each instruction variant stores the result of the function applied to the source operand(s), in t, and then updates the stack accordingly. The source operands in (b), (c), (d), and (e), are one source register, two source registers, one source register and a value, and one source register and a class, respectively.

Examples include:

```
op v0 v5,
op v1 v7 v6,
op v3 v2 7,
op v5 v9 Laaaaaa/qjjqjj;.
```

$$\frac{S_2 = S_1\ [t \mapsto h(S_1(s))] \qquad S_2, H_1 \vdash (l+1)_m :\ \_ \ \Downarrow S_3, H_2}{S_1, H_1 \vdash l_m :\ \mathrm{op}_h\ \ t\ \ s\ \Downarrow S_3, H_2}\ \text{(b)}$$

(b) Here, h denotes a unary operation on the source register's contents. For example, h could denote the operations `not-int`, `neg-long`, `int-to-long`, `long-to-float`, etc.

$$\frac{S_2 = S_1\ [t \mapsto h(S_1(s),\ S_1(s'))] \qquad S_2, H_1 \vdash (l+1)_m :\ \_ \ \Downarrow S_3, H_2}{S_1, H_1 \vdash l_m :\ \mathrm{op}_h\ \ t\ \ s\ \ s'\ \Downarrow S_3, H_2}\ \text{(c)}$$

(c) The h here denotes either the operation of comparing two floating point or long values (as specified by the `cmp` opcodes), or a binary operation on the two source registers, such as `sub-int`, `and-long`, `add-float/2addr`. The former operation stores 0 in t if $s == s'$, 1 if $s > s'$ and -1 if $s < s'$ and the latter merely stores the result of the operation.

$$\frac{S_2 = S_1\ [t \mapsto h(S_1(s),\ v)] \qquad S_2, H_1 \vdash (l+1)_m :\ \_ \ \Downarrow S_3, H_2}{S_1, H_1 \vdash l_m :\ \mathrm{op}_h\ \ t\ \ s\ \ v\ \Downarrow S_3, H_2}\ \text{(d)}$$

(d) h here defines a binary operation on the source register and literal value, e.g. `div-int/lit16`, `rem-int/lit8`, `or-int/lit16`.

$$\frac{S_2 = S_1\ [t \mapsto h(S_1(s),\ C)] \qquad S_2, H_1 \vdash (l+1)_m :\ \_ \ \Downarrow S_3, H_2}{S_1, H_1 \vdash l_m :\ \mathrm{op}_h\ \ t\ \ s\ \ C\ \Downarrow S_3, H_2}\ \text{(e)}$$

(e) This represents the `instance-of t s C` opcode. Therefore, h defines the operation of checking whether the given reference in the source register is of the specified type. 1 is stored in t if the given reference is an instance of the given type, and 0 otherwise.

3.3.3 `jmp` instruction

The `jmp` instruction, in essence, specifies a label that the machine may jump to. This may be an unconditional jump or a conditional one, like a switch-case statement.

$$\frac{S_1, H_1 \vdash l'_m : \_ \Downarrow S_2, H_2}{S_1, H_1 \vdash l_m : \text{jmp}\ \ l' \ \Downarrow S_2, H_2} \quad \text{(a)}$$

(a) Unconditionally jump to the label specified, which can have an instruction of any type and may evaluate to a different configuration of the stack and heap. l'_m denotes label l' in m. E.g. `jmp 15`.

$$\frac{\begin{array}{c} l'' \ = \ h(S_1(s))?\ l' : l+1 \\ S_1, H_1 \vdash l''_m : \_ \Downarrow S_2, H_2 \end{array}}{S_1, H_1 \vdash l_m : \text{jmp}_h\ \ l' \ \ s \ \Downarrow S_2, H_2} \quad \text{(b)}$$

(b) Apply the unary function, h, to the source register's value. Depending on the result, either jump to the given label (l') when $true$ or move to the next label, otherwise. h is used to encode conditional branches (if statements). Here, it specifies a comparison with zero, e.g. `if-eqz`, `if-ltz`, `if-gez`.

For example, for `jmp`$_h$ `10 v1`, where the h function is `if-eqz`:

$$h(v1) = \begin{cases} true, & \text{if } v1 = 0. \\ false, & \text{otherwise.} \end{cases}$$

Here, if the result is $true$, the jump is made to the instruction with label "10" and otherwise to the next label. Others are handled similarly.

$$\frac{\begin{array}{c} l'' \ = \ h(S_1(s),\ S_1(s'))?\ l' : l+1 \\ S_1, H_1 \vdash l''_m : \_ \Downarrow S_2, H_2 \end{array}}{S_1, H_1 \vdash l_m : \text{jmp}_h\ \ l' \ \ s \ \ s' \ \Downarrow S_2, H_2} \quad \text{(c)}$$

(c) Here, h is a binary function and is applied to the two source registers' values. It either jumps to the given label or the next label, depending on the outcome of the function. h denotes a comparison between the two source registers' values, e.g. `if-eq`, `if-lt`, `if-ge`.

An example of this instruction variant: `jmp 7 v3 v4`.

$$\frac{\begin{array}{c} 0 \leqslant i < len(ls) \\ l' \;=\; h(S_1(s))?\; ls[i] : l+1 \\ S_1,H_1 \vdash l'_m :\; \_ \;\Downarrow S_2,H_2 \end{array}}{S_1,H_1 \vdash l_m :\; \text{jmp}_h \;\; ls \;\; s \;\Downarrow S_2,H_2} \quad \text{(d)}$$

(d) This is equivalent to a switch-case statement, where the cases are the elements of ls, which is a list of labels. The function, h, is applied to the source register's value, comparing it to the indexes of ls. Depending on whether there is an index matching the source register's value, it may jump to the label specified in that index or move to the next label, if there is no match.

For example: `jmp ls v0`, where $v0 = 2$, and ls = [l_1, l_2, l_3], the indexes of which are 0, 1, and 2, respectively. Here, the jump is made to l_3.

3.3.4 new instruction

The `new` instruction specifies an allocation that is made in the heap; e.g., for creating a new instance or array. A target register is specified as its first operand, which records the location of the allocation made.

$$\frac{\begin{array}{c} o \notin \text{dom}(H_1) \quad C \notin \text{dom}(H_1) \\ fs \;=\; \{f \mapsto \bot \mid f \in \text{iflds}(C)\} \\ fs' \;=\; \{f \mapsto \bot \mid f \in \text{sflds}(C)\} \\ S_2 \;=\; S_1\; [t \mapsto o] \quad H_2 = \; H_1\; [o \mapsto fs]\; [C \mapsto fs'] \\ S_2,H_2 \vdash (l+1)_m :\; \_ \;\Downarrow S_3,H_3 \end{array}}{S_1,H_1 \vdash l_m :\; \text{new} \;\; t \;\; C \;\Downarrow S_3,H_3} \quad \text{(a)}$$

(a) This variant of the `new` instruction creates a new class, which does not already exist in the domain of the heap. Two objects (maps), fs and fs', are declared to store the instance fields and static fields of the class, respectively.

The stack is updated to show t being mapped to this class's location, o, which must not already be in the heap's domain. The heap is updated to show o being mapped to the instance fields, fs, and the class, C, to its static fields, fs'.

$$\frac{\begin{array}{c} o \notin \mathrm{dom}(H_1) \quad C \in \mathrm{dom}(H_1) \\ fs \ = \ \{f \mapsto \bot \mid f \in \mathrm{iflds}(C)\} \\ S_2 \ = \ S_1\ [t \mapsto o] \quad H_2 = \ H_1\ [o \mapsto fs] \\ S_2, H_2 \vdash (l+1)_m :\ \_\ \Downarrow S_3, H_3 \end{array}}{S_1, H_1 \vdash l_m :\ \text{new}\ \ t\ \ C\ \Downarrow S_3, H_3} \ \text{(b)}$$

(b) This is similar to (a), except the class is in the domain of the heap and only fs is created for the instance fields of the class. The updated stack maps t to o and the heap maps o to fs.

3.3.5 `inv` instruction

The `inv` instruction invokes or calls a specified method. It will be explained with the help of an example - "`inv v Ljava/lang/Math;, min, (II)I, v3, v4`". The inference rule for evaluating this rule is as given below:

$$\frac{\begin{array}{c} (xs,\ B,\ rs) \ = \ \Pi\ (C,\ m',\ T) \\ fs \ = \ \{rs[f] \mapsto \bot \mid f \in [1..\#rs]\} \quad fs' \ = \ fs\ [xs\ [f] \mapsto S_1(args\ [f]) \mid f \in [1..\#xs]] \\ fs', H_1 \vdash B_{C.m':T} \Downarrow S_2, H_2 \quad S_1[ret\_result \mapsto S_2(ret\_result)], H_2 \vdash (l+1)_m :\ \_\ \Downarrow S_3, H_3 \end{array}}{S_1, H_1 \vdash l_m :\ \text{inv}\ \ ret\_result\ \ C.m' : T\ \ args\ \Downarrow S_3, H_3}$$

The instruction consists of the following operands, in order:

- *ret_result* – the return value register of this callee method. ["`v`" in the example]
- $C.m' : T$ – details of the method's name and its disambiguating class and type. ["`Ljava/lang/Math;, min, (II)I`" in the example]
- *args* – the method's argument registers. ["`v3, v4`" in the example]

m' refers to the callee method, to distinguish it from the method associated with the label, l_m. As described in Section 3.1:

$$\Pi\text{: Cls} \times \text{Mtd} \times \text{Typ} \rightarrow \text{Args} \times (\text{Lab} \rightharpoonup \text{Ins}) \times \text{Regs}$$

Therefore, the right-hand side of "$(xs,B,rs) = \Pi(C,m',T)$" yields the tuple (Args, Lab⇀Ins, Regs), which is stored in the left-hand side tuple as follows:

- xs – argument registers (Args)
- B – the code block consisting of label and instruction pairs (Lab ⇀ Ins)
- rs – registers (Regs)

$\#rs$ refers to the number of registers and $\#xs$ refers to the number of argument registers. fs creates an empty array of registers, of length $\#rs$. fs' maps registers initialised in fs, from xs to the contents of the argument registers, *args*, in the instruction, retrieved from the stack of the caller method.

With this new object (register map), fs', the code block of the invoked method, $B_{C.m':T}$, is executed, which may result in a new stack and heap configuration, S_2,H_2. The dedicated register, "`v`", for storing the return result of the invoked method, is mapped to the contents of the return value register of the callee method, retrieved from the callee's stack, S_2.

3.3.6 `ret` instruction

The `ret` instruction is a return statement. This instruction is usually at the end of the method, therefore, the state of computation does not move to a next label-instruction pair.

$$\frac{}{S_1,H_1 \vdash l_m :\ \text{ret}\ \Downarrow S_1,H_1}\ \text{(a)}$$

$$\frac{S_2\ =\ S_1\ [ret\_result \mapsto S_1(s)]}{S_1,H_1 \vdash l_m :\ \text{ret}\ s\ \Downarrow S_2,H_1}\ \text{(b)}$$

(a) Return from a void method and do nothing.

(b) Map the return value register of this method to the contents of the given source register, s, and update the stack.

Chapter 4

Light FuncDroid

The purpose of the Light FuncDroid program[1] is to extract functions from the output of Light Android. It converts the Light Android instructions to enriched lambda expressions. A functional representation is useful for carrying out subsequent analyses, to check security properties, e.g. to prove that certain undesirable features are not present in an application. The lambda calculus can also be reasoned about mathematically and helps with conducting rigorous proofs, as opposed to the light assembly code that is the output of Light Android.

The Light FuncDroid program is simpler and more concise than the Light Android program. Clearly, this is because Light Android does all of the heavy-lifting in terms of dealing with the Dalvik code and its executable file, storing and distinguishing between class names, method names, and instructions, etc. It decodes all the opcodes, which are more than 200, as opposed to just a handful that need to be translated by Light FuncDroid. Consequently, the design and implementation of Light FuncDroid is also much more straightforward.

This chapter deals with the design and formal definition of the Light FuncDroid program, including its syntax and operational semantics. The implementation details, specifying the procedure for the conversion of Light Android instructions into lambda expressions, are given in the next chapter.

[1]Design and implementation decisions were aided by [12].

4.1 Syntax

The terms used to describe the operational semantics of the Light FuncDroid program are given in this section. The syntax used here is similar to that presented in the previous chapter, for Light Android. Therefore, we do not repeat their explanation here. We denote the terms or lambda terms by the metavariable, *e*, and refer to them as expressions.

$e \in$ **Exp** ::=	*expressions* :
x	*variable*
| λ x . e	*abstraction*
| e_1 e_2	*application*
| let e_1 = e_2 in e_3	*assignment*
| if e_1 then e_2 else e_3	*conditional*
| l	*label*
| v	*value*
| ls	*list of target labels*
| args	*method arguments*
| C.m:T	*a method's class, method name and type*
| h (e_1,..,e_n)	*an operation with parameter(s)*

The variables in the body of a lambda expression are specified below. In our case, they are value, reference-bearing or object registers. We denote the list of registers allocated for a method's use as *xs*. The arguments passed to an invoked function is a list of argument registers, denoted by *args*. Essentially, they are lists of variables.

$x \in$ **Var** ::=	*variables* :
s	*source register*
| t	*target register*
| ret_result	*return value register*
| s.f, t.f, C.f	*object registers (containing fields)*
| C	*index referencing a non-array class*

The core lambda calculus grammar presented in Section 2.1.2 only consisted of lambda abstractions as values, as the simplest reduced state. The values presented below have been augmented with String, numeric, and Boolean constants.

$v \in$ **Val** ::=	*values* :
λx . e	*abstraction value*
| nv	*numeric values*
| sv	*String values*
| true	*true value*
| false	*false value*
| unit	*unit or void*

We make use of a mapping, $F(l)$, from a particular label, l, to its translated expression, e. These labels are carried forward from the Dalvik and Light Android instructions. In our translation, we introduce an expression corresponding to each label in the program. Subsequent to the translation, the labels can be unfolded (replaced by their respective expression) to resemble lexically nested functions. However, some labels remain that exhibit looping behaviour or recursive calls.

$$F(l) \rightarrow e$$

The following, simplistic examples are given, to briefly visualise the process and syntax of the translation:

On the left are Light Android instructions, and on the right are its equivalent functional expressions. The first example demonstrates basic allocation of values, and then performing an operation, `h`, e.g. addition or subtraction, using those values. Since it is a void method, there is no return value, and that is denoted by `unit`. Here, the list of method registers, *xs*, consists of `v0` and `v1`.

```
0:  const [v0] [10]
1:  const [v1] [5]
2:  op [v0] [v0, v1]
3:  ret [] []
```

```
0:  λ v0 λ v1 .  let v0 = 10 in (1) v0 v1
1:  λ v0 λ v1 .  let v1 = 5 in (2) v0 v1
2:  λ v0 λ v1 .  let v0 = h(v0, v1) in (3)
v0 v1
3:  λ v0 λ v1 .  unit
```

The second example demonstrates a method invocation, $C.m : T$, with a list of argument registers, *args*, where:

C – "`Lcom/example/app/MainActivity;`"
m – "`getResult`"
T – "`(II)I`"

args – "`v0, v1`".

This example shows storing the result of the invoked function, to the return value register, *ret_result*, which is "`v`". This value is then moved to another register, `v2`. This method then returns the value stored in `v2`.

```
0:  const [v0] [20]
1:  const [v1] [30]
2:  inv [ret_result] [Lcom/example/app/MainActivity;,
    getResult,(II)I, v0, v1]
3:  mov [v2] [ret_result]
4:  ret [] []
```

```
0:  λ v0 λ v1 λ v2.  let v0 = 20 in (1) v0 v1 v2
1:  λ v0 λ v1 λ v2.  let v1 = 30 in (2) v0 v1 v2
2:  λ v0 λ v1 λ v2.  let ret_result = (Lcom/example/app/
    MainActivity;.getResult:(II)I) v0 v1 in (3) v0 v1 v2
3:  λ v0 λ v1 λ v2.  let v2 = ret_result in (4) v0 v1 v2
4:  λ v0 λ v1 λ v2.  v2
```

4.2 Translation – from Light Android instructions to Light FuncDroid expressions

This section provides details of the translation procedure; of Light Android instructions being converted to Light FuncDroid expressions. We do not delve into the details of the input; the Light Android instructions and their syntax have been covered extensively in the previous chapter. We also continue to omit the explanation of instructions dealing with arrays, for simplicity.

Table 4.1 provides a quick look at the mapping from all of the instructions in Light Android to expressions in Light FuncDroid. The left-hand side column lists all Light

Android instruction variants, at a specific location or label, *l*. The right-hand side specifies the resultant functional expression. As mentioned earlier, *xs* refers to the list of registers a method makes use of. It can be retrieved from the data structure defined in Light Android, as will be evident in Section 5.1.2.

Light Android Instruction (at location l)	Lambda Expression
mov t v	λxs. let t = v in (l+1) xs
mov t s	λxs. let t = s in (l+1) xs
mov t s.f	λxs. let t = s.f in (l+1) xs
mov t.f s	λxs. let t.f = s in (l+1) xs
mov t C.f	λxs. let t = C.f in (l+1) xs
mov C.f s	λxs. let C.f = s in (l+1) xs
op	λxs. l+1 xs
op$_h$ t s	λxs. let t = h(s) in (l+1) xs
op$_h$ t s s'	λxs. let t = h(s, s') in (l+1) xs
op$_h$ t s v	λxs. let t = h(s, v) in (l+1) xs
op$_h$ t s C	λxs. let t = h(s, C) in (l+1) xs
jmp l'	λxs. l' xs
jmp$_h$ l' s	λxs. if h(s) then l' xs else (l+1) xs
jmp$_h$ l' s s'	λxs. if h(s, s') then l' xs else (l+1) xs
jmp$_h$ ls s	λxs. if h(s) then ls[0] xs, ..., ls[n] xs else (l+1) xs
new t C	λxs. let t = unit in (l+1) xs
inv ret_result C.m:T args	λxs. let ret_result = C.m:T args in (l+1) xs
ret	λxs. unit
ret s	λxs. s

Table 4.1: Light Android to Lambda Expressions.

There are three main types of expressions that are the result of translating Light Android instructions to lambda expressions: an assignment, a conditional statement, and a variable which contains the return value register of a method.

Most Light Android instructions are converted to an assignment expression. The target or destination is similar in all of the instructions: it is a register name, stored in a variable term. However, the source differs from one instruction to another.

`mov` *instructions:*

Light Android instructions of type "mov" denote an assignment and consist of two operands. It assigns the second operand (the source) to the first (the target). In Light FuncDroid we perform the same assignment, with the scope of this assignment being the next label.

`op` *instructions:*

Light Android instructions of type "op" specify an operation being performed, either unary or binary, containing one or two operands. An operation with no operands, `op`, simply moves to the next label; we abstract away from the operation performed. It is represented as the application of the next label to the method registers, *xs*.

All other instructions of type "op", containing one or two source registers, are translated in the same way. The result of the operation performed on the source register(s) is assigned to the target register specified. The scope of this assignment is the next label, therefore the expression consists of an application of the next label to the method registers.

`jmp` *instructions:*

Light Android instructions of type "jmp" specify an unconditional, or conditional jump to a label. Unconditional jumps are often used to exhibit looping behaviour. Conditional jumps emulate if-then statements and switch cases.

`jmp l` signifies an unconditional jump to the label, l. As a lambda expression, we denote this as the application of the this label to the method registers, *xs*.

`jmp`$_h$ `l s` and `jmp`$_h$ `l s s'` represent control flow statements, with the function, h, being its guard. As a lambda expression, we encode this in an if-then-else conditional statement. If the function evaluates to true, for a given relational comparison for the source registers (s and s'), then the jump is made to the target label, l, else the execution moves to the next label.

Instructions of type `jmp`$_h$ `ls s`, which consist of a list of target labels, *ls*, and a source register to test, s, emulate switch-case statements. The value of the source register is checked, whether it is equal to any of the indexes of the list of target labels. If an equality is found, the jump is made to the specific target label at that index, otherwise the jump is made to the next label.

As a lambda expression, we encode this in an if-then-else conditional statement, where the equality test is the guard and the true branch is the list of target labels, and the false branch is the next label, both of which are an application to the method registers.

`new` *instruction:*

The Light Android instruction "new" creates a new instance of the specified type, a non-array class, and stores its reference in the target register. In Light FuncDroid, we do not store the reference of the new instance; we only assign the value of "unit" to the target register.

`inv` *instruction:*

Instructions with `inv` are used to call or invoke methods. They consist of the class name, method name, and method type, of the method that needs to be invoked. It also specifies the arguments that are to be supplied to this method call. As a lambda expression, we encode the method details as a function, applied to the arguments, the result (return value) of which is stored in the return value register, ret_result.

`ret` *instructions:*

Light Android "ret" instructions represent return statements at the end of methods, after which there is no instruction to be executed. The variants of this instruction type are `ret` and `ret s`.

The former denotes returning from a method of type void, while the latter denotes a return statement of a non-void method. `ret s` specifies a source register, which contains the return value. The Light FuncDroid equivalent of these instructions are the term "unit" (representing void), and the source register stored in a variable, respectively.

4.3 Operational Semantics

The operational semantics[2] described here are a derivative of the Light Android operational semantics, and follow some of its terminology and structure. Therefore, we do not delve into the minutiae in our explanation, as in-depth descriptions have already been presented in Chapter 3. We continue with the big-step style or natural semantics, accounting for terminating computations only. Likewise, our context or environment also consists of the stack and the heap. However, every conclusion here presents an expression.

$$\frac{}{S_1, H_1 \vdash \lambda x \,.\, e \Downarrow \lambda x \,.\, e,\ S_1, H_1} \text{ (Abstraction Value)}$$

- This rule shows that a lambda abstraction is a value, as we have seen in the pure lambda calculus in Section 2.1.2.

$$\frac{}{S_1, H_1 \vdash v \Downarrow v,\ S_1, H_1} \text{ (Value)}$$

- Values cannot be evaluated further and therefore, no change is made to the stack and heap.

$$\frac{S_1, H_1 \vdash e_1 \Downarrow v_1,\ S_2, H_2 \qquad S_2, H_2 \vdash e_2\,[x \mapsto v_1] \Downarrow v_2,\ S_3, H_3}{S_1, H_1 \vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow v_2,\ S_3, H_3} \text{ (Assignment)}$$

- Here, we see that the expression, e_1, has value, v_1, as its final result of computation. All occurrences of the variable, x, in the term, e_2, are replaced by this value. The resultant expression has the final arbitrary value of v_2. The stack and heap configurations are updated accordingly.

[2]This operational semantics was not described in [18], and is therefore developed specifically for this project.

$$\frac{S_1,H_1 \vdash e_1 \Downarrow \lambda x\,.\,e_3,\ S_2,H_2 \qquad S_2,H_2 \vdash e_2 \Downarrow v_1,\ S_3,H_3 \qquad S_3,H_3 \vdash e_3\,[x \mapsto v_1] \Downarrow v_2,\ S_4,H_4}{S_1,H_1 \vdash e_1\ e_2 \Downarrow v_2,\ S_4,H_4}\ \text{(Application)}$$

- This rule pertains to the application of two expressions. The first, e_1, evaluates to an abstraction of a variable, x, from another expression, e_3. The second expression is shown to compute to a value, v_1, which is substituted for all instances of x in e_3. After the substitution, the resultant expression is shown to evaluate to the value, v_2. All computations update the stack and heap.

$$\frac{S_1,H_1 \vdash e_1 \Downarrow \text{true},\ S_2,H_2 \qquad S_2,H_2 \vdash e_2 \Downarrow v_1,\ S_3,H_3}{S_1,H_1 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_1,\ S_3,H_3}\ \text{(Conditional-True)}$$

- The rule above represents a conditional statement that executes the `true` branch. e_1 is the guard of the conditional statement, which evaluates to `true`. Therefore, e_2 is evaluated, which results in v_1, and is the final result of computing the conditional.

$$\frac{S_1,H_1 \vdash e_1 \Downarrow \text{false},\ S_2,H_2 \qquad S_2,H_2 \vdash e_3 \Downarrow v_1,\ S_3,H_3}{S_1,H_1 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_1,\ S_3,H_3}\ \text{(Conditional-False)}$$

- The reverse of the above rule represents a conditional statement that executes the `false` branch, as its guard, e_1, evaluates to `false`. Therefore, the evaluation of e_3 presents the final value of computation, v_1.

$$\frac{S_1,H_1 \vdash e_1 \Downarrow v_1,\ S_2,H_2 \qquad .. \qquad S_n,H_n \vdash e_n \Downarrow v_n,\ S_{n+1},H_{n+1} \qquad h(e_1,..,e_n) \Downarrow v_2}{S_1,H_1 \vdash h(e_1,..,e_n) \Downarrow v_2,\ S_2,H_2}\ \text{(Operation)}$$

- Here, the evaluation of an arbitrary function, h, is given. Its parameters are expressions, which evaluate to their respective values. Performing the operation, with the values of the expressions, results in the final value, v_2.

$$\frac{\begin{array}{c} S_1, H_1 \vdash C.m : T \Downarrow \lambda xs \,.\, e,\ S_2, H_2 \\ S_2, H_2 \vdash args \Downarrow v_1,\ S_3, H_3 \\ S_3, H_3 \vdash e\,[xs \mapsto v_1] \Downarrow v_2,\ S_4, H_4 \end{array}}{S_1, H_1 \vdash C.m : T\ args \Downarrow v_2,\ S_4, H_4}\ \text{(Method Call)}$$

- This rule pertains to a method call or invocation. A method is identified by its unique tuple of its class name, method name, and disambiguating type. The body of a method consists of expressions that are abstractions of its method registers. The values of the argument registers, *args*, are assigned to the method registers, *xs*.

$$\frac{S_1(ret\_result) = v_1}{S_1, H_1 \vdash ret\_result \Downarrow v_1,\ S_1, H_1}\ \text{(a)}$$

$$\frac{S_1(t) = v_1}{S_1, H_1 \vdash t \Downarrow v_1,\ S_1, H_1}\ \text{(b)}$$

$$\frac{S_1(s) = v_1}{S_1, H_1 \vdash s \Downarrow v_1,\ S_1, H_1}\ \text{(c)}$$

$$\frac{H_1(S_1(s))\ (f) = v_1}{S_1, H_1 \vdash s.f \Downarrow v_1,\ S_1, H_1}\ \text{(d)}$$

$$\frac{H_1(S_1(t))\ (f) = v_1}{S_1, H_1 \vdash t.f \Downarrow v_1,\ S_1, H_1}\ \text{(e)}$$

$$\frac{H_1(C)\ (f) = v_1}{S_1, H_1 \vdash C.f \Downarrow v_1,\ S_1, H_1}\ \text{(f)}$$

$$\frac{H_1(C) \;=\; v_1}{S_1, H_1 \vdash \; C \;\Downarrow\; v_1,\; S_1, H_1} \quad \text{(g)}$$

- The rules above are for variable lookups in the stack or heap, for the (a) return value register; (b) target register; (c) source register; (d), (e), (f), object registers containing fields; and (g) class index, respectively. All of these produce values.

Chapter 5

Implementation

This chapter presents the implementation of the Light Android and Light FuncDroid programs, as designed in the previous chapters. Figure 5.1 provides an idea of the implementation, specifying the steps of the procedure.

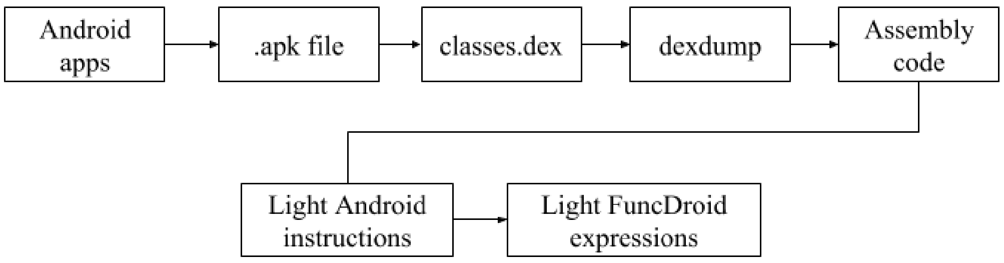


Figure 5.1: An overview of the translation procedure.

5.1 Light Android

The Light Android program takes as input a text file generated by `dexdump` using the Dalvik Executable, `classes.dex`. The file contains all of the classes defined in a particular application, along with information regarding its methods, registers, instance and static fields, instructions, etc. The file is read and processed line by line, in order to decode it.

The output consists of Light Android instructions, derived from the Dalvik opcodes in a method body, which we are most interested in. The Light Android program also stores information about all of the classes, such as the number of registers a method uses, as extracted from the `dexdump` text file.

5.1.1 Program Structure

The `Ins` (instruction) class defines the format of a Light Android instruction, which consists of an operation name, followed by a list of target(s) and a list of source(s). These lists contain register names or values.

Op_name [Target] [Source]

The `LightAndroid` class consists of multiple objects; where an object is a Scala construct for creating a unique instance or singleton object of a class. The most notable is the object `dex`, which contains all of the functions, namely `decode_line`, `decode_op` and `extra_data`, that deal with processing the input file. The data structure of Light Android consists of various objects such as `superclass`, `static_field`, `interface`, to store information about classes, including maps from a class to its superclass(es), its static field(s), and the interface(s) it implements, respectively.

These objects will be elaborated upon in the following sections[1].

5.1.2 Storing data

All components of a class have their respective object to store information. For example, the interface(s) a class implements is stored in an object, `interface`. It comprises a map with the class as the key, and a list to store the interface(s) it implements, as the value, as shown in Listing 5.1. Typical operations like adding a new entry to the map (a new class and its interfaces), updating an existing entry (to add an interface to the list, for a particular class), and printing these entries, are included. A class's superclass, static fields and instance fields are handled similarly.

e.g. Class $\rightarrow$ [Interface]

[1]Dealing with arrays and exceptions has been omitted from this explanation.

```
type Class = String
type Interface = Class
object interface {
  private val tb = Map[Class, List[Interface]]()
```

Listing 5.1: The map from a class to a list of interfaces it implements.

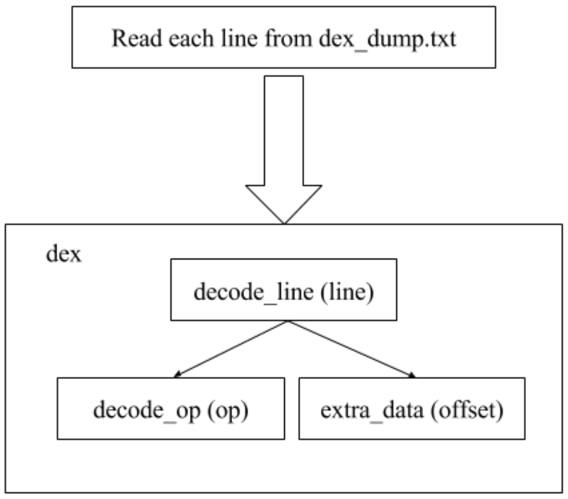


Figure 5.2: Illustration of program entry and flow. The input is the text file generated from `dexdump`, which is processed line by line to decode and store information.

The `method` object consists of a table storing its information. A method is uniquely identified by its class, name, and type, used as the map's key. The corresponding value is the tuple containing – the method's arguments, the method body containing instructions with their labels, and the method's registers. The arguments, body and registers of a particular method can be accessed individually. Functions include adding a new entry of a method with its arguments and registers into the map, and adding an instruction to the body of a particular method.

$$\textit{Class, Name, Type} \rightarrow \textit{Arguments, Body, Registers}$$

The `switch` object consists of a table of information regarding switch cases. Recall that switch cases are handled in two parts – a) the switch instruction, `packed- switch s l` or `sparse-switch s l`, specifying the register to test, s, and the label of its data payload location, l, and b) the data payload, containing the list of target labels to jump to.

The map's key is the tuple containing – a method's class, name, type, and label of the data payload. The key's value is the label of the original switch instruction (`packed-switch s l` or `sparse-switch s l`) and the register to test, s. New entries can be added, and the instruction label and source register can also be individually retrieved.

Class, Name, Type, Payload_Label $\rightarrow$ *Instruction_Label, Source_Register*

5.1.3 Dealing with input

Extracting patterns from the text file:

Regular expressions are used to extract desired patterns such as class names and method names, as well as to distinguish between opcodes. There exist maps from the regular expression patterns to its String-equivalent, for ease of use in the program.

Converting opcode names to Light Android operation names:

Each regex pattern matching an opcode is mapped to its corresponding Light Android operation name according to Table 3.1. This mapping captures the conversion of opcode names to Light Android operation names. For example, the opcodes `nop` and `goto` are mapped to "op" and "jmp" in LightAndroid, respectively.

For some of the operations – specifically, `packed-switch`, `sparse-switch` and their data payloads `packed-switch-payload` and `sparse-switch-payload` – these representations are intermediary and are not the final Light Android operation name; they need to be handled further.

Therefore, to distinguish them from the final conversions of other opcodes having the same operation name, they are annotated differently. They are dealt with in the function, `decode_line`, as detailed in the next subsection.

Decoding each input line:

The input file is processed line by line and passed as a parameter to the function, `decode_line`, in object `dex`. This function finds matches for patterns in the line and stores the value in its respective data structure. This is illustrated in Listing 5.2.

```
case Some(pat(name)) =>
  found_pattern = true
  patTag(pat) match {
    case "CLASS" => cls = name
    case "SUPERCLASS" => superclass.insert(cls, name)
```

Listing 5.2: `pat` refers to a regex pattern from the entire list of patterns defined. `patTag` maps a regex pattern to its String-equivalent. The snippet shows matching a class or superclass, and storing it.

On encountering a line with code, the offset is extracted as well as the label of the instruction. These are converted from hexadecimal to decimal. The functions, `decode_op` and `extra_data`, are called to decode opcodes, and deal with data payloads of switch cases, respectively, as illustrated in Figure 5.2.

`decode_op` is invoked to get the Light Android instruction for a particular opcode, with the opcode passed as the parameter. A Light Android instruction is returned according to our conversion in Table 3.1. The format of the instruction is as described by the `Ins` class, in Section 5.1.1.

The instruction and label are then inserted in the method body, along with the method information (its class, name, and type), in the object, `method`.

Handling switch case instructions:

Instructions dealing with switch cases are identified in `decode_line`, as they need further processing, as mentioned earlier. `packed-switch s l` and `sparse-switch s l` are converted to "jmp l". The label of this instruction, the label of its data payload, *l*, and the register to test, *s*, are stored in the object, `switch`, for later reference by its respective data payload.

For data payload instructions, `packed-switch-payload` and `sparse-switch-payload`, the extracted offset is passed to the `extra_data` function to get the target labels. This is detailed in the next subsection. The label of the switch instruction and the register to test, *s*, stored earlier, are retrieved.

These instructions are translated to "jmp ls s", where *ls* is the list of target labels relative to the original switch instruction.

Reading data payloads from the Dalvik Executable:

The `extra_data` function is used to read switch tables and consists of an offset as a parameter. Unlike the rest of the program that uses the text file from the disassembler to extract information, this function accesses the Dalvik Executable, `classes.dex`, which contains Dalvik bytecode.

This is because the data payloads of the switches, `packed-switch s l` and `sparse-switch s l`, are not given in the text file. The locations of the data payloads are specified by offsets, *l*, in these instructions, which denote the position to seek in the `classes.dex` file.

The format of the data payloads consists of the following, in order:

- an identifying pseudo opcode (in short, ident), of type, unsigned short – 0x0100 (256, in decimal) for `packed-switch-payload` and 0x0200 (512, in decimal) for `sparse-switch-payload`.
- the size, of type, unsigned short – number of table entries.
- `packed-switch-payload` consists of the first key of type, int; this is the first and lowest switch case value. `sparse-switch-payload` consists of a list of key values, sorted in ascending order.
- a list of target labels (denoted by ls) – list of branch targets, where the offset is relative to the original switch opcode and not this table.

On identifying the type of switch and retrieving the size of the table, the bytes are read according to their format, and the list of target labels is returned.

5.1.4 Opcode Details

Method invocation with a return result:

A non-void method returns a (single-word, double-word or an object) result, which is stored in its return value register by an appropriate `return` opcode. An invocation on such a method is immediately followed by `move-result t`, where *t* specifies the register to store this return value of the invoked method. However, this allocation to *t* is not made explicit in the Dalvik code, as the `move-result` opcode only specifies the target register and not the source.

In Light Android, we attempt to allude to this operation by introducing an additional argument, *ret_result*. This dedicated return value register is used in the implementation for this purpose, which stores the value from the source register in a non-void return instruction. It does not represent an actual register or its contents from the Dalvik code. The Light Android translation for `move-result t` is thus `mov t ret_result`, similar to instructions having a target and source register, and the method invocation instruction, `inv`, also contains this argument.

Abstraction of operations:

The function, *h*, is used as an abstraction; initially, we did not store the details of this function or its different instances. `op` is simply used to denote an operation, regardless of its nature, since we do not give importance to the operation performed. Similarly for `jmp`, we do not distinguish between the kinds of conditional jumps. The main purpose of this is to leave out the details that aren't necessary for our analysis, for the modelling of effects.

Opcodes specifying an allocation but not mapped to `mov`*:*

In the implementation, `sget`, `sput`, `iget`, and `iput` do not get mapped to `mov`; they retain their original opcode name. This is to differentiate their format from other `mov` instructions that do not pertain to fields, to know when to extract and deal with the field and object register. Similarly, `const`, which allocates a value to a register, retains its opcode name.

However, in the next stage of converting them to enriched lambda expressions, they do get translated to an expression denoting allocation, as they would if converted to `mov`. Therefore, it is just a design choice that affects the intermediate translation; it does not correspond to our earlier mapping, but ultimately refers to the same semantics and format.

5.1.5 Changes made to Light Android

The first step to incorporating this piece of work into the project was to test and briefly analyse it, to have a working knowledge of the program. Perhaps the simplest way to accomplish the first is via blackbox testing: providing an input to the program and observing its output, without examining the internal workings of the program. This also proves the functionality of the program. The latter was accomplished by an informal code review, which gave better insight into the workings of the program.

On testing the program with sample apps, two minor bugs were found and corrected.

- The first being the incorrect form of argument registers of a method, due to a mistake in its formatting.

- The second was the imprecise extraction of a String value, which erroneously included the String's hashcode in the name.

Light Android was designed to primarily focus on providing an abstraction for modelling effects, as mentioned earlier. However, in this project, we attempt to evaluate programs with the help of evaluators, as detailed in the next chapter. A code review revealed the need for a few changes and additions, due to this difference in purpose. These include:

- Changing the data structure for the switch cases, to store the indexes/keys of the list of target labels. This was initially absent, as determining which target label the program jumps to, was not required. This change needed to be implemented to compare the source register's value with the indexes.
- The list of target labels returned for the data payloads of switch cases were not in order, and therefore had to be corrected. This was necessary to precisely determine the target label according to its index, for evaluation.
- One major change was storing the operation performed for each instruction. This was discarded in the earlier implementation as values did not need to be computed. This is especially important for evaluating `op` instructions, to determine the operation to be performed.
- Similar to the above, another change was made in the pattern extraction for opcodes translated to the `jmp` instructions. They were only distinguished based on them carrying out a unary or binary operation, omitting the specific comparison that is to be performed. New patterns were introduced to capture all of the variations to store them separately.

5.2 Light FuncDroid

The Light FuncDroid program processes the output of Light Android, which is a list of label and instruction pairs. As its output, it translates the instructions into lambda expressions. The Light FuncDroid program was implemented from scratch for this project, and the overall code is given in A.1.

5.2.1 Program Structure

The first implementation step is to retrieve the instructions from Light Android, for a particular method; using its class name, method name and type. The program then retrieves the registers used by the method, and its body, which contains the required list of labels and instructions. These labels and instructions are then serialised and stored, before being translated into Light FuncDroid expressions. This process can be visualised in Figure 5.3. More specific details about the procedure are given in the following sections.

5.2.2 Storing data

Serialising instructions:

The labels in Dalvik code are not consecutive. Therefore, simply incrementing the label in the expressions, to refer to the next instruction, would be erroneous. Thus, all of the instructions are stored with an index, which we use for incrementation. An instruction can then be retrieved by its label or index.

$$\textit{Class, Name, Type, Index} \rightarrow \textit{(Label, Instruction)}$$

$$\textit{Class, Name, Type, Label} \rightarrow \textit{Index}$$

Storing the lambda expressions:

The instructions are processed, one by one, to be translated into lambda expressions, as given in Table 4.1. However, to aid this translation, we make use of an expression class, to define a clear data structure. It helps to define the format of a lambda expression, as well as to store the expression in the form of distinct terms. These terms correspond to the syntax we have described in 4.1. This makes it easier for further processing, such as for evaluation.

For example, an abstraction is made distinct from other terms, by its specific case class called `Abs`, which takes a list of terms (e.g. variables) followed by another term (the body of the abstraction). Register names are stored as variables, denoted by the term `Var`. A code snippet for this is shown below.

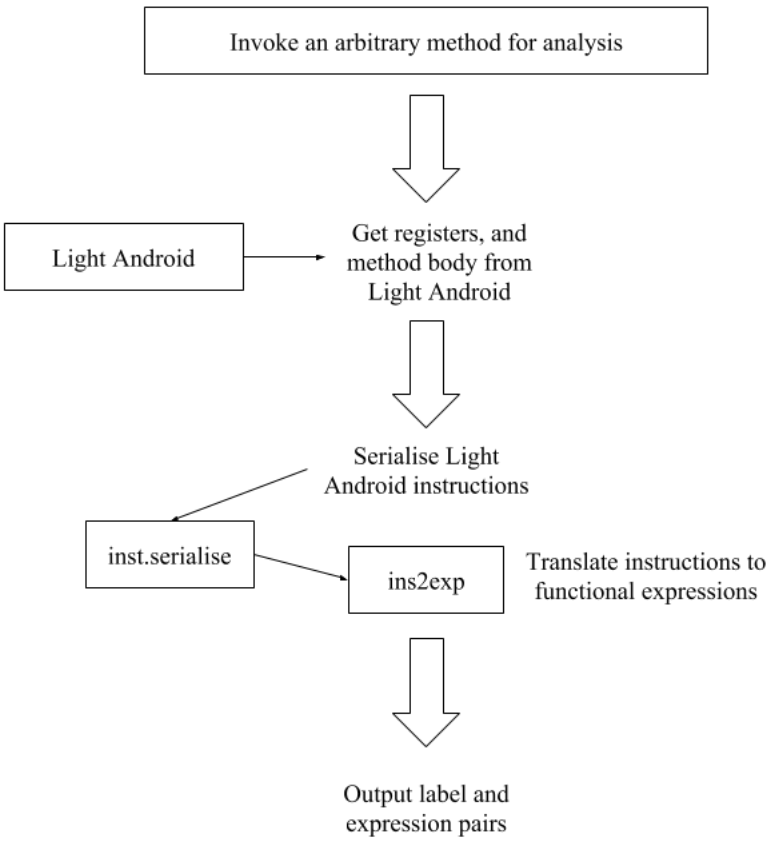

Invoke an arbitrary method for analysis
Light Android
Get registers, and method body from Light Android
Serialise Light Android instructions
inst.serialise
ins2exp
Translate instructions to functional expressions
Output label and expression pairs

Figure 5.3: Illustration of program entry and flow, beginning with an arbitrary method call. The body and registers of this method are obtained from the Light Android data structure. The instructions in the body are serialised and translated to functional expressions.

```
abstract class Exp
  case class Abs (xs:List[Exp], e:Exp) extends Exp {override def
     toString = "\\" + xs.map(x => x.toString).foldLeft("")(_+ " " +_)
     + " . " + e}
  case class Var (s:String) extends Exp {override def toString = s}
```

The lambda expressions are then stored, along with the operation performed, with their respective label as the key, as shown below. Functions include getting the operation or expression, by specifying – a given label, class name, method name, and its type.

Class, Name, Type, Label $\rightarrow$ *(Operation, Expression)*

5.2.3 Translating Light Android instructions

As mentioned earlier, a Light Android instruction consists of an operation name, followed by a list of target(s) and a list of source(s). The operation name is identified through pattern matching. After that, the instruction variant must also be recognised, for it to be handled correctly. This can be done by examining the lists of targets (accessed by `ins.ta` in the code) and sources (accessed by `ins.src` in the code), which have differentiable characteristics for each instruction variant.

The resultant expression for each instruction is of the same form: an abstraction of the method registers from its body, which can be any other term. The operation names, and how some of their instruction variants are distinguished between and handled, are briefly discussed below.

- **jmp :**

  An unconditional jump, or an instruction of type `jmp l`, is recognised by its empty sources' list. The translated expression's body consists of an application of the specified label, to the method registers, `xs`. The expression is encoded in a term, `Cond`, that represents a conditional branch or if-statement. Here, `Star()` signifies the unconditional nature of the jump, and is equivalent to the Boolean "true" in a conditional guard. Its usage is shown in the code extract below.

  ```
  if (ins.src.length == 0) { // source list is empty, unconditional
     jump
    val exp = Abs(xs, Cond(Star(), ins.ta.map(x => App(Lab(x.toInt),
        xs))))
  ```

  A conditional jump may consist of one or two source registers to test and are of the form `jmp l s` and `jmp l s s'`. Their targets' list consists of one target label.

  The translated expression's body consists of the term, `Cond`, that represents a conditional branch or if-then-else statement. Here, the guard of the conditional is an operation on the source register(s) from the list of sources. Possible jumps include: to the specified label if the comparison evaluates to "true", or to the next label otherwise; these are stored in the form of applications to the method registers. This is shown in the code extract below.

```
else if (ins.ta.length == 1) { // conditional jump, only one target
    label
  val exp = Abs(xs, Cond(Op(ins.src.map(x => Var(x))),
                         App(Lab(next_label), xs)::
                         ins.ta.map(x => App(Lab(x.toInt), xs))))
```

The resultant lambda expression's body for `jmp ls s` consists of the term for a conditional statement: with the equality test to perform and check, and the application of the target labels to the method registers as its "true" branch, and the application of the next label to the method registers as its "false" branch. Here, the next label is the one after the original switch instruction, and not the data payload table.

```
else { // switches, more than one target label

  // get label of the original switch instruction (packed-switch
     or sparse-switch)
  val orglabel = la.switch.orglabel(cls, name, typ, l)
  .
  .
  val exp = Abs(xs, Cond(Op(ins.src.map(x => Var(x))),
                         App(Lab(next_label), xs)::
                         ins.ta.map(x => App(Lab(x.toInt), xs))))
```

- **op :**

  The translated expression for `op` contains the application of the next label to the method registers, as shown below.

```
if (ins.ta.isEmpty && ins.src.isEmpty) { // no operands
  val exp = Abs(xs, App(Lab(next_label), xs))
```

The lambda expression for all other instructions of type "op" consists of an assignment to the target register, with the `Let` term. The value of this assignment is the result of the operation performed on the source register(s). The scope of this assignment is the next label, therefore the expression consists of an application of the next label to the method registers. The formation of the translated expression can be seen below.

```
else { // src.length >=1
   val exp = Abs(xs, Let(Var(ins.ta.head),
                         Op(ins.src.map(x => Var(x))),
                         App(Lab(next_label), xs)))
```

- **ret :**

  `ret` and `ret s` are distinguished by the list of sources. This is because the former has an empty list, as it does not return anything.

  The translated expressions for these instructions thus either include a term of type "Unit" or a variable term containing the return value register, as demonstrated below.

  ```
  if (ins.src.isEmpty) { // returns unit (nothing)
     val exp = Abs(xs, Unit())
      .
      .
  else { // assuming the source list contains only one variable
     val exp = Abs(xs, Var(ins.src.head))
  ```

- **inv :**

  The sources' list consists of the identifying details of the invoked method, appended with the arguments. The targets' list consists of a default return register.

  The resultant lambda expression for this instruction consists of a function, which is the invoked method, applied with the arguments; the result or return value of which is assigned to the target register. The method is defined as a function (`Fun`) with its class name (`Cls`), method name (`Nam`), and its disambiguating type (`Typ`). The scope of this is the next label applied to the method registers. The corresponding implementation for this can be seen below.

  ```
  val s = ins.src // src = [cls, name, typ] ++ args
  val t = s.tail.tail.tail // getting the args from the src
  val exp = Abs(xs, Let(Var(ins.ta.head),
                        App(Fun(Cls(s(0)), Nam(s(1)), Typ(s(2))),
                            t.map(x => Var(x))), // C.m:T args
                        App(Lab(next_label), xs)))
  ```

- **mov :**

  The value of the source register is assigned to the target register specified. Both registers are stored as variables, in the term, `Var`. The scope of this assignment is the next label, therefore the expression consists of an application of the next label to the method registers. This can be seen in the code extract below. The other "mov" instruction variants are handled similarly.

  ```
  val exp = Abs(xs, Let(Var(ins.ta.head),
                        Var(ins.src.head),
                        App(Lab(next_label), xs)))
  ```

- **new :**

  The list of targets consists of the target register, specified for storing the reference of the new instance created. It is assigned the value, "unit". The expression makes use of the `Let` term, for the assignment.

  ```
  // new t C
  val exp = Abs(xs, Let(Var(ins.ta.head),
                        Unit(),
                        App(Lab(next_label), xs)))
  ```

Chapter 6

Evaluation

Chapters 3 and 4 describe the operational semantics for the programs, Light Android and Light FuncDroid, respectively, which determine how terms are evaluated. This chapter follows through with a partial implementation. It attempts to display the correspondence between the translations, such that the meaning of the original program is retained. We show the transition from an Android application written in Java, to it being compiled to Dalvik code, then translated to Light Android instructions, and finally converted to lambda expressions.

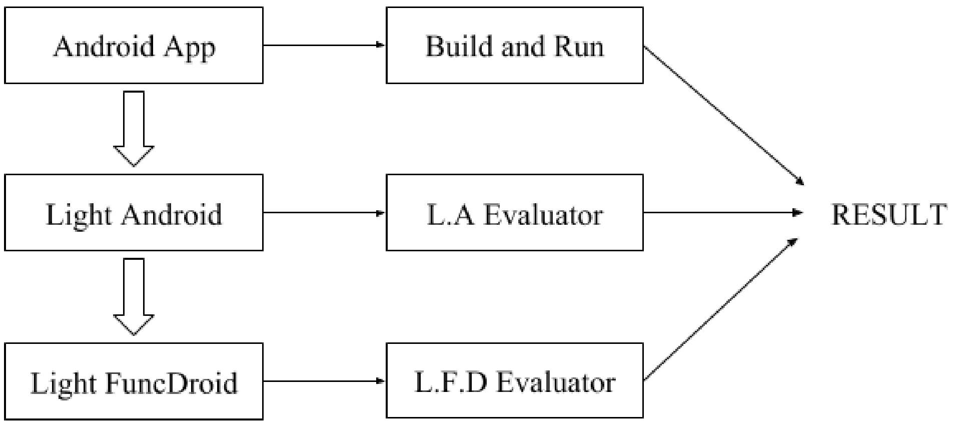


Figure 6.1: We show that all three test routes evaluate to the same result.

The Android app is built and run, to display the result of the program. Evaluators have been designed that present the results of evaluating Light Android instructions and Light FuncDroid expressions. This is done specifically for test cases that are simple.

For the purposes of this work, we shall restrict our attention to integer operations and manipulating String values only. Additionally, only the stack is modelled; thus, operations involving instance and static fields, and classes are not dealt with.

The following sections provide an overview of the implementation details of the evaluators, after which the test cases and the results are presented.

6.1 Implementation Overview

Both of the evaluators are similar in design, and consist of four main objects that carry out tasks as given below.

Invoking methods:

An invoke function takes a class name, method name, and its type as parameters. The Light Android evaluator retrieves the specified method's body from the Light Android program, which returns a list of label and Light Android instruction tuples. It also stores the last invoked method's details for future reference, specifically for the `move-result` instructions that deal with the result, or return value, of the invoked method. The Light FuncDroid evaluator retrieves the method's body from the Light FuncDroid program, which returns a list of label and lambda expression tuples.

Instruction storage and handling:

The labels in Dalvik code are not consecutive. Therefore, we map all of the instructions to a serialised index, so as to know which instruction comes next, with a simple increment. An instruction/expression can then be retrieved with a specific index or its label.

$$\textit{Class, Name, Type, Index} \rightarrow \textit{(Label, Instruction)}$$

$$\textit{Class, Name, Type, Label} \rightarrow \textit{Index}$$

Storing register names and values:

The register names used in instructions are stored with their values. Functions include inserting a new register and value pair, and retrieving the value of a specified register. Since the evaluators also deal with method invocations, we avoid the overlap of similar register names by specifying the disambiguating method details (its class, name and type).

$$\textit{Class, Name, Type, Register} \rightarrow \textit{Value}$$

Decoding and evaluation:

The evaluation starts by computing the value of the first instruction/expression in a method body and then follows the program execution flow, incorporating loops and jumps. Decoding the instructions or expressions thus returns the next index to move to, as shown in Listing 6.1. Encountering a return instruction terminates the evaluation as expected, and the return value of a method can be used as our final result for comparison.

```
while (index >= 0 && index <= size) {
   var ins = inst.get_ins(cls, name, typ, index)
   index = decode(cls, name, typ, index, ins)
}
```

Listing 6.1: The evaluation procedure; receiving the next index after decoding an instruction (for the Light Android evaluator), starting from the first.

6.1.1 Light Android Evaluator

As mentioned earlier, the evaluation starts from the first Light Android instruction. Pattern matching with case classes is used to identify the instruction type and is dealt with accordingly. It is also defined whether the evaluation moves to the next instruction or to a specified label, depending on the type of instruction.

For example, a `const` instruction that allocates a value to a register always moves to the next instruction, whereas a `jmp` instruction may move to a different label in the next step, and this label must be determined based on the particular testing operation. Further details on how each instruction is dealt with are given below.

- **const:**

  Instructions of this type involve a target register and a value. A new entry is inserted in the registers' mapping, assigning the given value to the register name, as shown in the code excerpt below.

  ```
  case "const" =>
  register.insert(cls, name, typ, targets.head, sources.head)
  ```

- **mov :**

  Instructions of this type move the contents of the given source register to the target register. The register mapping updates the target register's value to that of the source register's.

  However, the instruction that deals with moving the result of the most recent method invocation, `move-result`, is dealt with differently. As alluded to earlier, the last invoked method's details are stored for this very purpose. Its return value is then retrieved from the default return value register. The value is stored in the target register.

  These operations can be seen in the code snippet below.

  ```
  case "MOV" =>
    val va = register.get_val(cls, name, typ, sources.head)
    register.insert(cls, name, typ, targets.head, va)

  case "MOVRESULT" =>
    // get the class, name and type of the last invoked method
    val inv_cls = inv.get_class
    val inv_name = inv.get_name
    val inv_typ = inv.get_typ
     // get the value in its return value register
    val va = register.get_val(inv_cls, inv_name, inv_typ, def_reg)
    register.insert(cls, name, typ, targets.head, va)
  ```

- **op :**

  Instructions of this type specify an operation to be performed; it can be unary or binary. The operation is identified and then the operands are used to compute the result. The resulting value is stored in the specified target register.

For example, a binary operation for addition with three arguments; the first being the target register and the other two being source registers, which serve as the operands. The code corresponding to this operation can be seen below.

```
case "ADD3" =>
   val dest = targets.head
   val operands = sources.map(x => register.get_val(cls, name, typ,
       x))
   register.insert(cls, name, typ, dest,
       (operands.map(_.toInt).reduceLeft((x, y) => x +
       y)).toString())
```

- **jmp :**

  Instructions of this type specify an unconditional or conditional jump to an instruction label, and are dealt with differently depending on the particular test function.

  An example of a conditional jump can be seen below, that specifies a relational comparison between the source register's value and zero, that it is greater than or equal to zero. If the guard evaluates to true, then the jump is made to the specified label. Otherwise the index of the next instruction is returned.

```
case "IFGEZ" =>
   val l = targets.head // target label
   val s0 = register.get_val(cls, name, typ, sources.head).toInt
   if (s0 >= 0) inst.get_index(cls, name, typ, l)
   else i + 1
```

- **ret :**

  On encountering a return instruction, the evaluation is terminated. The program does not move to any next instruction. If a method returns a value, then this value is stored in its dedicated return value register.

```
val def_reg = "v" // default return value register
.
.
```

```
case "RETVOID" =>
case "RET" =>
  val va = register.get_val(cls, name, typ, sources.head)
  register.insert(cls, name, typ, def_reg, va)
```

- **inv :**

  Instructions of this type invoke a method, using its unique information: class name, method name, and type. If a method is invoked with arguments, then the values of these arguments must be passed to the callee method.

  In order to do this, we retieve the argument registers defined for the callee, and then assign them values of the caller methods' argument registers. This process can be visualised by a code excerpt given below.

```
// argument registers passed by the caller method
val arg = sources.tail.tail.tail

// get the callee method's registers
val arg_reg = la.method.args(n_cls, n_name, n_typ) match {
  case Some(lst) =>
    if (!lst.isEmpty) {
      for (i <- 0 to arg.length - 1) { // for each argument
          register passed by the caller
        var va = register.get_val(cls, name, typ, arg(i)) // get
            its value
        register.insert(n_cls, n_name, n_typ, lst(i), va) //
            store in callee method's registers
      }
    }
  case None =>
}
```

6.1.2 Light FuncDroid Evaluator

The Light FuncDroid evaluator mimics the structure and design of the Light Android evaluator, reusing components that deal with certain operations, with slight modifications only. Evaluation starts from the first expression and moves according to the program flow, depending on the kind of expression encountered. There are three main types of expressions that are the result of translating Light Android instructions to lambda expressions: an assignment, a conditional statement, and a variable which contains the return value register of a method. The details on how these expressions are evaluated are given below:

Assignment

Most Light Android instructions are converted to Light FuncDroid expressions in the form: let *target* = *source* in *scope*.

- **Target:**

  The destination is in the form of a target register's name stored in a variable term, which is retrieved as shown below.

  ```
  // destination
  ea match {

    // register
    case Var(t) =>
      target = t

    case _ =>
  }
  ```

- **Source:**

  The source differs and can be of different types, involving different procedures for evaluation.

  (a) *Source - constant value and source register:*

  It can be a source register's value or a constant value, which are extracted as shown below.

```
// source
eb match {
   // register
   case Var(s) =>
      source = register.get_val(cls, name, typ, s)

    // value
    case Const(s) =>
      source = s
```

(b) *Source - return value of invoked method:*

It can be the return value of an invoked method. The arguments are register names stored as variables. The method is defined as a function with its class name, method name, and its disambiguating type.

We pass the list of argument registers to our `invoke` function, which then performs the invocation. The processing of the argument registers and the invocation is similar to that in the Light Android evaluator[1]. After that, we can retrieve the return value of the method and use it as our source value.

```
// invoke
case App(Fun(Cls(n_cls), Nam(n_name), Typ(n_typ)), ls) =>

   var arg_reg = List[String]()

   ls.foreach (elem => elem match {
   case Var(x) =>
      arg_reg = x :: arg_reg // collect each argument register
   case _ =>
   })
   arg_reg = arg_reg.reverse

   invoke(n_cls, n_name, n_typ, h, arg_reg)
   source = register.get_val(n_cls, n_name, n_typ, target) // get
       the return value of the invoked method
```

[1]Refer to the evaluation of the `inv` instruction in sub-section 6.1.1, for the details.

(c) *Source - result of a specified operation:*

It can be the result of a specified operation with one or more operands, which are stored as register names. We retrieve the operand registers and pass it to our `op` function, that performs the operation with the operands and returns the result. The computation is handled similarly as in the Light Android evaluator[2].

```
// op
case Op(ls) =>
  var operand_reg = List[String]()
  ls.foreach( elem => elem match {
    case Var(x) =>
      operand_reg = x :: operand_reg // collect each register as
         an operand
    case _ =>
  })

  operand_reg = operand_reg.reverse
  source = op(cls, name, typ, h, operand_reg)
```

If-then-else conditional

Light Android "jmp" instructions are represented by if-then-else statements in Light FuncDroid. There are two kinds of conditional expressions to be evaluated, the details of which are given below.

- **Unconditional jump:**

  An unconditional jump consists of a `Star()` term as its condition, which represents the unconditional nature of the jump, or that it always evaluates to its true branch. The index of the specified label is retrieved and returned, for the evaluation to proceed to that index.

```
// unconditional jump
case Star() =>
  es.head match { // get the label to jump to
    case App(Lab(l), xs) => lfd.inst.get_index(cls, name, typ,
       l.toString)
    case _ => -1 }
```

[2]Refer to the evaluation of the `op` instruction in sub-section 6.1.1, for an example.

- **Conditional jump:**

  A conditional jump involves performing a relational comparison with one or two operands. The operands are extracted in a similar fashion as in the case of dealing with a unary or binary operation, as demonstrated previously.

  The index of the label specified, if the test succeeds, is stored as `jump_index`. The index of the next label, if the test fails, is stored as `next_index`.

```
// get the label to jump to, if test succeeds
es.last match {
   case App(Lab(l), xs) =>
      jump_index = lfd.inst.get_index(cls, name, typ, l.toString)
   case _ =>
}

// get the next label to jump to, if test fails
es.head match {
   case App(Lab(l), xs) =>
      next_index = lfd.inst.get_index(cls, name, typ, l.toString)
   case _ =>
}
```

  The specified test is identified and performed. Accordingly, either the `jump_index` is returned, or the `next_index`. The tests are carried out in a similar fashion as presented for the Light Android evaluator[3].

  However, the next label to jump to, if the condition evaluates to false, is different for conditional jumps representing switch cases. The next label is with respect to the original switch instruction. Additionally, the label to jump to is determined from a list of target labels, as opposed to just one. Therefore, determining the `jump_index` is handled differently, comparing the operand's (source register's value) with the indexes of the list of target labels.

Return value register

A return instruction from a non-void method consists of a register that contains the return value. This register is encoded as a variable, in Light FuncDroid. On encounter-

[3]Refer to the evaluation of the `jmp` instruction in sub-section 6.1.1, for an example.

ing this expression, we retrieve the value from the specified register and store it in the default return value register, *ret_result*. Then, the evaluation stops as we have reached the end of the method.

A lambda expression with just `unit` as its term represents a void method's return instruction, that does not return anything. On encountering this expression we simply stop the evaluation, without any further processing.

6.2 Test Cases

This section presents test cases for evaluation. We present the original program written in Java and the result of running it, along with its compiled Dalvik code. The resultant Light Android instructions, and the Light FuncDroid expressions are also given. The results of their evaluation are presented in the form of register values.

The purpose is to compute values, and ultimately to demonstrate the similarity in results between the different translations, and as compared to the original program. The examples appear to be trivial but perform several operations. They cover a range of Dalvik opcodes, including: addition, subtraction, division, finding the remainder, relational comparisons, etc.

It also covers varied program structures, such as loops and conditionals, including them being nested. Its simplicity allows one to clearly see what the expected result should be, at once.

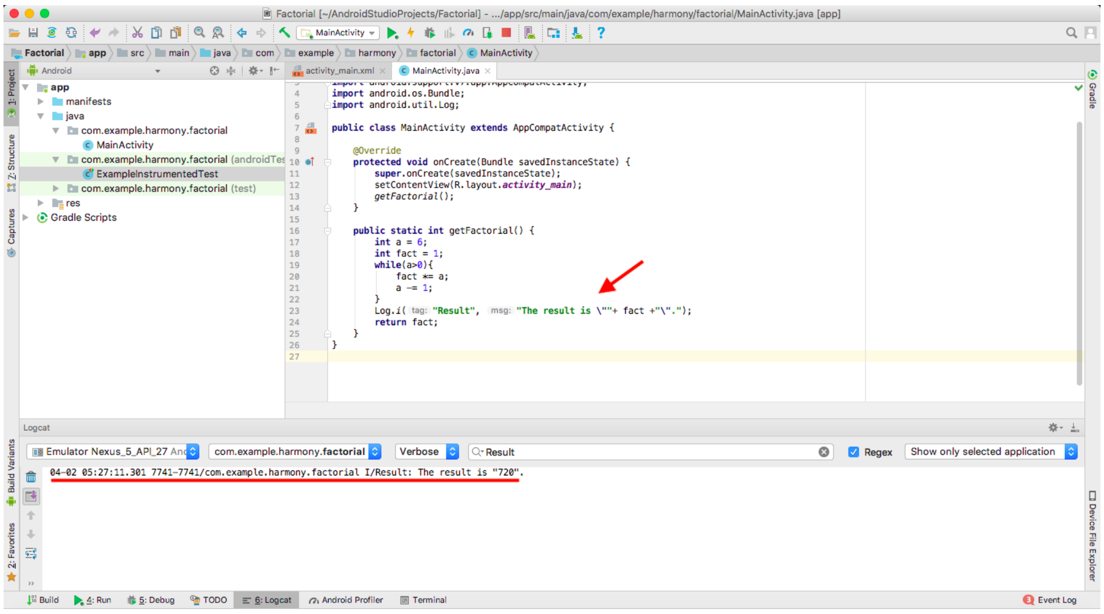
Experimental set-up:

- Test data: 10 Android apps and their APK files.
- Environment: Mac OS X 10.13.3, Android Studio 3.1, Scala IDE 4.4.1.

Procedure:

- Build and run the Android app on a device or emulator. View the result in the Logcat [8] window, in Android Studio.

  The Log class is used to write messages, which can then be viewed as shown in Figure 6.2.



```
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        getFactorial();
    }

    public static int getFactorial() {
        int a = 6;
        int fact = 1;
        while(a>0){
            fact *= a;
            a -= 1;
        }
        Log.i( tag: "Result",  msg: "The result is \""+ fact +"\".");
        return fact;
    }
}
```

Logcat: Emulator Nexus_5_API_27 | com.example.harmony.factorial | Verbose | Result | Regex ✓ | Show only selected application

```
04-02 05:27:11.301 7741-7741/com.example.harmony.factorial I/Result: The result is "720".
```

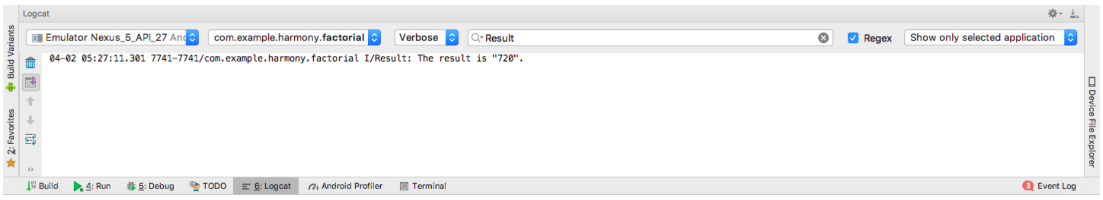
Figure 6.2: A Log.i (information) message consists of a tag and a message. The tag can then be searched in the Logcat window, to display the message.

- Build and unzip the app's APK file[4].
- Use `dexdump` to convert the `classes.dex` file into a readable text file.
- Record the Dalvik code compiled.
- Use the `classes.dex` and `dexdump` text files as input to the translation programs.
- Record the instructions and expressions generated, and run the evaluators.
- Observe and compare results.

6.2.1 Factorial

This test case computes the factorial of a given number, which in this case is 720 for the number, 6. A while loop is used to perform the calculation. We can see from the evaluator results that both the Light Android instructions, as well as the Light FuncDroid expressions, return the same result – 720, when evaluated. This result is stored in the register, `v1`. The program also returns "720" as its result, as can be seen below.

[4]Refer to Figures 2.4 and 2.5.


Emulator Nexus_5_API_27 And · com.example.harmony.factorial · Verbose · Result · Regex [✓] · Show only selected application

```
04-02 05:27:11.301 7741-7741/com.example.harmony.factorial I/Result: The result is "720".
```

Program in Java:

```java
int a = 6;
int fact = 1;
while(a>0){
   fact *= a;
   a -= 1;
}
return fact;
```

Dalvik code:

```
0000:  const/4 v0, #int 6
0001:  const/4 v1, #int 1
0002:  if-lez v0, 0008
0004:  mul-int/2addr v1, v0
0005:  add-int/lit8 v0, v0, #int -1
0007:  goto 0002
0008:  return v1
```

Light Android instructions:

0: const [v0] [6]

1: const [v1] [1]

2: $\texttt{jmp}_{if\text{-}lez}$ [8] [v0]

4: $\texttt{op}_{mul\text{-}int/2addr}$ [v1] [v1, v0]

5: $\texttt{op}_{add\text{-}int/lit8}$ [v0] [v0, -1]

7: jmp [2] []

8: ret [] [v1]

Light Android Evaluator result:

```
v0 -> 0
v1 -> 720
```

Light FuncDroid expressions:

0: λ `v0` λ `v1` . `let v0 = 6 in (1) v0 v1`

1: λ `v0` λ `v1` . `let v1 = 1 in (2) v0 v1`

2: λ `v0` λ `v1` . `if` $\texttt{op}_{if\text{-}lez}$`(v0) then (8) v0 v1 else (4) v0 v1`

4: λ `v0` λ `v1` . `let v1 =` $\texttt{op}_{mul\text{-}int/2addr}$`(v1, v0) in (5) v0 v1`

5: λ `v0` λ `v1` . `let v0 =` $\texttt{op}_{add\text{-}int/lit8}$`(v0, -1) in (7) v0 v1`

7: λ `v0` λ `v1` . `(2) v0 v1`

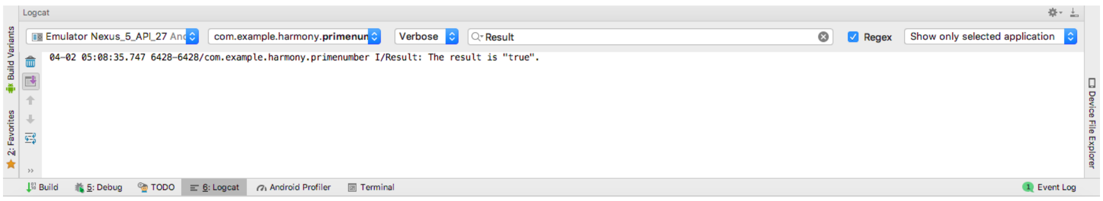
8: λ `v0` λ `v1` . `v1`

Light FuncDroid Evaluator result:

```
v0 -> 0

v1 -> 720
```

6.2.2 Prime number

This test case checks if a given number is prime or not. It consists of a nested conditional in a loop that checks whether the number is divisible by another number, other than 1 and itself. "true" is returned if the number is prime and, "false" otherwise. We can see from the evaluator results that both the translation programs return "true" for the number 11, which is a prime number. This value is stored in the register, `v2`. The program also returns "true" as its result, as can be seen below.



```
04-02 05:08:35.747 6428-6428/com.example.harmony.primenumber I/Result: The result is "true".
```

Program in Java:

```
int number = 11;
String result = "true";
for(int i=2; i<number; i++){
   if(number%i==0){
      result = "false";
      break;
   }
}

return result;
```

Dalvik code:

```
0000:  const/16 v1, #int 11
0002:  const-string/jumbo v2, "true"
0005:  const/4 v0, #int 2
0006:  if-ge v0, v1, 000f
0008:  rem-int v3, v1, v0
000a:  if-nez v3, 0010
000c:  const-string/jumbo v2,
"false"
000f:  return-object v2
0010:  add-int/lit8 v0, v0, #int 1
0012:  goto 0006
```

Light Android instructions:

```
0:    const [v1] [11]
2:    const [v2] ["true"]
5:    const [v0] [2]
6:    jmp_if-ge [15] [v0, v1]
8:    op_rem-int [v3] [v1, v0]
10:   jmp_if-nez [16] [v3]
12:   const [v2] ["false"]
15:   ret [] [v2]
16:   op_add-int/lit8 [v0] [v0, 1]
18:   jmp [6] []
```

Light Android Evaluator result:

```
v0 -> 11
v1 -> 11
v2 -> "true"
v3 -> 1
```

Light FuncDroid expressions:

```
0:   λ v0 λ v1 λ v2 λ v3 .  let v1 = 11 in (2) v0 v1 v2 v3
2:   λ v0 λ v1 λ v2 λ v3 .  let v2 = "true" in (5) v0 v1 v2 v3
5:   λ v0 λ v1 λ v2 λ v3 .  let v0 = 2 in (6) v0 v1 v2 v3
```

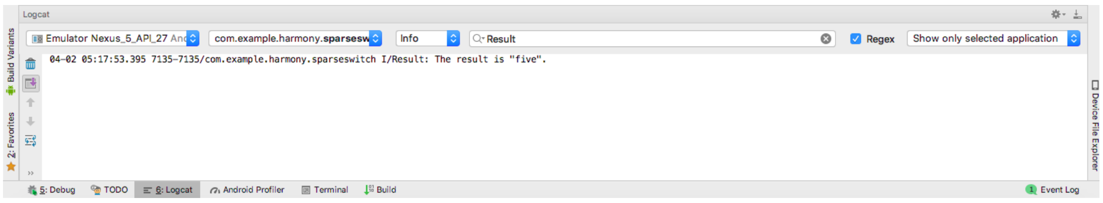
```
6:   λ v0 λ v1 λ v2 λ v3 .  if op$_{if-ge}$(v0, v1) then (15) v0 v1 v2 v3
     else (8) v0 v1 v2 v3
8:   λ v0 λ v1 λ v2 λ v3 .  let v3 = op$_{rem-int}$(v1, v0) in (10) v0 v1
     v2 v3
10:  λ v0 λ v1 λ v2 λ v3 .  if op$_{if-nez}$(v3) then (16) v0 v1 v2 v3
     else (12) v0 v1 v2 v3
12:  λ v0 λ v1 λ v2 λ v3 .  let v2 = "false" in (15) v0 v1 v2 v3
15:  λ v0 λ v1 λ v2 λ v3 .  v2
16:  λ v0 λ v1 λ v2 λ v3 .  let v0 = op$_{add-int/lit8}$(v0, 1) in
     (18) v0 v1 v2 v3
18:  λ v0 λ v1 λ v2 λ v3 .  (6) v0 v1 v2 v3
```

Light FuncDroid Evaluator result:

```
v0 -> 11

v1 -> 11

v2 -> "true"

v3 -> 1
```

6.2.3 Switch case (sparse-switch)

This test case demonstrates the use of switch case that results in the `sparse-switch` opcode. The cases are stored in the form of a list of target labels, which lead to the respective instructions carrying out the desired operations. The indexes of this list, for the target labels, are retrieved from the Dalvik executable file. The value of the register to test, `v0`, is 5 and used to find a match in the indexes. The results show that for the number 5, indeed the value "five" (in `v1`) is returned, as expected. The program also returns "five" as its result, as can be seen below.



```
04-02 05:17:53.395 7135-7135/com.example.harmony.sparseswitch I/Result: The result is "five".
```

Program in Java:

```
int number = 5;
String res = "";
switch(number){
   case 10: res = "ten";
      break;
   case 1: res = "one";
      break;
   case 50: res = "fifty";
      break;
   case 90: res = "ninety";
      break;
   case 5: res = "five";
      break;
}
return res;
```

Dalvik code:

```
0000:  const/4 v0, #int 5
0001:  const-string/jumbo v1, ""
0004:  sparse-switch v0, 0000001c
0007:  return-object v1
0008:  const-string/jumbo v1, "ten"
000b:  goto 0007
000c:  const-string/jumbo v1, "one"
000f:  goto 0007
0010:  const-string/jumbo v1, "fifty"
0013:  goto 0007
0014:  const-string/jumbo v1, "ninety"
0017:  goto 0007
0018:  const-string/jumbo v1, "five"
001b:  goto 0007
001c:  sparse-switch-data (22 units)
```

Light Android instructions:	*Light Android Evaluator result:*
`0:  const [v0] [5]`	`v0 -> 5`
`1:  const [v1] [""]`	`v1 -> "five"`
`4:  jmp [28] []`	
`7:  ret [] [v1]`	
`8:  const [v1] ["ten"]`	
`11: jmp [7] []`	
`12: const [v1] ["one"]`	
`15: jmp [7] []`	
`16: const [v1] ["fifty"]`	
`19: jmp [7] []`	
`20: const [v1] ["ninety"]`	
`23: jmp [7] []`	
`24: const [v1] ["five"]`	
`27: jmp [7] []`	
`28: jmp [12, 24, 8, 16, 20] [v0]`	

Light FuncDroid expressions:

```
0:   λ v0 λ v1 .  let v0 = 5 in (1) v0 v1
1:   λ v0 λ v1 .  let v1 = "" in (4) v0 v1
4:   λ v0 λ v1 .  (28) v0 v1
7:   λ v0 λ v1 .  v1
8:   λ v0 λ v1 .  let v1 = "ten" in (11) v0 v1
11:  λ v0 λ v1 .  (7) v0 v1
12:  λ v0 λ v1 .  let v1 = "one" in (15) v0 v1
15:  λ v0 λ v1 .  (7) v0 v1
16:  λ v0 λ v1 .  let v1 = "fifty" in (19) v0 v1
19:  λ v0 λ v1 .  (7) v0 v1
20:  λ v0 λ v1 .  let v1 = "ninety" in (23) v0 v1
23:  λ v0 λ v1 .  (7) v0 v1
24:  λ v0 λ v1 .  let v1 = "five" in (27) v0 v1
```

```
27:  λ v0 λ v1 .  (7) v0 v1
28:  λ v0 λ v1 .  if op_{value=index}(v0) then [(12), (24), (8), (16),
     (20)] v0 v1 else (7) v0 v1
```

Light FuncDroid Evaluator result:

```
v0 -> 5
v1 -> "five"
```

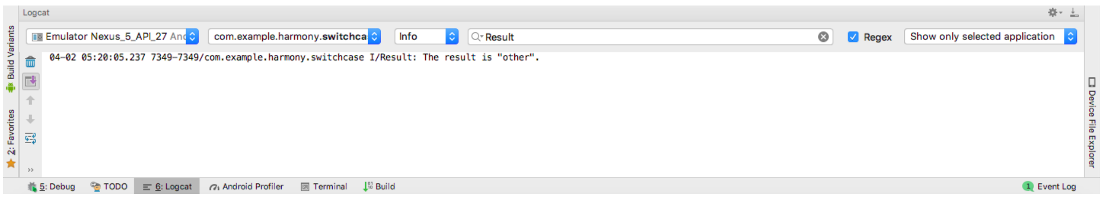
6.2.4 Switch case (packed-switch)

This test case demonstrates the use of switch case that results in the `packed-switch` opcode. This happens when the values are consecutive, as opposed to them being spread out like the previous test case (that resulted in the `sparse-switch` opcode).

The cases are stored in the form of a list of target labels, which lead to the respective instructions carrying out the specific operations. Only the first or lowest index of this list, for the target labels, can be retrieved from the Dalvik executable file. Therefore, we keep incrementing that index value by one for each consecutive target label in the list.

The value of the register to test, `v0`, is used to find a match in the indexes. If there is no match, the jump is made to the label after the switch instruction, which in our case is the label, `8`, that stores "other" in the resultant String.

The results of evaluation show that for the number 10, the value "other" is stored in `v1` as the return value, as we'd expect. The program also returns "other" as its result, as can be seen below.



```
04-02 05:20:05.237 7349-7349/com.example.harmony.switchcase I/Result: The result is "other".
```

Program in Java:

```
int number = 10;
String res = "";
switch(number){
   case 1: res = "one";
      break;
   case 2: res = "two";
      break;
   case 4: res = "four";
      break;
   case 5: res = "five";
      break;
   default: res = "other";
      break;
}
return res;
```

Dalvik code:

```
0000:  const/16 v0, #int 10
0002:  const-string/jumbo v1, ""
0005:  packed-switch v0, 0000001c
0008:  const-string/jumbo v1, "other"
000b:  return-object v1
000c:  const-string/jumbo v1, "one"
000f:  goto 000b
0010:  const-string/jumbo v1, "two"
0013:  goto 000b
0014:  const-string/jumbo v1, "four"
0017:  goto 000b
0018:  const-string/jumbo v1, "five"
001b:  goto 000b
001c:  packed-switch-data (14 units)
```

Light Android instructions:

```
0:   const [v0] [10]
2:   const [v1] [""]
5:   jmp [28] []
8:   const [v1] ["other"]
11:  ret [] [v1]
12:  const [v1] ["one"]
15:  jmp [11] []
16:  const [v1] ["two"]
19:  jmp [11] []
20:  const [v1] ["four"]
23:  jmp [11] []
24:  const [v1] ["five"]
27:  jmp [11] []
28:  jmp [12, 16, 8, 20, 24] [v0]
```

Light Android Evaluator result:

```
v0 -> 10
v1 -> "other"
```

Light FuncDroid expressions:

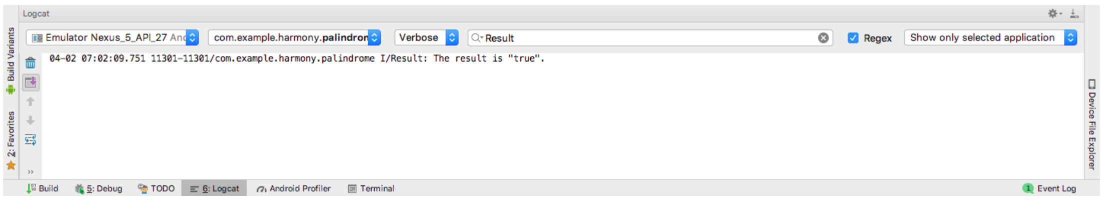
```
0:   λ v0 λ v1 .  let v0 = 10 in (2) v0 v1
2:   λ v0 λ v1 .  let v1 = "" in (5) v0 v1
5:   λ v0 λ v1 .  (28) v0 v1
8:   λ v0 λ v1 .  let v1 = "other" in (11) v0 v1
11:  λ v0 λ v1 .  v1
12:  λ v0 λ v1 .  let v1 = "one" in (15) v0 v1
15:  λ v0 λ v1 .  (11) v0 v1
16:  λ v0 λ v1 .  let v1 = "two" in (19) v0 v1
19:  λ v0 λ v1 .  (11) v0 v1
20:  λ v0 λ v1 .  let v1 = "four" in (23) v0 v1
23:  λ v0 λ v1 .  (11) v0 v1
24:  λ v0 λ v1 .  let v1 = "five" in (27) v0 v1
27:  λ v0 λ v1 .  (11) v0 v1
28:  λ v0 λ v1 .  if op_{value=index}(v0) then [(12), (16), (8), (20),
    (24)] v0 v1 else (8) v0 v1
```

Light FuncDroid Evaluator result:

```
v0 -> 10

v1 -> "other"
```

6.2.5 Palindrome

This test case checks if a given number is a palindrome or not. It takes a number and reverses its digits, and compares the reverse to the original number. If the two numbers are equal, it returns "true", and returns "false" otherwise. The resultant Light Android instructions and Light FuncDroid expressions both evaluate to "true", for the number 10401, as stored in `v4`. The program also returns "true" as its result, as can be seen below.


Emulator Nexus_5_API_27 And | com.example.harmony.palindron | Verbose | Result | Regex ✓ | Show only selected application

```
04-02 07:02:09.751 11301-11301/com.example.harmony.palindrome I/Result: The result is "true".
```

Program in Java:

```
int num = 10401;
int number = num;
int reverse = 0;
int  digit = 10000;

while(number>0){
   reverse +=
       (number%10)*digit;
   number /= 10;
   digit /= 10;
 }

if (num == reverse) return
    "true";
else return "false";
```

Dalvik code:

```
0000:  const/16 v1, #int 10401
0002:  move v2, v1
0003:  const/4 v3, #int 0
0004:  const/16 v0, #int 10000
0006:  if-lez v2, 0011
0008:  rem-int/lit8 v4, v2, #int 10
000a:  mul-int/2addr v4, v0
000b:  add-int/2addr v3, v4
000c:  div-int/lit8 v2, v2, #int 10
000e:  div-int/lit8 v0, v0, #int 10
0010:  goto 0006
0011:  if-ne v1, v3, 0017
0013:  const-string/jumbo v4, "true"
0016:  return-object v4
0017:  const-string/jumbo v4,
"false"
001a:  goto 0016
```

Light Android instructions:

```
0:   const [v1] [10401]
2:   mov [v2] [v1]
3:   const [v3] [0]
4:   const [v0] [10000]
6:   jmp_{if-lez} [17] [v2]
8:   op_{rem-int/lit8} [v4] [v2, 10]
10:  op_{mul-int/2addr} [v4] [v4, v0]
11:  op_{add-int/2addr} [v3] [v3, v4]
12:  op_{div-int/lit8} [v2] [v2, 10]
14:  op_{div-int/lit8} [v0] [v0, 10]
16:  jmp [6] []
17:  jmp_{if-ne} [23] [v1, v3]
19:  const [v4] ["true"]
22:  ret [] [v4]
23:  const [v4] ["false"]
26:  jmp [22] []
```

Light Android Evaluator result:

```
v0 -> 0
v1 -> 10401
v2 -> 0
v3 -> 10401
v4 -> "true"
```

Light FuncDroid expressions:

```
0:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v1 = 10401 in (2) v0 v1 v2 v3 v4
2:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v2 = v1 in (3) v0 v1 v2 v3 v4
3:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v3 = 0 in (4) v0 v1 v2 v3 v4
4:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v0 = 10000 in (6) v0 v1 v2 v3 v4
6:  λ v0 λ v1 λ v2 λ v3 λ v4 .  if op_{if-lez}(v2) then (17) v0 v1 v2 v3
    v4 else (8) v0 v1 v2 v3 v4
8:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v4 = op_{rem-int/lit8}(v2, 10) in (10)
    v0 v1 v2 v3 v4
10: λ v0 λ v1 λ v2 λ v3 λ v4 .  let v4 = op_{mul-int/2addr}(v4, v0) in (11)
    v0 v1 v2 v3 v4
11: λ v0 λ v1 λ v2 λ v3 λ v4 .  let v3 = op_{add-int/2addr}(v3, v4) in (12)
    v0 v1 v2 v3 v4
```

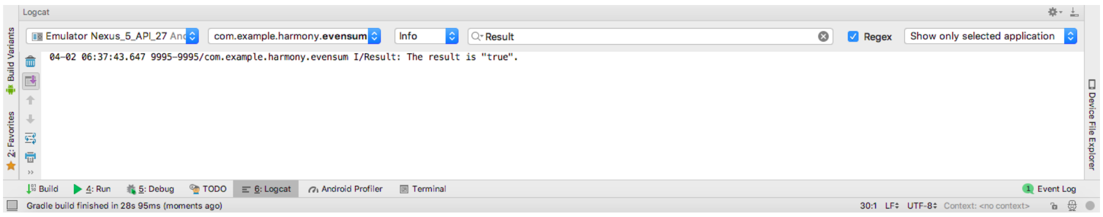
```
12:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v2 = op_div-int/lit8(v2, 10) in (14)
     v0 v1 v2 v3 v4
14:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v0 = op_div-int/lit8(v0, 10) in (16)
     v0 v1 v2 v3 v4
16:  λ v0 λ v1 λ v2 λ v3 λ v4 .  (6) v0 v1 v2 v3 v4
17:  λ v0 λ v1 λ v2 λ v3 λ v4 .  if op_if-ne(v1, v3) then (23) v0 v1 v2
     v3 v4 else (19) v0 v1 v2 v3 v4
19:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v4 = "true" in (22) v0 v1 v2 v3
     v4
22:  λ v0 λ v1 λ v2 λ v3 λ v4 .  v4
23:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v4 = "false" in (26) v0 v1 v2 v3
     v4
26:  λ v0 λ v1 λ v2 λ v3 λ v4 .  (22) v0 v1 v2 v3 v4
```

Light FuncDroid Evaluator result:

```
v0 -> 0

v1 -> 10401

v2 -> 0

v3 -> 10401

v4 -> "true"
```

6.2.6 Check if a sum is even

This test case mainly demonstrates the usage of a nested loop, and its evaluation. The program merely checks if the sum of two consecutive numbers, in the range 1 to 5, is even. It returns "true" if it is, and "false" otherwise. The results from the evaluators show that both the Light Android instructions and the Light FuncDroid expressions evaluate to "true", as stored in `v2`. The program also returns "true" as its result, as can be seen below.



```
04-02 06:37:43.647 9995-9995/com.example.harmony.evensum I/Result: The result is "true".
```

Gradle build finished in 28s 95ms (moments ago)

Program in Java:

```java
String result = "false";
for(int i=1;i<=5;i++){
   for(int j=i+1; j<=5; j++){
      if((i+j)%2==0){
         result = "true";
      }
   }
}
return result;
```

Dalvik code:

```
0000:  const/4 v4, #int 5
0001:  const-string/jumbo v2,
"false"
0004:  const/4 v0, #int 1
0005:  if-gt v0, v4, 001a
0007:  add-int/lit8 v1, v0, #int 1
0009:  if-gt v1, v4, 0014
000b:  add-int v3, v0, v1
000d:  rem-int/lit8 v3, v3, #int 2
000f:  if-nez v3, 0017
0011:  const-string/jumbo v2, "true"
0014:  add-int/lit8 v0, v0, #int 1
0016:  goto 0005
0017:  add-int/lit8 v1, v1, #int 1
0019:  goto 0009
001a:  return-object v2
```

Light Android instructions:

```
0:   const [v4] [5]
1:   const [v2] ["false"]
4:   const [v0] [1]
5:   jmp_if-gt [26] [v0, v4]
7:   op_add-int/lit8 [v1] [v0, 1]
9:   jmp_if-gt [20] [v1, v4]
11:  op_add-int [v3] [v0, v1]
13:  op_rem-int/lit8 [v3] [v3, 2]
15:  jmp_if-nez [23] [v3]
17:  const [v2] ["true"]
20:  op_add-int/lit8 [v0] [v0, 1]
22:  jmp [5] []
23:  op_add-int/lit8 [v1] [v1, 1]
25:  jmp [9] []
26:  ret [] [v2]
```

Light Android Evaluator result:

```
v0 -> 6
v1 -> 6
v2 -> "true"
v3 -> 1
v4 -> 5
```

Light FuncDroid expressions:

```
0:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v4 = 5 in (1) v0 v1 v2 v3 v4
1:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v2 = "false" in (4) v0 v1 v2 v3
    v4
4:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v0 = 1 in (5) v0 v1 v2 v3 v4
5:  λ v0 λ v1 λ v2 λ v3 λ v4 .  if op_if-gt(v0, v4) then (26) v0 v1 v2
    v3 v4 else (7) v0 v1 v2 v3 v4
7:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v1 = op v0 1 in (9) v0 v1 v2 v3
    v4
9:  λ v0 λ v1 λ v2 λ v3 λ v4 .  if op_if-gt(v1, v4) then (20) v0 v1 v2
    v3 v4 else (11) v0 v1 v2 v3 v4
11: λ v0 λ v1 λ v2 λ v3 λ v4 .  let v3 = op_add-int(v0, v1) in
    (13) v0 v1 v2 v3 v4
13: λ v0 λ v1 λ v2 λ v3 λ v4 .  let v3 = op_rem-int/lit8(v3, 2) in
```

```
     (15) v0 v1 v2 v3 v4
15:  λ v0 λ v1 λ v2 λ v3 λ v4 .  if op_{if-nez}(v3) then (23) v0 v1 v2 v3
     v4 else (17) v0 v1 v2 v3 v4
17:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v2 = "true" in (20) v0 v1 v2 v3
     v4
20:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v0 = op_{add-int/lit8}(v0, 1) in
     (22) v0 v1 v2 v3 v4
22:  λ v0 λ v1 λ v2 λ v3 λ v4 .  (5) v0 v1 v2 v3 v4
23:  λ v0 λ v1 λ v2 λ v3 λ v4 .  let v1 = op_{add-int/lit8}(v1, 1) in
     (25) v0 v1 v2 v3 v4
25:  λ v0 λ v1 λ v2 λ v3 λ v4 .  (9) v0 v1 v2 v3 v4
26:  λ v0 λ v1 λ v2 λ v3 λ v4 .  v2
```

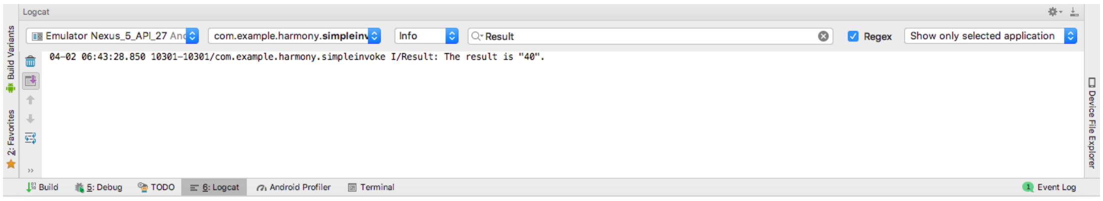
Light FuncDroid Evaluator result:

```
v0 -> 6

v1 -> 6

v2 -> "true"

v3 -> 1

v4 -> 5
```

6.2.7 Get the minimum of two numbers

This test case demonstrates a method invocation without passing any arguments. It shows the invoked method finding the minimum of two numbers, 40 and 50, and returning the result. The caller method then stores this result. Both the Light Android instructions and Light FuncDroid expressions show the value 40 being retrieved from the return value register, `v`, of the callee, and stored by the caller method in `v0`. The program also stores "40" as its result, as can be seen below.

We present the code in Java, Dalvik code, Light Android instructions, and Light FuncDroid expressions for both the caller and callee methods. Though the evaluation shows the register values of the caller method only, both methods have been evaluated to get the result.



```
04-02 06:43:28.850 10301-10301/com.example.harmony.simpleinvoke I/Result: The result is "40".
```

Program in Java:

```java
public static void
    invoke_method() {
   int result = get_min();
}


public static int get_min() {
   int a = 40;
   int b = 50;
   if(a < b) return a;
   else return b;
}
```

Dalvik code:

```
0000:  invoke-static {},
 Lcom/example/harmony/simpleinvoke/
 MainActivity;.get_min:()I
0003:  move-result v0
0004:  return-void


0000:  const/16 v0, #int 40
0002:  const/16 v1, #int 50
0004:  if-ge v0, v1, 0007
0006:  return v0
0007:  move v0, v1
0008:  goto 0006
```

Light Android instructions:

```
0:  inv [v]
[Lcom/example/harmony/
simpleinvoke/MainActivity;,
get_min, ()I]
3:  mov [v0] [v]
4:  ret [] []

0:  const [v0] [40]
2:  const [v1] [50]
4:  jmp_{if-ge} [7] [v0, v1]
6:  ret [] [v0]
7:  mov [v0] [v1]
8:  jmp [6] []
```

Light Android Evaluator result:

```
v0 -> 40
v  -> 40
```

Light FuncDroid expressions:

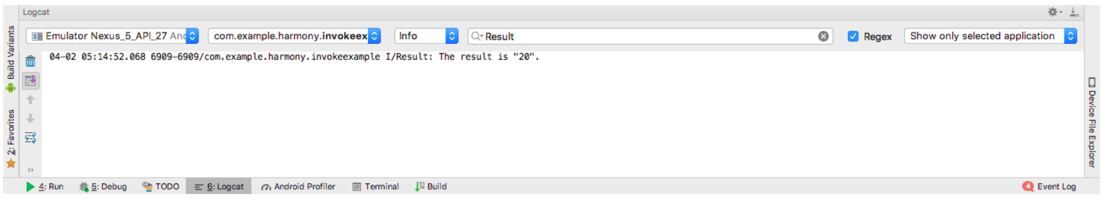
```
0:  λ v0 .  let v = (Lcom/example/harmony/simpleinvoke/MainActivity;
    .get_min:()I) in (3) v0
3:  λ v0 .  let v0 = v in (4) v0
4:  λ v0 .  unit

0:  λ v0 λ v1 .  let v0 = 40 in (2) v0 v1
2:  λ v0 λ v1 .  let v1 = 50 in (4) v0 v1
4:  λ v0 λ v1 .  if op_{if-ge}(v0, v1) then (7) v0 v1) else (6) v0 v1
6:  λ v0 λ v1 .  v0
7:  λ v0 λ v1 .  let v0 = v1 in (8) v0 v1
8:  λ v0 λ v1 .  (6) v0 v1
```

Light FuncDroid Evaluator result:

```
v0 -> 40

v  -> 40
```

6.2.8 Get the maximum of two numbers

This test case demonstrates a method invocation with arguments passed. It shows the caller method passing two numbers, 10 and 20. The callee method calculates the maximum of the two numbers and then returns the result, which is retrieved by the caller. Both the Light Android instructions and Light FuncDroid expressions show the value 20 being retrieved from the return value register, `v`, of the callee, and stored by the caller method in `v2`. The program also stores "20" as its result, as can be seen below.


Emulator Nexus_5_API_27 And · com.example.harmony.invokeex · Info · Result · Regex [✓] · Show only selected application

```
04-02 05:14:52.068 6909-6909/com.example.harmony.invokeexample I/Result: The result is "20".
```

We present the code in Java, Dalvik code, Light Android instructions, and Light FuncDroid expressions for both the caller and callee methods. Though the evaluators show the register values of the caller method only, both methods have been evaluated to get the result.

Program in Java:	*Dalvik code:*

`public static void`
`    invoke_example() {`
`   int a = 10;`
`   int b = 20;`
`   int result = get_max(a, b);`
`}`
 	
`public static int get_max(int a,`
`    int b) {`
`   if(a > b) return a;`
`   else return b;`
`}`
	`0000:  const/16 v0, #int 10`
	`0002:  const/16 v1, #int 20`
	`0004:  invoke-static {v0, v1},`
	`Lcom/example/harmony/invokeexample`
	`/MainActivity;.get_max:(II)I`
	`0007:  move-result v2`
	`0008:  return-void`
	
	`0000:  if-le v0, v1, 0003`
	`0002:  return v0`
	`0003:  move v0, v1`
	`0004:  goto 0002`

Light Android instructions:	*Light Android Evaluator result:*

`0: const [v0] [10]`
`2: const [v1] [20]`
`4: inv [v] [Lcom/example/harmony/`
`invokeexample/MainActivity;,`
`get_max,(II)I, v0, v1]`
`7: mov [v2] [v]`
`8: ret [] []`
 	
0: jmp$_{if\text{-}le}$ [3] [v0, v1]
`2: ret [] [v0]`
`3: mov [v0] [v1]`
`4: jmp [2] []`
	`v0 -> 10`
	`v1 -> 20`
	`v  -> 20`
	`v2 -> 20`

Light FuncDroid expressions:

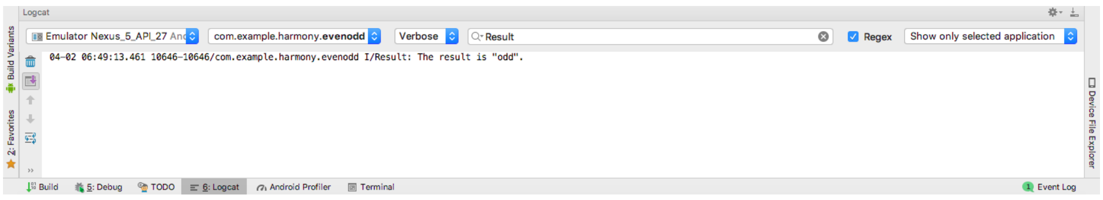
```
0:  λ v0 λ v1 λ v2 .  let v0 = 10 in (2) v0 v1 v2
2:  λ v0 λ v1 λ v2 .  let v1 = 20 in (4) v0 v1 v2
4:  λ v0 λ v1 λ v2 .  let v = (Lcom/example/harmony/invokeexample/
    MainActivity;.get_max:(II)I) v0 v1 in (7) v0 v1 v2
7:  λ v0 λ v1 λ v2 .  let v2 = v in (8) v0 v1 v2
8:  λ v0 λ v1 λ v2 .  unit

0:  λ v0 λ v1 .  if op_if-le(v0, v1) then (3) v0 v1 else (2) v0 v1
2:  λ v0 λ v1 .  v0
3:  λ v0 λ v1 .  let v0 = v1 in (4) v0 v1
4:  λ v0 λ v1 .  (2) v0 v1
```

Light FuncDroid Evaluator result:

```
v0 -> 10

v1 -> 20

v  -> 20

v2 -> 20
```

6.2.9 Even or odd

This test case checks if a given number is even or odd, with the help of nested conditionals in a while loop. It returns "even" if the number is even, and "odd" otherwise. Evaluating the Light Android instructions and Light FuncDroid expressions, both return "odd" in `v1`, for the number 19. The program also returns "odd" as its result, as can be seen below.



Emulator Nexus_5_API_27 And · com.example.harmony.evenodd · Verbose · Result · Regex [✓] · Show only selected application

```
04-02 06:49:13.461 10646-10646/com.example.harmony.evenodd I/Result: The result is "odd".
```

Program in Java:

```java
int number = 19;
String result = "";
while(number>=0){
   if(number==0) {
      result = "even";
      number = -1;
   }
   else {
      number -= 1;
      if(number==0) {
         result = "odd";
         number = -1;
      }
      number -= 1;
   }
}
return result;
```

Dalvik code:

```
0000:  const/16 v0, #int 19
0002:  const-string/jumbo v1, ""
0005:  if-ltz v0, 0019
0007:  if-nez v0, 000e
0009:  const-string/jumbo v1, "even"
000c:  const/4 v0, #int -1
000d:  goto 0005
000e:  add-int/lit8 v0, v0, #int -1
0010:  if-nez v0, 0016
0012:  const-string/jumbo v1, "odd"
0015:  const/4 v0, #int -1
0016:  add-int/lit8 v0, v0, #int -1
0018:  goto 0005
0019:  return-object v1
```

Light Android instructions:

```
0:   const [v0] [19]
2:   const [v1] [""]
5:   jmp_{if-ltz} [25] [v0]
7:   jmp_{if-nez} [14] [v0]
9:   const [v1] ["even"]
12:  const [v0] [-1]
13:  jmp [5] []
14:  op_{add-int/lit8} [v0] [v0, -1]
16:  jmp_{if-nez} [22] [v0]
18:  const [v1] ["odd"]
21:  const [v0] [-1]
22:  op_{add-int/lit8} [v0] [v0, -1]
24:  jmp [5] []
25:  ret [] [v1]
```

Light Android Evaluator result:

```
v0 -> -2
v1 -> "odd"
```

Light FuncDroid expressions:

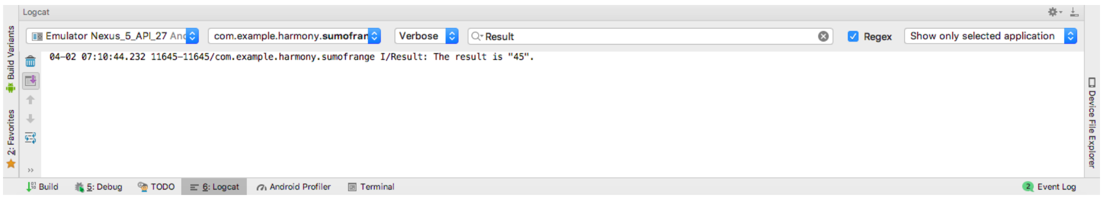
```
0:   λ v0 λ v1 .  let v0 = 19 in (2) v0 v1
2:   λ v0 λ v1 .  let v1 = "" in (5) v0 v1
5:   λ v0 λ v1 .  if op_{if-ltz}(v0) then (25) v0 v1 else (7) v0 v1
7:   λ v0 λ v1 .  if op_{if-nez}(v0) then (14) v0 v1 else (9) v0 v1
9:   λ v0 λ v1 .  let v1 = "even" in (12) v0 v1
12:  λ v0 λ v1 .  let v0 = -1 in (13) v0 v1
13:  λ v0 λ v1 .  (5) v0 v1
14:  λ v0 λ v1 .  let v0 = op_{add-int/lit8}(v0, -1) in (16) v0 v1
16:  λ v0 λ v1 .  if op_{if-nez}(v0) then (22) v0 v1 else (18) v0 v1
18:  λ v0 λ v1 .  let v1 = "odd" in (21) v0 v1
21:  λ v0 λ v1 .  let v0 = -1 in (22) v0 v1
22:  λ v0 λ v1 .  let v0 = op_{add-int/lit8}(v0, -1) in (24) v0 v1
24:  λ v0 λ v1 .  (5) v0 v1
25:  λ v0 λ v1 .  v1
```

Light FuncDroid Evaluator result:

```
v0 -> -2
v1 -> "odd"
```

6.2.10 Sum of numbers in a range

This test case calculates the sum of numbers from 1 to a given number. However, the numbers must be a multiple of 3 or 5. For the number 13, the sum is 45 (3+5+6+9+10+12). The evaluator results demonstrate that for the number 13, the sum stored in `v2` is indeed 45. The program also returns "odd" as its result, as can be seen below.



```
04-02 07:10:44.232 11645-11645/com.example.harmony.sumofrange I/Result: The result is "45".
```

Program in Java:

```
int sum = 0;
int number = 13;

for(int i=1; i<=number; i++){
   if(i%3==0 || i%5==0)
      sum += i;
}

return sum;
```

Dalvik code:

```
0000:  const/4 v2, #int 0
0001:  const/16 v1, #int 13
0003:  const/4 v0, #int 1
0004:  if-gt v0, v1, 0012
0006:  rem-int/lit8 v3, v0, #int 3
0008:  if-eqz v3, 000e
000a:  rem-int/lit8 v3, v0, #int 5
000c:  if-nez v3, 000f
000e:  add-int/2addr v2, v0
000f:  add-int/lit8 v0, v0, #int 1
0011:  goto 0004
0012:  return v2
```

Light Android instructions:

```
0:   const [v2] [0]
1:   const [v1] [13]
3:   const [v0] [1]
4:   jmp$_{if\text{-}gt}$ [18] [v0, v1]
6:   op$_{rem\text{-}int/lit8}$ [v3] [v0, 3]
8:   jmp$_{if\text{-}eqz}$ [14] [v3]
10:  op$_{rem\text{-}int/lit8}$ [v3] [v0, 5]
12:  jmp$_{if\text{-}neqz}$ [15] [v3]
14:  op$_{add\text{-}int/2addr}$ [v2] [v2, v0]
15:  op$_{add\text{-}int/lit8}$ ADD2L [v0] [v0, 1]
17:  jmp [4] []
18:  ret [] [v2]
```

Light Android Evaluator result:

```
v0 -> 14
v1 -> 13
v2 -> 45
v3 -> 3
```

Light FuncDroid expressions:

```
0:   λ v0 λ v1 λ v2 λ v3 .  let v2 = 0 in (1) v0 v1 v2 v3
1:   λ v0 λ v1 λ v2 λ v3 .  let v1 = 13 in (3) v0 v1 v2 v3
3:   λ v0 λ v1 λ v2 λ v3 .  let v0 = 1 in (4) v0 v1 v2 v3
4:   λ v0 λ v1 λ v2 λ v3 .  if op$_{if\text{-}gt}$(v0, v1) then (18) v0 v1 v2 v3
     else (6) v0 v1 v2 v3
6:   λ v0 λ v1 λ v2 λ v3 .  let v3 = op$_{rem\text{-}int/lit8}$(v0, 3) in (8) v0 v1
     v2 v3
8:   λ v0 λ v1 λ v2 λ v3 .  if op$_{if\text{-}eqz}$(v3) then (14) v0 v1 v2 v3 else
     (10) v0 v1 v2 v3
10:  λ v0 λ v1 λ v2 λ v3 .  let v3 = op$_{rem\text{-}int/lit8}$(v0, 5) in (12) v0 v1
     v2 v3
12:  λ v0 λ v1 λ v2 λ v3 .  if op$_{if\text{-}neqz}$(v3) then (15) v0 v1 v2 v3 else
     (14) v0 v1 v2 v3
14:  λ v0 λ v1 λ v2 λ v3 .  let v2 = op$_{add\text{-}int/2addr}$(v2, v0) in (15) v0
     v1 v2 v3
15:  λ v0 λ v1 λ v2 λ v3 .  let v0 = op$_{add\text{-}int/lit8}$(v0, 1) in (17) v0 v1
```

```
    v2 v3
17:  λ v0 λ v1 λ v2 λ v3 .  (4) v0 v1 v2 v3
18:  λ v0 λ v1 λ v2 λ v3 .  v2
```

Light FuncDroid Evaluator result:

```
v0 -> 14

v1 -> 13

v2 -> 45

v3 -> 3
```

Chapter 7

Results and Discussion

Light Android:

This project aimed to extract functions in the form of lambda expressions from Android apps. A preliminary step of this was to convert Dalvik code into an abstract analysis language called Light Android, which was part of existing work done. The methodical extraction of data from the Dalvik executable, accomplished in previous work, served as a springboard for our current work.

However, the initial purpose of the work done was to design an effect system for the behavioral analysis of Android apps, resulting in the design choice of abstracting operations performed. The evaluation of the light assembly code and functional expressions was not intended. Therefore, adjustments were made in the code to better align with our goals, which were easily incorporated, as the code was flexible to change.

The operational semantics for Light Android was obtained as a work in progress, which was a helpful starting point. On first glance, it seemed like a trivial job to use it, as it is, and explain the inference rules. However, on close inspection, some changes needed to be made for clarity and accuracy. Therefore, the semantics that we present are a refined version, with modifications. This invariably took a considerable amount of time to work out, as opposed to what we initially thought.

Delving into the details of Dalvik opcodes and understanding their behaviour was no mean feat, particularly those dealing with method invocations and switch cases. Although most of them were intuitive and simple to understand, it was a matter of tediousness due to its sheer number. In particular, how Dalvik handles the return value of an invoked method and returns it to a caller method was not made specific. Additionally, the format of switch cases' data payload had to be understood, and how they are evaluated by comparing a register's value to a set of table indexes.

The data payload is not extracted by pattern matching from a text file, like the rest of the instructions and app information. Rather, the bytecode had to be read form the Dalvik executable. One of the two variants, of the opcodes dealing with switch cases, only specified the first and lowest index. The procedure of evaluating instructions, and how the rest of the indexes are processed based on the first index, were unclear at first.

However, with synthesised examples, more reading, and progress made in the project, the details were made clearer. This in-depth understanding was also essential for grasping the intuition behind the translation of Dalvik opcodes to our light assembly code, before moving forward to the next implementation step. All things considered, there was significant work put into researching the pre-existing background, both in terms of acquiring the basics and understanding the existing work done.

Light FuncDroid:

The Light FuncDroid program was successfully developed to convert Light Android instructions into functional expressions. The program works without any errors, as far as we know, based on the test runs on sample programs. Indeed, multiple revisions were necessary, to fix bugs that arose from the testing, and also for general improvement of the program structure.

The bulk of the implementation done in this project was undoubtedly developing the Light FuncDroid program. The formal design, including its operational semantics, and the implementation were developed from scratch. Furthermore, having no previous knowledge of lambda calculus, as well as minimal experience in functional programming, meant additional time spent in learning the fundamentals outlined in Section 2.1.

Evaluators:

Evaluators for both of the translation programs have been designed, based on an example-driven approach. The coverage of programs that can be evaluated is limited, and not exhaustive, as we present these evaluations for illustrative purposes only; to demonstrate similarity in results. However, we cover a considerably wide range of opcodes. Additionally, it also provides confidence in our implementation, suggesting correspondence to the original program.

General opcodes that define allocation of (String or Integer) values and return statements are fundamental to the evaluation. The evaluators are able to handle more than 20 kinds of binary and unary operations, and all 12 kinds of relational comparisons, only excluding operations such as shift-right, shift-left and bitwise operations (AND, OR, XOR). Additionally, the coverage also extends to method invocations and switch cases, both of which were initially rather tricky to evaluate, as alluded to earlier.

We also presented 10 simple test cases, showing the full cycle of an Android app written in Java being compiled to Dalvik code, being converted to light assembly code or Light Android instructions, and finally to functional expressions. The equivalence of evaluation results is evident, due to its basic nature and as compared to the original program's results. They appear to be trivial due to their simplicity, however we attempted to cover varied operations and program structures, as described in Section 6.2.

Extensions:

There definitely a few extensions that can be made to the work done in this project, mainly to include more cases or specific instances.

First off, we have omitted the explanation of instructions dealing with arrays. The operational semantics for Light Android and Light FuncDroid can be extended to deal with the corresponding instructions. Secondly, the coverage of evaluators can indeed be expanded to deal with more types of values and also handle object registers, to operate on fields. This will allow us to have test cases across a broader range.

Chapter 8

Future Work

As mentioned in the beginning, this project is part of a broader research for a mobile security analysis framework, for the behavioural analysis of Android apps. The functional representation that has been developed will be used as input to an automated verification tool. A type system will be used to check the functional expressions against security properties, e.g. whether an uncommon or malicious call-sequence appears.

In this chapter, we briefly explore some of these concepts, which are in the research phase only and have not yet been implemented.

We introduce the typing rules used to capture unwanted behaviours in apps. A type and effect system is proposed, whereby the side-effect of evaluating a term is captured. A simple abstracted scenario is considered where we model an unwanted behavioural pattern as follows:

- `f` and `g` are functions we are interested in.
- `N` represents the set of functions we are interested in collecting, containing `f` and `g`.
- The main specification is that `f` must not be executed before `g`.

The corresponding typing rules are presented below:

$$\frac{}{\vdash x \ \& \ \{[\varepsilon]\}} \ \text{(Variable)}$$

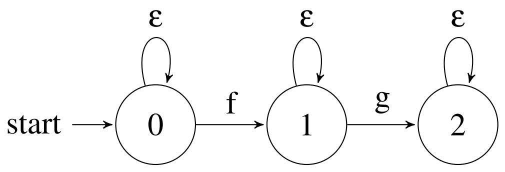
$$\frac{f \ \varepsilon \ N}{\vdash f \ \& \ \{[f]\}} \ \text{(Function)}$$

$$\frac{\vdash t_1 \ \& \ U_1 \quad \vdash t_2 \ \& \ U_2}{\vdash t_1 \ t_2 \ \& \ U_2 \ U_1} \ \text{(Application)}$$

$$\frac{\vdash t \ \& \ u}{\vdash \lambda x. \ t \ \& \ U} \ \text{(Abstraction)}$$

$$\frac{\vdash t_1 \ \& \ U_1 \quad \vdash t_2 \ \& \ U_2}{\vdash cond \ (?, \ t_1, \ t_2) \ \& \ U_1 \ \cup \ U_2} \ \text{(Conditional)}$$

The first rule specifies that when a variable is encountered, nothing is collected – represented by the empty word, ε. The second rule with regards to a function checks if the function is in the set of functions that we are interested in. If it is present in the set, then we collect it. The next few rules collect the effects in U.



In the form of a behaviour automaton, our scenario can be represented as above, for illustration purposes. The alphabet of the language captured by the automaton is $\Sigma = \{f, g, \varepsilon\}$. Analysing each input word in Σ gives us the following sets, which represent the set of states a particular word can reach (as can be seen from the automaton):

$$\begin{array}{rl} \varepsilon: & 0 \rightarrow \{0\} \\ & 1 \rightarrow \{1\} \\ & 2 \rightarrow \{2\} \end{array} \qquad \begin{array}{rl} f: & 0 \rightarrow \{0,1\} \\ & 1 \rightarrow \{1\} \\ & 2 \rightarrow \{2\} \end{array} \qquad \begin{array}{rl} g: & 0 \rightarrow \{0\} \\ & 1 \rightarrow \{1,2\} \\ & 2 \rightarrow \{2\} \end{array}$$

$$\begin{array}{rl} f.f: & 0 \rightarrow \{0,1\} \\ & 1 \rightarrow \{1\} \\ & 2 \rightarrow \{2\} \end{array} \qquad \begin{array}{rl} f.g^{+}: & 0 \rightarrow \{0,1,2\} \\ & 1 \rightarrow \{1,2\} \\ & 2 \rightarrow \{2\} \end{array} \qquad \begin{array}{rl} g.g: & 0 \rightarrow \{0\} \\ & 1 \rightarrow \{1,2\} \\ & 2 \rightarrow \{2\} \end{array}$$

$$\begin{array}{rl} g.f: & 0 \rightarrow \{0,1\} \\ & 1 \rightarrow \{1,2\} \\ & 2 \rightarrow \{2\} \end{array}$$

From the proof of equivalence of sets, as presented in [25], we can conclude that ff

and f are in the same equivalence class, as well as gg and g, since their sets are equal. The classes can then be listed as:

$$\begin{aligned} [\varepsilon] &= \{\varepsilon\} & [fg] &= \Sigma^* f \Sigma^* g \Sigma^* \\ [f] &= f^+ & [gf] &= g^+ . f^+ \\ [g] &= g^+ \end{aligned}$$

Keeping the equivalence classes in mind, the following table can be constructed, taking the union of each class with every class, including itself. The results of this can be seen in Table 8.1.

	[ε]	[f]	[g]	[f g]	[g f]
[ε]	[ε]	[f]	[g]	[f g]	[g f]
[f]	[f]	[f]	[f g]	[f g]	[g f]
[g]	[g]	[g f]	[g]	[f g]	[g f]
[f g]	[f g]	[f g]	[f g]	[f g]	[f g]
[g f]	[g f]	[g f]	[f g]	[f g]	[f g]

Table 8.1: Equivalence classes.

Appendix A

Code Appendix

A.1 Light FuncDroid program

```
package lang

import scala.collection.mutable.Map
import scala.annotation.tailrec

  /** Expression.
  * Exp ::= Const (String) | Var (String) | Nam (String) | Cls (String) |
      SFld (Cls, Nam) | IFld (Var, Nam) | Typ (String)
  Fun (Cls, Nam, Typ) | ...
  */

  abstract class Exp
    case class Const (s:String) extends Exp {override def toString = s}
    case class Var (s:String) extends Exp {override def toString = s}
    case class Nam (s:String) extends Exp {override def toString = s}
    case class Cls (s:String) extends Exp {override def toString = s}
    case class SFld (cls:Cls, f:Nam) extends Exp {override def toString =
        cls + "." + f}
    case class IFld (v:Var, f:Nam) extends Exp {override def toString = v
        + "." + f}
    case class Typ (s:String) extends Exp {override def toString = s}
    case class Fun (cls:Cls, name:Nam, typ:Typ) extends Exp {override def
```

```
        toString = cls + "." + name + ":" + typ}
    case class Op (xs:List[Exp]) extends Exp {override def toString =
        "op" + xs.map(x => x.toString).foldLeft("")(_+ " " +_)}
    case class Lab (i:Int) extends Exp {override def toString =
        i.toString}
    case class Abs (xs:List[Exp], e:Exp) extends Exp {override def
        toString = xs.map(x => x.toString).foldLeft("")(_+ " \\ " +_) + "
        . " + e}
    case class App (ea:Exp, xs:List[Exp]) extends Exp {override def
        toString = "(" + ea + ")" + xs.map(x =>
        x.toString).foldLeft("")(_+ " " +_)}
    case class Let (ea:Exp, eb:Exp, ec:Exp) extends Exp {override def
        toString = "let " + ea + " = " + eb + " in " + ec}
    case class Cond (c:Exp, es:List[Exp]) extends Exp {override def
        toString = "cond(" + c + es.map(x => x.toString).foldLeft("")(_+
        ", " +_) + ")"}
    case class Unit () extends Exp {override def toString = "unit"}
    case class Star () extends Exp {override def toString = "*"}
    case class Fix (v:Var, e:Exp) extends Exp {override def toString =
        "fix(" + v + ", " + e + ")"}

class LightFuncDroid(_la:LightAndroid) {

  def la:LightAndroid = _la

  type Label = String
  type Index = Int
  type Operation = String

  type Class = String
  type Name = String
  type MtdType = String

  def invoke (cls: Class, name: Name, typ: MtdType) {
     val xs =
      la.method.regs(cls, name, typ) match {
        case Some(lst) => lst.map(x => Var(x)) //adding all registers to
```

```
          a list
        case None => List()
      }

    val bd = la.method.body(cls, name, typ)

    bd match {
     case Some(lst) =>
        inst.initialise // initialise index, for each method
        lst.reverse.map(x => inst.serialise(cls, name, typ, x._1, x._2))
           // store each ins with an index, in order
        lst.reverse.map(x => translate.ins2exp(cls, name, typ, x._1,
           x._2, xs)) // translate ins to lambda exp
     case None =>
    }
  }

  def get_all(cls: Class, name: Name, typ: MtdType){
    val size = inst.get_size(cls, name, typ)
    for(i <- 0 to size - 1) {
      var label = inst.get_label(cls, name, typ, i)
      println(i + " " + label + " " + translate.get_operation(cls, name,
          typ, label) +" " + translate.get_exp(cls, name, typ, label))
    }
  }

  /**
   * Store the instructions in a method's body.
   */
  object inst {
    private val tb = Map[(Class, Name, MtdType, Index), (Label, Ins)]()
    private val labels = Map[(Class, Name, MtdType, Label), Index]()
    private val size = Map[(Class, Name, MtdType), Int]()

    var index: Index = -1

    // reset index
```

```
def initialise {
  index = -1
}

// storing each instruction with an index
def serialise(cls: Class, name: Name, typ: MtdType, l: Label, ins:
    Ins) {
  index += 1

  val key = (cls, name, typ, index)
  tb += (key -> (l, ins)) // store each label and instruction pair,
      with an index

  val l_key = (cls, name, typ, l)
  labels += (l_key -> index) // store the index corresponding to a
      label

  val s_key = (cls, name, typ)
  if (size contains s_key) size(s_key) += 1 // increment counter for
      no. of instructions
  else size += (s_key -> 1)
}

// get a particular instruction
def get_ins(cls: Class, name: Name, typ: MtdType, i: Index): Ins = {
  val key = (cls, name, typ, i)
  if (tb contains key) tb(key)._2
  else new Ins("", "", List(), List())
}

// get the label of an ins, at a given index
def get_label(cls: Class, name: Name, typ: MtdType, i: Index): Label
    = {
  val key = (cls, name, typ, i)
  if (tb contains key) tb(key)._1
  else "Not found"
}
```

```
  // get the index of an ins, with a given label
  def get_index(cls: Class, name: Name, typ: MtdType, l: Label): Index
      = {
    val l_key = (cls, name, typ, l)
    if (labels contains l_key) labels(l_key)
    else -1
  }

  def get_all_labels: Map [(Class, Name, MtdType, Label), Index] =
      labels

  // get all instructions
  def get_all: Map[(Class, Name, MtdType, Index), (Label, Ins)] = tb

  // get number of total instructions
  def get_size(cls: Class, name: Name, typ: MtdType): Int = {
    val s_key = (cls, name, typ)
    if (size contains s_key) size(s_key)
    else -1
  }
} // object inst

/**
 * Implements the translation procedure.
 */
object translate {

   private val expr = Map[(Class, Name, MtdType, Label), (Operation,
       Exp)]()

   def get_all: Map[(Class, Name, MtdType, Label), (Operation, Exp)] =
       expr

   def get_operation(cls: Class, name: Name, typ: MtdType, l: Label):
       Operation = {
    val key = (cls, name, typ, l)
```

```
     if (expr contains key) expr(key)._1
    else ""
   }

   def get_exp(cls: Class, name: Name, typ: MtdType, l: Label): Exp = {
    val key = (cls, name, typ, l)
     if (expr contains key) expr(key)._2
    else Unit()
 }

  /** Converts instructions to enriched Lambda expressions, returns a
      tuple containing a label and an Exp
  * Takes an expression as a parameter.
  */
  def ins2exp(cls: Class, name: Name, typ: MtdType, l:Label, ins:Ins,
      xs:List[Exp]) {

    var next_index = -1
    var next_label = -1

    val size = inst.get_size(cls, name, typ)
    val current_index = inst.get_index(cls, name, typ, l)

    if (current_index < size - 1) { // check if the next index/label
        exists, exclude last instruction
       next_index = current_index + 1
       next_label = inst.get_label(cls, name, typ, next_index).toInt
    }

    ins.op match {
      case "jmp" => if (ins.src.length == 0) { // source list is empty,
          unconditional jump
                    val exp = Abs(xs, Cond(Star(), ins.ta.map(x =>
                        App(Lab(x.toInt), xs))))
                    expr += ((cls, name, typ, l) -> (ins.h, exp))
                    //apply each target label to xs
                  }
```

```
                else if (ins.ta.length == 1) { // conditional jump,
                    only one target label
                  val exp = Abs(xs, Cond(Op(ins.src.map(x => Var(x))),
                                      App(Lab(next_label), xs) ::
                                          ins.ta.map(x =>
                                          App(Lab(x.toInt), xs))))
                  expr += ((cls, name, typ, l) -> (ins.h, exp))
                }

                else { // switches, more than one target label

                  // get label of the original switch instruction
                      (packed-switch/sparse-switch)
                  val orglabel = la.switch.orglabel(cls, name, typ, l)

                  // get the next index and label
                  next_index = inst.get_index(cls, name, typ,
                      orglabel) + 1
                  next_label = inst.get_label(cls, name, typ,
                      next_index).toInt

                  val exp = Abs(xs, Cond(Op(ins.src.map(x => Var(x))),
                                      App(Lab(next_label), xs) ::
                                          ins.ta.map(x =>
                                          App(Lab(x.toInt), xs))))
                  expr += ((cls, name, typ, l) -> (ins.h, exp))
                }

  case "op" => if (ins.ta.isEmpty && ins.src.isEmpty) {// no
      operands
                 val exp = Abs(xs, App(Lab(next_label), xs))
                 expr += ((cls, name, typ, l) -> (ins.h, exp))
               }

               else { // src.length >=1 and assuming the target list
```

```
                        contains only one variable
                      val exp = Abs(xs, Let(Var(ins.ta.head),
                                            Op(ins.src.map(x => Var(x))),
                                            App(Lab(next_label), xs)))
                      expr += ((cls, name, typ, l) -> (ins.h, exp))
                    }

        case "ret" => if (ins.src.isEmpty) { // returns unit (nothing)
                       val exp = Abs(xs, Unit())
                       expr += ((cls, name, typ, l) -> (ins.h, exp))
                     }

                     else { // assuming the source list contains only one
                         variable
                       val exp = Abs(xs, Var(ins.src.head))
                       expr += ((cls, name, typ, l) -> (ins.h, exp))
                     }

                     // ins.ta is always empty with "ret" commands

        case "inv" => val s = ins.src // ta = [def_reg = "v"], src =
            [cls, name, typ] ++ args
                     val t = s.tail.tail.tail //getting the args from the
                         src

                     val exp = Abs(xs, Let(Var(ins.ta.head),
                                           App(Fun(Cls(s(0)), Nam(s(1)),
                                               Typ(s(2))), t.map(x =>
                                               Var(x))), // C.m:T args
                                           App(Lab(next_label), xs)))
                     expr += ((cls, name, typ, l) -> (ins.h, exp))

        case "mov" => val exp = Abs(xs, Let(Var(ins.ta.head), // assuming
            ta.length & src.length == 1
                                           Var(ins.src.head),
                                           App(Lab(next_label), xs)))
                     expr += ((cls, name, typ, l) -> (ins.h, exp))
```

```
case "const" => val exp = Abs(xs, Let(Var(ins.ta.head),
                                Const(ins.src.head),
                                App(Lab(next_label), xs)))
                expr += ((cls, name, typ, l) -> (ins.h, exp))


case "iget" => val exp = Abs(xs, Let(Var(ins.ta.head),
                                IFld(Var(ins.src(0)),
                                   Nam(ins.src(1))),
                                App(Lab(next_label), xs)))
              expr += ((cls, name, typ, l) -> (ins.h, exp))

case "iput" => val exp = Abs(xs, Let(IFld(Var(ins.ta(0)),
    Nam(ins.ta(1))),
                                Var(ins.src.head),
                                App(Lab(next_label), xs)))
              expr += ((cls, name, typ, l) -> (ins.h, exp))

case "aget" => val exp = Abs(xs, Let(Var(ins.ta.head),
                                IFld(Var(ins.src(0)),
                                   Nam(ins.src(1))),
                                App(Lab(next_label), xs)))
              expr += ((cls, name, typ, l) -> (ins.h, exp))

case "aput" => val exp = Abs(xs, Let(IFld(Var(ins.ta(0)),
    Nam(ins.ta(1))),
                                Var(ins.src.head),
                                App(Lab(next_label), xs)))
              expr += ((cls, name, typ, l) -> (ins.h, exp))

case "sget" => val exp = Abs(xs, Let(Var(ins.ta.head),
                                SFld(Cls(ins.src(0)),
                                   Nam(ins.src(1))),
                                App(Lab(next_label), xs)))
              expr += ((cls, name, typ, l) -> (ins.h, exp))
```

```
        case "sput" => val exp = Abs(xs, Let(SFld(Cls(ins.ta(0)),
            Nam(ins.ta(1))),
                                        Var(ins.src.head),
                                        App(Lab(next_label), xs)))
                      expr += ((cls, name, typ, l) -> (ins.h, exp))

        case "new" => if (ins.src.length == 1) { // new t C or new t n
                        val exp = Abs(xs, Let(Var(ins.ta.head),
                                          Unit(),
                                          App(Lab(next_label), xs)))
                        expr += ((cls, name, typ, l) -> (ins.h, exp))
                      }

                     else { // new t #ns ns
                       val t = ins.ta.head
                       val ns = ins.src.tail // all src list elements
                           except the first
                       val fin:Exp = App(Lab(next_label), xs) // last
                           finishing expression
                       val f = (x:String, y:Exp) => Let(IFld(Var(t),
                           Nam(ns.indexOf(x).toString)), Const(x), y)
                       val exp = ns.foldRight(fin)(f)

                       expr += ((cls, name, typ, l) -> (ins.h, exp))
                     }

        case _ =>
      }
    } // ins2exp ends
  } // object translate ends
}
```

Bibliography

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[2] Android by Google Inc. Android Studio. https://developer.android.com/studio/index.html. Last accessed: 28-07-2017.

[3] Android by Google Inc. Application Fundamentals. https://developer.android.com/guide/components/fundamentals.html. Last accessed: 19-07-2017.

[4] Android by Google Inc. ART and Dalvik. https://source.android.com/devices/tech/dalvik/. Last accessed: 19-07-2017.

[5] Android by Google Inc. Dalvik Bytecode Format. https://source.android.com/devices/tech/dalvik/dalvik-bytecode. Last accessed: 14-08-2017.

[6] Android by Google Inc. Dalvik Executable format. https://source.android.com/devices/tech/dalvik/dex-format. Last accessed: 19-07-2017.

[7] Android by Google Inc. Dalvik Executable instruction formats. https://source.android.com/devices/tech/dalvik/instruction-formats. Last accessed: 14-08-2017.

[8] Android by Google Inc. Debug Your App Write and View Logs. https://developer.android.com/studio/debug/am-logcat.html. Last accessed: 01-04-2018.

[9] Android by Google Inc. The Android Open Source Project: Platform tools. https://android.googlesource.com/platform/dalvik.git/+/android-4.3_r3/dexdump/Android.mk. Last accessed: 29-07-2017.

[10] Android by Google Inc. The Android Source Code. https://source.android.com/source/. Last accessed: 16-07-2017.

[11] Android by Google Inc. Verifying App Behavior on the Android Runtime (ART). https://developer.android.com/guide/practices/verifying-apps-art.html. Last accessed: 19-07-2017.

[12] Wei Chen. Personal Communication.

[13] Wei Chen and David Aspinall. Resilient security analysis framework for mobile apps.

[14] Wei Chen, David Aspinall, Andrew D Gordon, Charles Sutton, and Igor Muttik. Learning and verifying unwanted behaviours.

[15] Wei Chen, David Aspinall, Andrew D Gordon, Charles Sutton, and Igor Muttik. More semantics more robust: Improving android malware classifiers. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 147–158. ACM, 2016.

[16] Wei Chen, David Aspinall, Andrew D Gordon, Charles Sutton, and Igor Muttik. On robust malware classifiers by verifying unwanted behaviours. In *International Conference on Integrated Formal Methods*, pages 326–341. Springer, 2016.

[17] Wei Chen, David Aspinall, Andrew D Gordon, Charles A Sutton, and Igor Muttik. Explaining unwanted behaviours in context.

[18] Wei Chen, David Aspinall, and Martin Hofmann. Behavioural analysis for Android apps.

[19] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[20] Alonzo Church. *The calculi of lambda-conversion*. Number 6. Princeton University Press, 1941.

[21] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

[22] Robert Grabowski, Martin Hofmann, and Keqin Li. Type-based enforcement of secure programming guidelines-code injection prevention at sap. *Formal Aspects in Security and Trust*, 7140:182–197, 2011.

[23] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[24] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851. ACM, 2013.

[25] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Automata theory, languages, and computation. *International Edition*, 24, 2006.

[26] Peter J Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[27] Peter J Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[28] Peter J Landin. A correspondence between ALGOL 60 and Church's Lambda-notations: Parts I and II. *Communications of the ACM*, 8(2,3):89–101, 158–167, Feburary and March 1965.

[29] Rick Linger, Kirk Sayre, Tim Daly, and Mark Pleszkoch. Function extraction technology: computing the behavior of malware. In *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, pages 1–9. IEEE, 2011.

[30] Abadi Martin and Cardelli Luca. A theory of objects. *Monographs in Computer Science, Springer-Verlag, New York, NY*, 1996.

[31] Robin Milner. The polyadic π-calculus: A tutorial, laboratory for foundations of computer science. *Computer Science Department, University of Edinburgh*, 1991.

[32] Magnus O Myreen and Michael JC Gordon. Function extraction. *Science of Computer Programming*, 77(4):505–517, 2012.

[33] The Statistics Portal. Number of apps available in leading app stores as of March 2017. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Last accessed: 20-07-2017.

[34] US Department of Homeland Security. Threats to Mobile Devices Using the Android Operating System. https://info.publicintelligence.net/DHS-FBI-AndroidThreats.pdf. Last accessed: 01-04-2017.

[35] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[36] Mark G Pleszkoch, Richard C Linger, and Alan R Hevner. Introducing function extraction into software testing. *ACM SIGMIS Database*, 39(3):41–50, 2008.

[37] Cornelia Pusch. Formalizing the java virtual machine in isabelle/hol. 1998.

[38] Claire L Quigley. Proof for optimization: Programming logic support for java jit compilers. In *Supplementary Proceedings of 2001 International Conference on Theorem Proving in Higher Order Logics, number EDI-INFRR-0046 in Informatics Research Report. Division of Informatics, University of Edinburgh*, 2001.

[39] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Reverse Engineering Android Apps With CodeInspect. In *IMPS@ ESSoS*, pages 1–8, 2016.

[40] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. How current android malware seeks to evade automated code analysis. In *IFIP International Conference on Information Security Theory and Practice*, pages 187–202. Springer, 2015.

[41] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[42] Milner Robin, Parrow Joachim, and Walker David. A calculus for mobile processes, parts 1 and 2. *Information and Computation*, 100(1–77), 1992.

[43] H. Singh. Informatics Research Proposal. April, 2017.

[44] Bruce Snell. Mobile threat report: What's on the horizon for 2016. *Intel Security and McAfee, published March*, 1, 2016.

[45] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(2):245–296, 1994.

[46] Carnegie Melon University. CERT Functional Extraction Project The CERT Division of the Software Engineering Institute (SEI). https://www.cert.org/historical/function-extraction.cfm. Accessed: 10-08-2017.

[47] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.