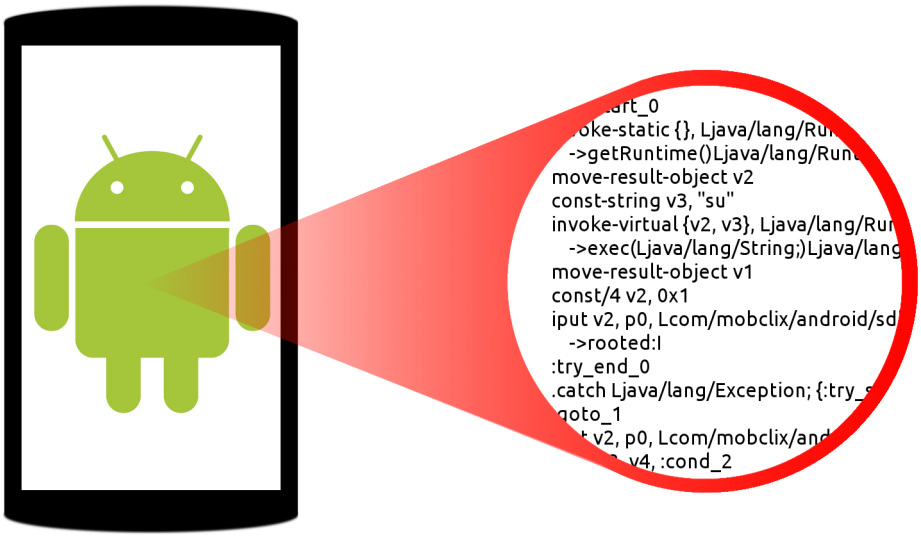
Study, Formalization, and Analysis of Dalvik Bytecode



Henrik Søndberg Karlsen and Erik Ramsgaard Wognsen
Software Engineering, Aalborg University
Student report, 9th semester, 2011

Title:
Study, Formalization, and Analysis of Dalvik Bytecode

Project period:
Software Engineering
SW9, Autumn 2011

Project group:
sw902e11

Authors:
Erik Ramsgaard Wognsen
Henrik Søndberg Karlsen

Supervisors:
René Rydhof Hansen
Mads Chr. Olesen

Abstract:

Android is the most popular smartphone operating system and several studies show an increasing problem with malicious third-party apps. Android apps run in the Dalvik virtual machine that is a register based VM for Java. We collect the 1,700 most popular free apps from Android Market and observe that many apps use all types of Dalvik instructions as well as multi-threading, reflection, and native ARM code. A thorough analysis for Android apps should consider all of these features.

We generalize the Dalvik instruction set and formalize it using operational semantics based on the Dalvik documentation, inspection of the Dalvik VM source code, and manual testing.

We define abstract domains for a control flow analysis based on the semantic rules. The analysis is a safe over-approximation of actual program behaviour defined by flow logic judgements for a simple control flow analysis that can form the basis of an analysis tool for Dalvik programs.

Number of printed copies: 5

Number of content pages: 68

Number of appendices: 3

Date of completion: January 5th 2012

Contents

Preface

The reader is assumed to have basic knowledge about the Java programming language, the Android platform, operational semantics, and program analysis.

We would like to thank René Rydhof Hansen and Mads Chr. Olesen for the project idea and for supervising the project.

Parts of the work done in this project has been submitted for *Bytecode 2012, the Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation* in collaboration with René Rydhof Hansen and Mads Chr. Olesen.

Chapter 1

Introduction

The Android operating system is gaining more users and is the most used operating system for smartphones [Nie11]. With the increasing number of users arise a demand for custom applications. This demand is met by Android developers through app markets, such as Android Market [Goo11a]. When the platform expands, so does the possibility of malicious apps which steal private information from the users [EGC+10], cost the users money by covertly sending overpriced text-messages [Ali11] or in other ways harm the user [FFC+11].

In this project, we focus on the increasing problems with malicious apps for Android. Through static program analysis it is possible to identify apps that misuse their access to personal information [EOMC11]. To identify instruction and API usage in typical Android apps, we study 1,700 of the most popular apps from Android Market. These observations are used in a formalization that is necessary to implement a static analysis for Android.

1.1 Program Analysis

A typical example of when information leakage on Android can occur is in an app that is allowed to access personal information, e.g. the contacts on the phone, and at the same time is allowed to access the Internet. Previous studies have "uncovered pervasive use/misuse of personal/phone identifiers" [EOMC11] and others have shown that 66 % of a set of 50 popular apps that send personal information through the internet connection do not rely on it to function [HHJ+11], signifying that it is a leakage to the advantage

of advertisers and other third parties rather than to the user. Dynamic and static program analysis can be used to detect such leakage [EGC+10].

Dynamic analysis requires the ability to run the application to be analyzed and, at runtime, track the information and how it is used. Furthermore, dynamic analysis requires a complete input domain for the application if the goal is to determine whether leakage is possible in any case and not just during average use. In [EGC+10] and [HHJ+11], they develop a dynamic analysis for Android, where personal information is tracked at runtime in a modified Android base to determine if and how it leaves the phone. These studies only focus on privacy issues and require a custom version of Android in order to run.

Static analysis can be run on the program source, binary executable or any intermediate step. It requires no execution of the application, but is able to determine conservative approximations of the flow of control or data within the application. Using static analysis, it is possible to track where in the application personal information could propagate to, and in turn answer whether or not an application might leak this personal information. Static analysis can also be used to find patterns of malicious behaviour or typical programming errors in applications by recognising known patterns in the analyzed code.

1.2 Operational Semantics and Flow Logic

To develop a formal analysis of Android apps, it is necessary to have a formal specification of its instructions. Operational semantics formally specify exactly what instructions do, and to our knowledge there is no known formal specification of the Android platform's bytecode, Dalvik bytecode. In this project, we therefore formalize the instructions using operational semantics. In addition, a control flow analysis is needed as a basis for any detailed analysis of information flow. A control flow analysis for a similar language, Carmel (a Java Card bytecode subset) has previously been developed using flow logic [Han05]. Due to the similarities between the languages, we specify the control flow analysis as a flow logic [NNH99].

Chapter 2

Android and Dalvik

Android is an operating system for mobile devices that includes various middleware and key applications [And11c]. It is based on the Linux kernel and allows third-party developers to create apps (applications) that are distributed on Android Market [Goo11a] among others for end-users to download. The middleware is typically written in C or C++ while user facing apps run isolated in an application sandbox. In the sandbox, the Dalvik Virtual Machine runs Dalvik bytecode which is usually compiled from Java. Apps can include native code for the ARM processor, typically written in C or C++, and this is also run inside the sandbox. An overview of these layers in the Android architecture is shown in Figure 2.1.

2.1 Dalvik

Android apps are run in the Dalvik Virtual Machine. It is similar to regular Java virtual machines but there are several differences between them [EOMC11]:

Register architecture Dalvik VM instructions are based on a register architecture, while regular Java VMs are stack-based. This means that there is no operand stack available for the instructions, and thus instructions instead have register arguments to indicate which data to work with. There are 2^{16} virtual registers available on Dalvik.

A single DEX file All classes in a Dalvik application is stored in a single file, while standard Java compilers produce a `class` file for each Java class. The DEX file is produced by the `dx` compiler which packages

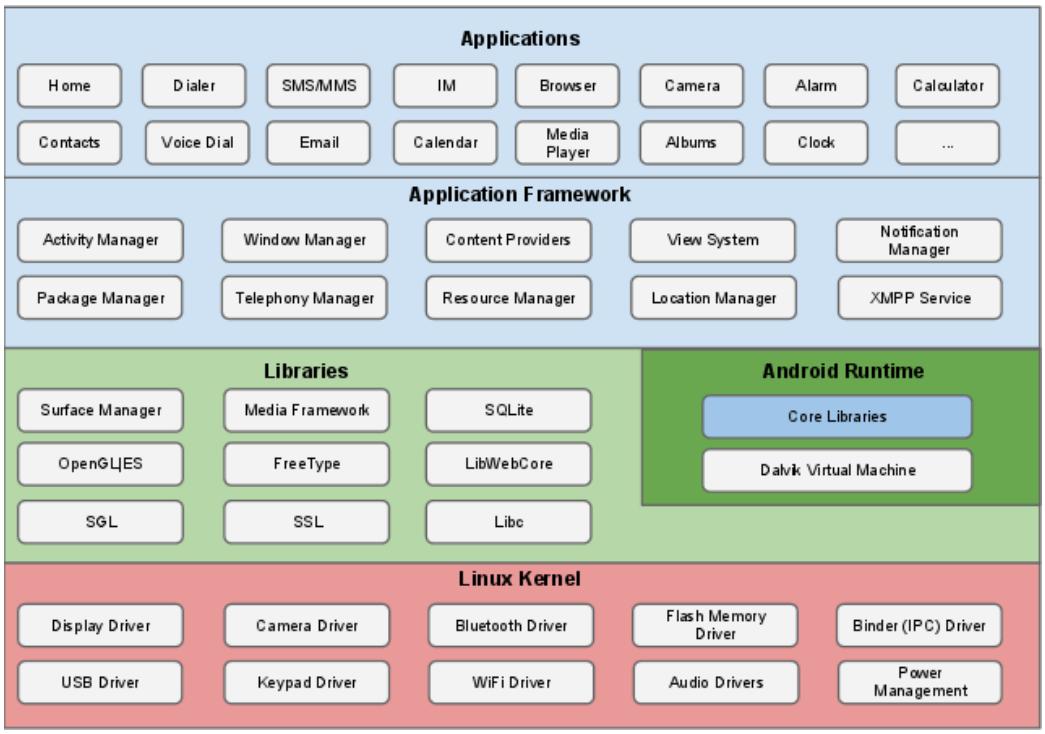


Figure 2.1: Android Architecture (from [And11d]).

`class` files into DEX and also performs some optimisations, such as inlining primitive constants.

Instruction set There are 200 opcodes in Java bytecode, and 218 in Dalvik bytecode, but the purposes of large parts of these sets are substantially different. A large part of the opcodes for Java bytecode is used to explicitly move data to and from the stack, while Dalvik has none of these. This results in fewer instructions used per Dalvik program, but as the instructions instead receive the destination and source registers as arguments, the instructions are in average longer than those for Java.

Ambiguous primitive types Instructions that access primitive types, such as `long` or `float`, specify only the width of the data type and not the actual type, which means that this part of the bytecode is ambiguous compared to regular Java VMs.

2.2 Android Permissions

By default, apps have very limited possibilities and are sandboxed from each other with separate Unix user IDs [And11d]. To read the user's private data, access the network or affect other apps or the operating system, an app must declare its needs to do so. This is done using permissions, for example `CALL_PHONE` or `READ_CONTACTS`. An app declares the permissions it needs statically in the Android Manifest which is an XML file [And11b]. They are presented at install time and if they are not accepted by the user, installation is aborted. Once an app is installed it cannot request more permissions, but is able to probe for permissions in order to check if a specific permission has been granted to the app. This is typically seen in reusable code in included programming libraries such as advertisement libraries. The permissions are enforced by Android using two methods [FCH+11]:

Unix groups Access to all files, including network sockets for Internet and Bluetooth, is enforced using Unix groups.

Validation mechanism in system process Inside the Dalvik VM each API that requires permissions implements an RPC interface, which is invoked by the system process to verify granted permissions.

The different enforcement methods mean that only the apps that run in the Dalvik VM are able to use regular API calls, hence it is not possible to make them from native code. However, the native code runs with the same Unix user ID as the rest of the app, thus making it possible to access Internet and Bluetooth sockets as well as regular files, e.g. files on an SD-card.

2.3 Application Package Files

Android apps are distributed and installed as APK (application package) files. The file format is a variant of JAR, used for Java, which is itself based on the Zip compression format. An APK file includes the following:

`META-INF` A directory that includes certificates and checksums for the application.

`res` A directory with resources, e.g. images and string values in XML files.

`lib` A directory with pre-compiled native libraries used by the application, usually ARM ELF shared object files.

`AndroidManifest.xml` An XML file that includes a static description of the app, e.g. name, version, required permissions and the required Android version. The Android manifest is not to be confused with the JAR Manifest which is included in the META_INF directory.

`classes.dex` A file with all the classes for the application, saved in the binary DEX format which runs on the Dalvik VM.

2.4 DEX Format

The DEX file format is different from `class` files for Java bytecode which means that standard Java bytecode analysis is not possible directly on a DEX file. In [EOMC11], they have analyzed Android apps based on a decompilation from DEX to Java, by using their own custom decompiler which is able to translate DEX to Java bytecode, which was then translated into Java using existing tools. This approach was chosen to leverage existing Java analysis tools, but presented problems in each of the translation steps, and this was our motivation to create an analysis directly for Dalvik bytecode.

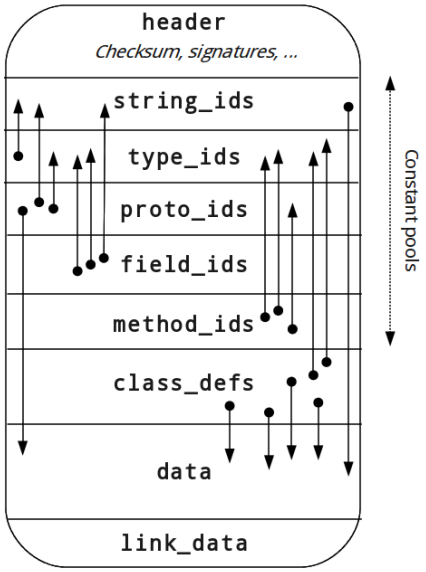


Figure 2.2: Layout of the DEX file format. The arrows indicate references between sections.

Figure 2.2 shows the overall layout of a DEX file.

header Data about the rest of the file, such as a checksum and sizes of the other sections.

string ids A list of string identifiers used throughout the file including type descriptors and constant strings referred to by the source code. The actual strings are stored in the `data` area, and thus an offset to where the string is kept is saved in this section.

type ids A list similar to the one for strings, except it only has identifier strings for types. Identifiers for all types referred to by the file must be placed here, and the items saved must be indices into the `string_id` section described above.

prototype ids Information about the prototypes (method signatures) used in the file. It consists of indices into the `string_id` section for a descriptor string of each prototype, an index into the `type_ids` for the return type and an offset into the `data` section where the details of the parameters for the prototype can be found.

fields ids Field items consisting of an index into the `type_ids` for the class where the field is defined, another index to the `type_ids` section for the type of the field and an index into `string_ids` for the name of the field.

methods ids Same as fields, except that an index into the `proto_ids` for the prototype of the method is used instead of the type index.

class defs Definitions of classes including an index into the `type_ids` section for each class type, access flags, an index into the `type_ids` for the superclass, offsets into the `data` section for interfaces that the class implements, static fields (if any) and the actual instructions in the method. In addition, it has indices to info about the source file and annotations.

data An area containing support data for the above listed sections.

link data Data used for statically linked files.

The id sections can be seen as constant pools, similar to the constant pool from `class` files for Java bytecode, except that these contain data for all the classes in the app and not just from a single class. The integrity of the file and its indices is verified by the Dalvik bytecode verifier before the app can run.

2.5 Smali and Apktool

For the study of Android apps in this project, we do not analyse the APK files directly. Instead, we use a tool to first extract the content of the APK file and then decompile the binary DEX file into a human readable format. `apktool` [Bru11] is an open-source utility that is able to decode resources and the DEX file from an APK and then run `smali` [jes11], another open-source tool, to disassemble the DEX file. This creates a file for each Java class in the app in a format similar to that of `Jasmin` [Jon11] which is an assembler for the Java VM. Below is an example of a method in `smali`:

```
1 .method public example(I)I
2     .locals 1
3     .parameter "a"
4     add-int/lit8 v0, p1, 0xa
5     .local v0, b:I
6     return v0
7 .end method
```

Line 1 is where the method definition begins, at which we can see that the method `example` takes one parameter of the type `int` (`I`), and returns an integer as well, represented as the last character of the signature. Line 2-3 tells us that the method uses one local register and that the parameter from before is called `a` in the original source code. Line 4 uses the instruction `add-int/lit8` which takes the destination register (`v0`), the source register (`p1`), and the integer literal `0xa`. The source register `p1` is an alias for the register `v2` which contains the first explicit parameter, `a` (register `v1` contains the object reference that the method is invoked on). Line 5 tells us that the variable in the register `v0` was named `b` and has the type `int`. Finally, the content of the register `v0` is returned and the method definition is ended.

The lines starting with a period are pseudo-instructions that represent the structure in the DEX file.

The corresponding Java code can be seen here:

```
1 public int example(int a)
2 {
3         int b = a + 10;
4         return b;
5 }
```

Chapter 3

Study of Apps

In order to identify what instructions are used in Android apps, we have collected 1,700 of the most popular Android apps from Android Market [Goo11a]. This data set is not only used to identify what types of instructions are used, but also to observe to what extent specialized Java features that could affect a static analysis, such as reflection, native libraries, and dynamic class loading, are used.

Other studies have had similarly sized data sets: [HHJ+11] had 1,100 apps and [FCH+11] had 940 apps. These publications do not describe how they retrieved the apps. In this chapter we first describe Android Market, how the apps were collected and what obstacles we encountered during the process. Then, we describe the results from our study of Android apps.

3.1 Android Market

Apps for Android devices are distributed through an online service called Android Market, where the apps are developed by professionals as well as amateurs. Publishing apps on the market requires a market account which can be bought for a small fee. The apps are categorised as either games or applications, which are sub-categorized in 8 and 26 categories, respectively. The market contains paid as well as free apps, with a total of nearly 320,000 different apps as of September 2011 [res11].

Apps from Android Market can only be downloaded using a Google Account on an Android device, and while browsing the market, settings from

this account are used to filter and sort the available apps. Furthermore, apps not supported by the hardware on the device are filtered out. We created a Google Account for this project and left all settings as default except for the language, which we explicitly changed from Danish to English. Once an Android device has been connected to an account, the account can be used to install the apps either through a web browser or through the standard Market app on the device. Each category for Android Market is by default sorted by popularity, where the sorting is based on the settings from the Google Account and the location of the device. The latter means that the most popular apps listed included several Danish apps.

3.2 App Retrieval

For this study, we downloaded the 50 most popular free apps from each of the application and game categories in November 2011. The apps are distributed as APK files, but these files can only be downloaded by the phone, or using custom software to circumvent the regular download process. We tried to use a crawler created to look like an Android device to access Android Market [Rau11] but we could not get it to work. Instead, we automated a browser on a PC using JavaScript to activate installation of the apps and created scripts to retrieve the APK file for each app from the phone after the app had been installed. By installing the apps on an actual Android device, we were also able to verify that they could in fact be installed since we experienced that the available apps were not always compatible with our device and could not be installed. The smartphone we used was a rooted Samsung Nexus S running Android 2.3. By default, Android devices are locked such that the user cannot act as the root user on the device. This has been circumvented using a custom bootloader and a root exploit to enable us to access all data on the phone and thereby retrieve all APK files.

3.2.1 Extracting the Files

The Android SDK includes a tool called `adb` (Android Debug Bridge) [Goo11b] which allows us to connect to a command line shell on the Android device through a USB cable. Android places the files in different locations on the phone, depending on the type of application:

/data/app Default location where regular free apps are placed.

/system/app Apps signed by Google or the phone manufacturer.

/mnt/asec/*app-pkg-name* An encrypted virtual filesystem for each app, if the app is installed on the external storage (e.g. SD-card).

/data/app-private Apps marked as "protected", and paid apps, are placed in this location which is only readable by the root user.

Using `adb` we were able to pull all the APK files from the phone to the PC. Apps cannot be placed in more than one category, except for the special categories Live Wallpapers and Widgets. We removed duplicates in our data set by removing the apps from these special categories, and replacing them with the next ones from the list of popular apps.

3.3 Generalization of Instructions

We used our data set to figure out which instructions have to be included in an analysis. Dalvik supports 218 opcodes which can be found at [And11e]. Some examples are:

- `move`
- `move-wide`
- `move-wide/from16`
- `move-object`
- `add-int`
- `add-float`
- `add-float/2addr`

Many of these are semantically similar. There are specific instruction variants for several of the primitive types, and for instructions working on types with specific widths. We have generalized them into a set of 39 instructions that each represent a group of semantically similar Dalvik instructions. The Dalvik bytecode verifier ensures that instructions are used with the proper types and widths, and therefore we can ignore these in our study. Some instructions are able to access only the first 16 or 256 registers, while we simulate that all instructions can access all registers. We use `binop` as a generalized instruction for all the instructions that are classified as binary

operations on registers, including `add-int` and `add-float`. The `2addr` variants let one of the source registers double as the destination register, so they are trivial to express as regular three-register instructions. Another instruction for binary operations, `binop-lit`, uses a literal value instead of a register for the second operand. Our `unop` instruction is a generalization of the instructions for unary operations. We have not generalized the instructions used for method invocation, as their behaviours are different from one another due to the nature of dynamic dispatch. The Dalvik `const` instructions are generalized into three instructions: `const` for primitive type values, and `const-string` and `const-class` for non-primitive type values.

The complete mapping from original Dalvik bytecode instructions to our generalized instructions can be found in Appendix A.

3.4 Usage of Instructions

We have counted the instructions used in our data set and found a total of 94,413,932 instructions. Table 3.1 shows the distribution of our generalized instructions and their usage in apps and total number of occurrences.

The results show that the instructions `invoke-direct` and `return-void` are used by all apps, and manual inspection has clarified that this is due to a constructor which is always present in the `R` class which specifies resources included in the app. The two most used instructions are `invoke-virtual` and `move-result` which are used to invoke a normal (virtual) method and to move its return value from a method to a register, respectively. From the results we can also see that while there are no generalized instructions that are not used, specialized array filling instructions (`fill-array-data` and `filled-new-array`) are among the most uncommon instructions, with the latter being used a total of just 1,930 times in our data set, but it is still present in more than 20 % of the apps. Monitors, the implementation of synchronized methods and blocks in Java, are used in 88 % of the apps, which indicates that developers use more than one thread per app. The use of `const-class` in 93 % of the apps indicate that a large amount of the apps use Java classes as objects or otherwise as parameters for methods, something which is usually seen in relation to the use of reflection, where classes are accessed and possibly changed dynamically (at runtime).

Instruction	Used by	Occurrences	Of total occ.
invoke-direct	100.00 %	4,533,934	4.80 %
return-void	100.00 %	2,683,104	2.84 %
invoke-virtual	99.59 %	12,718,970	13.47 %
const	99.53 %	8,157,468	8.64 %
move-result	99.47 %	12,391,920	13.13 %
invoke-super	99.47 %	215,434	0.23 %
const-string	99.29 %	5,200,603	5.51 %
new-instance	99.29 %	2,900,269	3.07 %
invoke-static	99.24 %	3,833,347	4.06 %
iput	99.12 %	3,389,122	3.59 %
iget	99.06 %	8,062,226	8.54 %
ifz	99.06 %	3,984,192	4.22 %
goto	98.76 %	3,263,902	3.46 %
return	98.06 %	2,166,727	2.29 %
move-exception	97.71 %	761,554	0.81 %
check-cast	97.53 %	1,055,790	1.12 %
if	97.24 %	1,304,228	1.38 %
binop-lit	96.59 %	1,232,732	1.31 %
invoke-interface	96.35 %	1,761,883	1.87 %
move	96.24 %	5,503,780	5.83 %
new-array	95.47 %	557,610	0.59 %
sget	95.18 %	1,792,583	1.90 %
aput	94.88 %	1,864,219	1.97 %
binop	94.53 %	1,218,279	1.29 %
aget	94.47 %	734,425	0.78 %
unop	94.00 %	530,779	0.56 %
sput	93.88 %	607,269	0.64 %
array-length	93.65 %	263,662	0.28 %
const-class	93.53 %	182,077	0.19 %
throw	93.47 %	521,299	0.55 %
packed-switch	93.35 %	86,468	0.09 %
nop	92.76 %	56,951	0.06 %
cmp	92.00 %	189,789	0.20 %
monitor-exit	88.76 %	287,310	0.30 %
monitor-enter	88.76 %	134,466	0.14 %
fill-array-data	86.71 %	97,906	0.10 %
instance-of	85.76 %	144,576	0.15 %
sparse-switch	69.71 %	21,149	0.02 %
filled-new-array	22.29 %	1,930	0.00 %
Total		94,413,932	100.00 %

Table 3.1: The generalized instructions in our data set ordered by the percentage of the 1,700 apps in our data set that use the instruction.

Feature	Used by apps	Hereof in libs
Obfuscated source	64.82 %	-
Has native libraries	20.35 %	-
`java/lang/Thread`	90.18 %	24.07 %
`java/lang/reflect`	73.00 %	55.92 %
`java/lang/ClassLoader`	39.71 %	81.19 %
`java/lang/Runtime;->exec`	19.53 %	80.44 %

Table 3.2: Percentages of apps in our data set that use various features.

3.5 Feature Usage

Table 3.2 shows some observations we have made from our data set, regarding the use of the following features:

Obfuscated source Code obfuscation is used to make reverse engineering of apps harder, and Google recommends [Goo11c] the use of ProGuard [Eri11] which renames classes and variables to short meaningless names (a, b, c etc.). We searched for the file `a.smali` within apps in our data set, and used this as an approximation to determine if an app is obfuscated. The same approach was used by [EOMC11] which found 36 % of apps to include any obfuscated source (some apps are partially obfuscated). We found the file in 64.82 % of the apps. Both numbers indicate if an app contains any obfuscated source, either in the source written by the developer or in a third-party library included.

Native libraries [EOMC11] have looked at the use of ARM shared object (`.so`) files and found that of their 1,100 studied apps from September 2010, 6.45 % included shared objects. We have found that this number has increased rapidly as 20.35 % of apps in our data set include shared objects. In addition, a small number of apps include ARM executables.

Threading We found use of monitors in 88 % of the apps, and wanted to know if the apps also explicitly use threads. We found that 90.18 % of the apps include a reference to `java/lang/Thread`.

Reflection With the high usage of `const-class` we wanted to know if it is common for Android Developers to use reflection in their apps. We found that 73 % of the apps include a reference to `java/lang/reflect` which is the basic library for reflection in Java.

Class Loading Dynamically loading classes can be done using a class loader, and potentially means that an app loads DEX files at runtime. We searched the apps for usage of `java/lang/ClassLoader` and found it present in 39.71 % of the apps.

`Runtime.exec()` The Java method `Runtime.exec()` is used to execute programs in a separate native process. This means that an app might run system commands, such as trying to read logs or gain root permissions. We searched for `java/lang/Runtime;->exec` and found it present in 19.53 % of the apps.

We look at two separate kinds of code that we call *developer code* and *library code*. We define developer code as code that lies within the natural packages for the application. For an application `net.company.app` extracted to the folder `net.company.app` this means all classes located directly in:

```
net.company.app/
net.company.app/net/
net.company.app/net/company/
net.company.app/net/company/app/
```

and any subdirectories in `net.company.app/net/company/app/`. We define library code as everything else in an app. The third column in Table 3.2 includes the same observations made for only the library code.

The number of classes for each Android app is rather large compared to what we expect for regular Java applications. Table 3.3 shows the average number of classes per app, along with the median, maximum and minimum number found in our data set. We also found the number of inner classes and inner anonymous classes for each app as Android apps usually include a lot of these, for example small anonymous classes for alert dialogs and event listeners. We classified inner classes to be any class with a `$` in their full class name, and anonymous inner classes to be any class with a `$` followed by a digit in their name, since this is how Java compilers usually name them. Obfuscated apps have fewer of these, if any at all, as obfuscation tools rename classes. These numbers should therefore be taken as minimum values (unless a developer has intentionally renamed Dalvik classes). We have included the same numbers for classes only found in developer code to give an insight into the number of classes actually written by the app developer for the specific app, and the number of classes typically found in libraries.

Android apps can declare different components in their Android manifest. The declared components indicate entry points for the application, which we discuss further in Section 4.8. Table 3.4 shows the distribution of declared

	Average	Median	Max	Min
Total classes	464.09	264	8335	4
Inner classes	186.76	99	2667	0
Anonymous inner classes	96.34	45	1304	0
Total dev. classes	182.55	100	2909	0
Inner dev. classes	91.78	37	1858	0
Anonymous inner dev. classes	56.73	16	976	0

Table 3.3: Statistics for the use of classes per app in our data set.

components in our data set. The numbers indicate that almost all the apps include at least one activity (a basic GUI element), that one activity is the most common thing to have, that services and broadcast receivers (global event handlers) are declared in about half of the apps and that 16.88 % of the apps declare a content provider (a manager for shared data, for example a database).

Component	Declares	Average	Median	Mode	Max
Activity	98.64 %	12.73	8	1	123
Service	48.82 %	1.02	0	0	23
Content Provider	16.88 %	0.27	0	0	12
Broadcast Receiver	50.88 %	1.38	1	0	43

Table 3.4: Percentage of apps in our data set that have declared at least one of the specific application components.

We discussed Android permissions and how they are enforced in Section 2.2. The use of permissions is interesting in relation to what information malicious apps might leak or misuse. Table 3.5 shows the 20 most declared permissions in our data set. The Internet permission is the most popular and is declared in 89.1 % of the apps, 49.4 % of the apps are able to write to external storage (SD-card) and 45.4 % of them are able to read private information such as the phone or IMEI number[1].

[1]The IMEI number is used to identify a physical mobile device

#	Permission	Decl. by
1	android.permission.INTERNET	89.1 %
2	android.permission.ACCESS_NETWORK_STATE	71.6 %
3	android.permission.WRITE_EXTERNAL_STORAGE	49.4 %
4	android.permission.READ_PHONE_STATE	45.4 %
5	android.permission.VIBRATE	27.8 %
6	android.permission.WAKE_LOCK	24.9 %
7	android.permission.ACCESS_COARSE_LOCATION	24.0 %
8	android.permission.ACCESS_FINE_LOCATION	22.6 %
9	android.permission.ACCESS_WIFI_STATE	18.4 %
10	android.permission.RECEIVE_BOOT_COMPLETED	11.4 %
11	android.permission.READ_CONTACTS	11.2 %
12	android.permission.GET_ACCOUNTS	10.2 %
13	android.permission.CAMERA	8.8 %
14	com.android.vending.BILLING	8.3 %
15	android.permission.GET_TASKS	7.6 %
16	com.android.launcher.permission.INSTALL_SHORTCUT	6.8 %
17	android.permission.WRITE_SETTINGS	6.7 %
18	android.permission.CALL_PHONE	6.6 %
19	com.google.android.c2dm.permission.RECEIVE	6.0 %
20	android.permission.WRITE_CONTACTS	5.6 %

Table 3.5: Android permissions and the part of apps in our data set that declare that they need them.

Chapter 4

Semantic Domains

Formal analysis of Android apps needs a formal definition of the behaviour of Dalvik bytecode. We will create one using operational semantics with an approach similar to the one for the Carmel language in [Han05]. Our semantic rules for the Dalvik bytecode are based on the documentation for Dalvik [And11e], inspection of the source code for the Dalvik VM in Android [And11f], tests of handwritten `smali` code, and experiments with disassembly of compiled Java code.

To express semantic rules we need semantic domains so we can refer to the different parts of an app.

4.1 Notation

We use domains for representation of app structure, but for convenience we will add access functions using record notation for domains [Siv04]. The domain $\mathsf{D} = \mathsf{D}_1 \times \ldots \times \mathsf{D}_n$ equipped with functions $f_i : D \to D_i$ is expressed

$$\mathsf{D} = (f_1 \colon \mathsf{D}_1) \times \ldots \times (f_n \colon \mathsf{D}_n)$$

The access functions will be used in an object-oriented style where, for $d \in \mathsf{D}$, $f_i(d)$ is written $d.f_i$ and $f_i(d, a_1, \ldots, a_m)$ is written $d.f_i(a_1, \ldots, a_m)$. The notation $d[f \mapsto x]$ expresses the domain d where the value of access function f is updated to x.

4.2 App Structure

To be able to formalize the semantics we will first need a formal definition of the structure of Android apps. An app $A \in \mathsf{App}$ consists of a name, a set of classes and a set of interfaces. In addition it has a manifest, a certificate, and sets of resources, assets, and libraries:

$$\begin{aligned}\mathsf{App} = {} & (\mathit{name}\colon \mathsf{AppName}) \times \\ & (\mathit{classes}\colon \mathcal{P}(\mathsf{Class})) \times \\ & (\mathit{interfaces}\colon \mathcal{P}(\mathsf{Interface})) \times \\ & (\mathit{manifest}\colon \mathsf{Manifest}) \times \\ & (\mathit{certificate}\colon \mathsf{Certificate}) \times \\ & (\mathit{resources}\colon \mathcal{P}(\mathsf{Resource})) \times \\ & (\mathit{assets}\colon \mathcal{P}(\mathsf{Asset})) \times \\ & (\mathit{libs}\colon \mathcal{P}(\mathsf{Lib}))\end{aligned}$$

For now we are only concerned with the classes and interfaces. The others are placeholders for the details of the contents of an APK as described in Section 2.3.

A class has a name, an app in which it is defined, the Java package it belongs to, a superclass, and then sets of methods, fields, access flags and implemented interfaces:

$$\begin{aligned}\mathsf{Class} = {} & (\mathit{name}\colon \mathsf{ClassName}) \times \\ & (\mathit{app}\colon \mathsf{App}) \times \\ & (\mathit{package}\colon \mathsf{Package}) \times \\ & (\mathit{super}\colon \mathsf{Class}_{\perp}) \times \\ & (\mathit{methods}\colon \mathcal{P}(\mathsf{Method})) \times \\ & (\mathit{fields}\colon \mathcal{P}(\mathsf{Field})) \times \\ & (\mathit{accessFlags}\colon \mathcal{P}(\mathsf{AccessFlag})) \times \\ & (\mathit{implements}\colon \mathcal{P}(\mathsf{Interface}))\end{aligned}$$

For the `java.lang.Object` class, the superclass will be defined to be $\perp$, hence the domain $\mathsf{Class}_{\perp} = \mathsf{Class} \cup \{\perp\}$. Interfaces are similar to classes except that they support multiple inheritance:

$$\begin{aligned}\mathsf{Interface} = {} & (\mathit{name}\colon \mathsf{InterfaceName})\times \\ & (\mathit{app}\colon \mathsf{App})\times \\ & (\mathit{package}\colon \mathsf{Package})\times \\ & (\mathit{super}\colon \mathcal{P}(\mathsf{Interface}))\times \\ & (\mathit{methods}\colon \mathcal{P}(\mathsf{Method}))\times \\ & (\mathit{fields}\colon \mathcal{P}(\mathsf{Field}))\times \\ & (\mathit{accessFlags}\colon \mathcal{P}(\mathsf{AccessFlag}))\times \\ & (\mathit{implementedBy}\colon \mathcal{P}(\mathsf{Class}))\end{aligned}$$

In Java, packages are used to organize classes in programs, but in Dalvik, they are only used to check access violations between packages. Packages have a name, belong to an app, and the set of classes it defines:

$$\begin{aligned}\mathsf{Package} = {} & (\mathit{name}\colon \mathsf{PackageName})\times \\ & (\mathit{app}\colon \mathsf{App})\times \\ & (\mathit{classes}\colon \mathcal{P}(\mathsf{Class}))\end{aligned}$$

We have not included packages in the App domain since they are only relevant from a class perspective.

Dalvik classes, methods and fields have a set of flags that indicate their accessibility and overall properties:

$$\begin{aligned}\mathsf{AccessFlag} = {} & \{\texttt{public}, \texttt{private}, \texttt{protected}, \texttt{final}, \texttt{abstract}, \\ & \texttt{varargs}, \texttt{native}, \texttt{enum}, \texttt{constructor}\}\end{aligned}$$

Some of the flags are not relevant for all classes, methods and fields, and some of the flags may not be set at the same time, but we do not reflect these rules in the domains. Some of the Dalvik access flags are represented in other parts of the structure, for example *isStatic* on Field, and the ones defined here are placeholders and not relevant for our present purposes.

A method has a name, the class where it is implemented, a sequence of types for its arguments (a sequence A^* meaning an element from the set $\{\emptyset, A, A \times A, A \times A \times A, \ldots\}$), a return type, a function mapping locations in the method (program counter values) to instructions, a kind indicating whether the method is virtual, static, or direct (non-overridable, i.e. a constructor or private method), an integer designating the maximal number of

registers needed for local variables, a set of access flags, and finally a function mapping locations of data tables in the bytecode to the union of the disjoint sets of array data tables and jump tables for packed and sparse switches:

$$\begin{aligned}\mathsf{Method} = {} & (\mathit{name}\colon \mathsf{MethodName})\times \\ & (\mathit{class}\colon \mathsf{Class})\times \\ & (\mathit{argType}\colon \mathsf{Type}^*)\times \\ & (\mathit{returnType}\colon \mathsf{Type}\cup\{\texttt{void}\})\times \\ & (\mathit{instructionAt}\colon \mathsf{PC}\to \mathsf{Instruction})\times \\ & (\mathit{kind}\colon \mathsf{Kind})\times \\ & (\mathit{maxLocal}\colon \mathbb{N}_0)\times \\ & (\mathit{accessFlags}\colon \mathcal{P}(\mathsf{AccessFlag}))\times \\ & (\mathit{tableAt}\colon \mathsf{PC}\to \mathsf{ArrayData}\cup\mathsf{PackedSwitch}\cup\mathsf{SparseSwitch})\end{aligned}$$

The program counter domain PC is modelled as integer indices of instructions in the method with 0 being the first instruction. Ignoring the different sizes of instructions in the bytecode makes analysis simpler because the next instruction is found at $pc + 1$ but does not make the analysis less powerful [Han05].

The kind of a method can be one of the following:

$$\mathsf{Kind} = \{\texttt{virtual}, \texttt{static}, \texttt{direct}\}$$

where `direct` is used for constructors and private or final methods, `static` for static methods, and `virtual` for normal overridable methods and all methods defined in interfaces. It would be straightforward to model Kind as part of the AccessFlag domain as in the Dalvik implementation, but it seems more appropriate to keep Kind separate since its elements are mutually exclusive.

We will also need to refer to the signature of a method which is the first four components of the Method domain, except that *class* represents the class or interface where it is defined:

$$\begin{aligned}\mathsf{MethodSignature} = {} & (\mathit{name}\colon \mathsf{MethodName})\times \\ & (\mathit{class}\colon \mathsf{Class}\cup\mathsf{Interface})\times \\ & (\mathit{argType}\colon \mathsf{Type}^*)\times \\ & (\mathit{returnType}\colon \mathsf{Type}\cup\{\texttt{void}\})\end{aligned}$$

A field of a class or an interface has a name, the class or interface where it is defined, a type, an indication of whether it is a static field or not, and then its access flags:

$$\begin{aligned}\mathsf{Field} = {} & (\mathit{name}\colon \mathsf{FieldName})\times \\ & (\mathit{class}\colon \mathsf{Class} \cup \mathsf{Interface})\times \\ & (\mathit{type}\colon \mathsf{Type})\times \\ & (\mathit{isStatic}\colon \mathsf{Bool})\times \\ & (\mathit{accessFlags}\colon \mathcal{P}(\mathsf{AccessFlag}))\end{aligned}$$

DEX files may contain tables for statically defined arrays:

$$\begin{aligned}\mathsf{ArrayData} = {} & (\mathit{size}\colon \mathbb{N}_0)\times \\ & (\mathit{data}\colon \mathbb{N}_0 \to \mathsf{Prim})\end{aligned}$$

Here, the width of data types is abstracted away so the size is simply the number of elements in the array.

DEX files may also contain jump tables for switches. Dalvik uses two functionally equivalent types of switches, presumably to minimize the size of the bytecode: `packed-switch` if the switch cases are consecutive values and `sparse-switch` otherwise:

$$\begin{aligned}\mathsf{PackedSwitch} = {} & (\mathit{firstKey}\colon \mathbb{N}_0)\times \\ & (\mathit{size}\colon \mathbb{N}_0)\times \\ & (\mathit{packedTargets}\colon \mathbb{N}_0 \to \mathsf{PC})\end{aligned}$$

$$\mathsf{SparseSwitch} = (\mathit{sparseTargets}\colon \mathbb{N}_0 \to \mathsf{PC})$$

For packed switches the first case has key *firstKey* and its target is at index 0 in the *packedTargets* function, second case has key *firstKey* + 1 and is at index 1 and so on. For sparse switches, *sparseTargets* is simply a mapping from the defined cases to the jump targets.

4.3 Types

The types we model in Dalvik are either reference types or primitive types:

$$\text{Type} \quad ::= \quad \text{RefType} \mid \text{PrimType}$$

Primitive types are split into two kinds based on their width:

$$\begin{aligned} \text{PrimType} &::= \text{PrimSingle} \mid \text{PrimDouble} \\ \text{PrimSingle} &::= \texttt{boolean} \mid \texttt{char} \mid \texttt{byte} \mid \texttt{short} \mid \texttt{int} \mid \texttt{float} \\ \text{PrimDouble} &::= \texttt{long} \mid \texttt{double} \end{aligned}$$

Reference types can be either references to classes or interfaces, or to arrays:

$$\begin{aligned} \text{RefType} &::= \text{SimpleRef} \mid \text{ArrayType} \\ \text{SimpleRef} &::= \text{Class} \mid \text{Interface} \end{aligned}$$

Array types are split into two kinds to accommodate the semantics for the `filled-new-array` instruction which only accepts arrays of single width elements:

$$\begin{aligned} \text{ArrayType} &::= \text{ArrayTypeSingle} \mid \text{ArrayTypeDouble} \\ \text{ArrayTypeSingle} &::= \texttt{array}\ (\text{RefType} \mid \text{PrimSingle}) \\ \text{ArrayTypeDouble} &::= \texttt{array}\ \text{PrimDouble} \end{aligned}$$

Here, the syntactic element `array` is used to distinguish array types from otherwise structurally similar types.

4.4 Subtyping

Java allows subtypes to be used in place of a supertype, so for use in semantics for type related instructions, we define the subtype relation for reference

types, $\preceq$, as in [Han05]. A class $cl \in \mathsf{Class}$ is a proper subtype of class cl' if cl' is found in the ancestry of cl:

$$\frac{cl' \in super^*(cl)}{cl \preceq cl'}$$

where $super^*$ is the set of superclasses found by traversing the class hierarchy transitively:

$$\begin{array}{lcl} super^*(\bot) & = & \emptyset \\ super^*(cl) & = & \{cl.super\} \cup (cl.super).super^* \end{array}$$

A class is a subtype of an interface if the interface belongs to those implemented by the class or its superclasses, or to the interfaces extended by the interface, and their extension hierarchy:

$$\frac{iface \in implements^*(cl)}{cl \preceq iface}$$

$$\begin{array}{lcl} implements^*(\bot) & = & \emptyset \\ implements^*(cl) & = & cl.implements \\ & & \cup\, (cl.super).implements^* \\ & & \cup\, (cl.implements).super^* \end{array}$$

Since interfaces extend sets of interfaces, the $super^*$ function on interfaces is defined on sets:

$$super^*(ifaces) = \bigcup_{iface \in ifaces} iface.super \cup (iface.super).super^*$$

An array is a subtype of an array when the subtype relation holds for the element types:

$$\frac{t \preceq t'}{(\texttt{array}\ t) \preceq (\texttt{array}\ t')}$$

Because of this rule and the following for classes, the subtype relation is reflexive:

$$\frac{cl \in \mathsf{Class}}{cl \preceq cl}$$

4.5 Dalvik Instructions

With the overall app structure formalized, we now show the generalized instruction set for Dalvik which is used to identify instructions in the semantic rules:

$$
\begin{array}{lll}
\textsf{Instruction} & ::= & \texttt{nop} \mid \texttt{const}\ v\ c \mid \texttt{const-class}\ v\ cl \mid \texttt{move}\ v_1\ v_2 \\
& \mid & \texttt{binop}_{bop}\ v_1\ v_2\ v_3 \mid \texttt{binop-lit}_{bop}\ v_1\ v_2\ c \\
& \mid & \texttt{unop}_{uop}\ v_1\ v_2 \mid \texttt{goto}\ pc \\
& \mid & \texttt{if}\ rop\ v_1\ v_2\ pc \mid \texttt{ifz}\ rop\ v\ pc \mid \texttt{cmp}\ bias\ v_1\ v_2\ v_3 \\
\textit{Objects}: & & \\
& \mid & \texttt{new-instance}\ v\ cl \mid \texttt{const-string}\ v\ s \\
& \mid & \texttt{instance-of}\ v_1\ v_2\ type \\
& \mid & \texttt{iget}\ v_1\ v_2\ fld \mid \texttt{iput}\ v_1\ v_2\ fld \\
& \mid & \texttt{sget}\ v\ fld \mid \texttt{sput}\ v\ fld \\
\textit{Methods}: & & \\
& \mid & \texttt{invoke-virtual}\ v_1 \ldots v_n\ meth \\
& \mid & \texttt{invoke-direct}\ v_1 \ldots v_n\ meth \\
& \mid & \texttt{invoke-super}\ v_1 \ldots v_n\ meth \\
& \mid & \texttt{invoke-interface}\ v_1 \ldots v_n\ meth \\
& \mid & \texttt{invoke-static}\ v_1 \ldots v_n\ meth \mid \texttt{invoke-static}\ \varepsilon\ meth \\
& \mid & \texttt{return-void} \mid \texttt{return}\ v \mid \texttt{move-result}\ v \\
\textit{Arrays}: & & \\
& \mid & \texttt{new-array}\ v_1\ v_2\ type \mid \texttt{array-length}\ v_1\ v_2 \\
& \mid & \texttt{aget}\ v_1\ v_2\ v_3 \mid \texttt{aput}\ v_1\ v_2\ v_3 \\
& \mid & \texttt{filled-new-array}\ v_1 \ldots v_n\ type \mid \texttt{filled-new-array}\ \varepsilon\ type \\
& \mid & \texttt{fill-array-data}\ v\ pc \\
\textit{Switches}: & & \\
& \mid & \texttt{packed-switch}\ v\ pc \mid \texttt{sparse-switch}\ v\ pc \\
\textit{Exceptions}: & & \\
& \mid & \texttt{throw}\ v \mid \texttt{move-exception}\ v \mid \texttt{check-cast}\ v\ type
\end{array}
$$

This covers all our generalized instructions for Dalvik except the monitor instructions `monitor-enter` and `monitor-exit` which are used for thread safety. We consider concurrency in Dalvik as out of scope for this project due to time constraints, and therefore the two instructions are excluded.

We define semantic rules for all of the above instructions in Chapter 5, but ignore exceptions at first to keep the rules simple. We show how exceptions can be added to the semantic domains and define rules for the relevant instructions in Section 5.6.

Some instructions may take an empty list of argument registers shown as ε above.

In the instruction definitions, we have that:

$$\begin{aligned} cl &\in \mathsf{Class} \\ fld &\in \mathsf{Field} \\ meth &\in \mathsf{MethodSignature} \\ type &\in \mathsf{RefType} \end{aligned}$$

where the domains are known from Sections 4.2 and 4.3. We also have:

$$\begin{aligned} v, v_1, \ldots, v_n &\in \mathsf{Register} \\ c &\in \mathsf{Prim} \\ s &\in \mathsf{String} \\ pc &\in \mathsf{PC} \end{aligned}$$

the domains of which will be presented in Section 4.6. The remaining instruction arguments and their domains are presented in the rest of this section.

For binary and unary operations we define:

$$bop \in \mathsf{BinOp} = \{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}, \texttt{rem}, \texttt{and}, \texttt{or}, \texttt{xor}, \texttt{shl}, \texttt{shr}, \texttt{ushr}\}$$
$$uop \in \mathsf{UnOp} = \{\texttt{neg}, \texttt{not}\}$$

Besides the unary operations, there is a number of type conversion instructions, such as `int-to-long` or `double-to-int`. These instructions cannot fail (throw exceptions), they can only lose precision (in the case of shortening operations) or information (the not-a-number floating point value `NaN` becomes zero when converted to an integral type). Therefore they are safe to ignore until types are included in the analysis.

For conditional jumps, Dalvik uses the following relational operators:

$$rop \in \mathsf{RelOp} = \{\texttt{eq}, \texttt{ne}, \texttt{lt}, \texttt{le}, \texttt{gt}, \texttt{ge}\}$$

For comparisons without jumps, the $\texttt{cmp}_{bias}$ instruction exists where:

$$bias \in \mathsf{Bias} = \{\texttt{l}, \texttt{g}, \varepsilon\}$$

Bias relates to the outcome of comparisons with `NaN` but since `cmp` may also be used with `longs`, there is also an unbiased version, here represented by ε. Beside noting that these operations are available we will not go into more depth with them.

4.6 Semantic Domains

Values in our representation of Dalvik programs are either primitive values, heap references, or classes:

$$\mathsf{Val} = \mathsf{Prim} + \mathsf{Ref} + \mathsf{Class}$$

The details of the values of variables will not be relevant so primitive values can simply be represented as integers:

$$\mathsf{Prim} = \mathbb{Z}$$

DEX files may contain strings that can be used with the `const-string` instruction. In other words, they are not yet objects of the `java.lang.String` class. These are represented as sequences of characters:

$$\mathsf{String} = \mathsf{Char}^*$$

where characters represent Unicode code points and can be modelled by $\mathbb{N}_0$.

As mentioned in Section 4.2, program counters are simply integer indices of instructions:

$$\mathsf{PC} = \mathbb{N}_0$$

A program counter value can be used to give an absolute address of an instruction in an app:

$$\mathsf{Addr} = \mathsf{Method} \times \mathsf{PC}$$

The Dalvik VM uses registers for computation and storage of local variables. It has a special register for holding a return value and then 2^{16} numbered regular registers which we will generalize to $\mathbb{N}_0$:

$$\mathsf{Register} = \mathbb{N}_0 \cup \{\texttt{retval}\}$$

We will also need to represent a function to map registers to values, with $\bot$ in the case of undefined register contents:

$$\mathsf{LocalReg} = \mathsf{Register} \to \mathsf{Val}_\bot$$

For storing objects, arrays and static fields, Dalvik uses the heap. To simplify the representation, we will use a static heap $S \in \mathsf{StaticHeap}$ which simply maps fields to values for fields $\mathit{fld} \in \mathsf{Field}$ where $\mathit{fld}.\mathit{isStatic}$ is true:

$$\mathsf{StaticHeap} = \mathsf{Field} \to \mathsf{Val}$$

The normal heap will map references to either objects or arrays:

$$\mathsf{Heap} = \mathsf{Ref} \to (\mathsf{Object} + \mathsf{Array})$$

References are abstract locations or a null reference. In Dalvik bytecode, null references are represented by the number zero but we use `null` to be able to distinguish them in the semantics:

$$\mathsf{Ref} = \mathsf{Location} \cup \{\texttt{null}\}$$

Since Dalvik does not support pointers and pointer arithmetic, it will not be necessary to know what locations are except that we can model an arbitrary number of unique locations.

An object belongs to a class and has a mapping of fields to values:

$$\mathsf{Object} = (\mathit{class}\colon \mathsf{Class}) \times (\mathit{field}\colon \mathsf{Field} \to \mathsf{Val})$$

The class of a field is also accessible from the field but this value is the class or interface where the field is defined while the above class is the instantiated concrete class.

Similarly, arrays have a type, a size and a mapping of indices to values:

$$\mathsf{Array} = (\mathit{type}\colon \mathsf{ArrayType}) \times (\mathit{length}\colon \mathbb{N}_0) \times (\mathit{value}\colon \mathbb{N}_0 \to \mathsf{Val})$$

4.7 Program Configurations

The configuration that we base our semantic rules on is defined as:

$$\begin{aligned} \mathsf{Configuration} &= \mathsf{StaticHeap} \times \mathsf{Heap} \times \mathsf{CallStack} \\ \mathsf{CallStack} &= \mathsf{Frame}^* \end{aligned}$$

Each configuration consists of the static and dynamic heap, both defined in the section above, and finally a call stack which is a sequence of frames. Each frame in the stack includes a method, a program counter offset into the method, and the local registers for the method:

$$\mathsf{Frame} = \mathsf{Method} \times \mathsf{PC} \times \mathsf{LocalReg}$$

Handling exceptions requires another type of frames, but we make the necessary changes when treating exceptions in Section 5.6. We use the following notation when referring to a call stack in CallStack:

$$\langle m, pc, R \rangle :: SF$$

where $\langle m, pc, R \rangle$ represents the top stack frame (with $m \in \mathsf{Method}$, $pc \in \mathsf{PC}$, $R \in \mathsf{LocalReg}$) and SF represents the rest of the stack. The :: operator is used to concatenate stack frames.

The semantic rules can now be defined, using reduction rules of the form:

$$A \vdash C \Longrightarrow C'$$

Where the app $A \in \mathsf{App}$ and $C, C' \in \mathsf{Configuration}$, or, equivalently:

$$A \vdash \langle S, H, SF \rangle \Longrightarrow \langle S', H', SF' \rangle$$

where $S, S' \in \mathsf{StaticHeap}$, $H, H' \in \mathsf{Heap}$, and $SF, SF' \in \mathsf{CallStack}$.

4.8 Entry Points and Termination State

Unlike Java programs, Android apps have no `main()` method where the application starts. Android applications have a set of entry points. Which one is used is based on the kind of application and how the application is

started. The entry points include constructors and `onCreate()` methods for Activities, Services, Content Providers, and Broadcast Receivers.

Android applications are not supposed to terminate by themselves, and their termination points vary depending on the type of the application. Once an application has been started, it is not supposed to be terminated unless the system runs out of memory, or if the application is an `Activity` and its `finish()` method is called. In most cases, the application has an `onDestroy()` method which is called as the last method, but there is no guarantee as the application can either kill itself directly, or the system can kill the application due to lack of memory. Therefore, we cannot presume any termination state, and we therefore do not provide any specific termination configuration for our semantics.

Chapter 5

Semantic Rules

With the domains and the notation defined, we are now ready to define the actual semantic rules. We start with regular instructions grouped logically together while ignoring exceptions until Section 5.6 where we show how they can be added. Concurrency and related Dalvik instructions are out of scope, but are discussed in Section 8.2. The semantic rules in this chapter are also summarized in Appendix B.

5.1 Imperative Core

The first group of instructions for which we define semantic rules is the one concerned with basic flow within a method, constants, comparisons and branching. The first rule of the imperative core instructions in Dalvik is the rule for the `nop` instruction. The instruction is also known as no-op, or no-operation in other languages, and does nothing except increment the program counter. Even though the instruction does essentially nothing it is still used, e.g., for padding and alignment around inlined data tables. The configuration is just updated with an increased program counter in order to move to the next instruction:

$$\frac{m.instructionAt(pc) = \texttt{nop}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

We use $m.instructionAt(pc)$ where $m \in \mathsf{Method}$, $pc \in \mathsf{PC}$ to identify the instruction we are working with.

Constants are put in registers using the `const` instruction. This requires an update of the registers on the next program counter where the destination

register is mapped to the constant:

$$\frac{m.instructionAt(pc) = \texttt{const}\ \ v\ \ c}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v \mapsto c]\rangle :: SF\rangle}$$

where the constant c can be a value of any of the primitive types in Dalvik. A specialized similar instruction is `const-class` which maps the destination register to a Dalvik class instead of a primitive typed value:

$$\frac{m.instructionAt(pc) = \texttt{const-class}\ \ v\ \ cl}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v \mapsto cl]\rangle :: SF\rangle}$$

Copying content from one register to another is done using the `move` instruction. The semantics are similar to `const` but it has to look up the content of register v_2:

$$\frac{m.instructionAt(pc) = \texttt{move}\ \ v_1\ \ v_2}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto R(v_2)]\rangle :: SF\rangle}$$

In general, the first argument of an instruction is the destination registers when a destination is relevant.

Binary operations on primitive types are done using the $\texttt{binop}_{op}$ instruction where $op \in \mathsf{BinOp}$ as defined in Section 4.5. We use an auxiliary function to carry out the operation for the given operator:

$$binOp_{op}(c_1, c_2) = c_1\ op\ c_2$$

The implementation of this is trivial and is skipped here. The auxiliary function is used in the semantic rule where the destination register is mapped to the result:

$$\frac{m.instructionAt(pc) = \texttt{binop}_{op}\ \ v_1\ \ v_2\ \ v_3 \qquad c = binOp_{op}(R(v_2), R(v_3))}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto c]\rangle :: SF\rangle}$$

For binary operations with a literal value, the same auxiliary function is used:

$$\frac{m.instructionAt(pc) = \texttt{binop-lit}_{op}\ \ v_1\ \ v_2\ \ c \qquad c' = binOp_{op}(R(v_2), c)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto c']\rangle :: SF\rangle}$$

For unary operations, such as negation, we use a similar auxiliary function:

$$unOp_{op}(c) = op\ \ c$$

where $op \in \mathsf{UnOp}$, and the implementation again is trivial. The semantic rule is then:

$$\frac{m.instructionAt(pc) = \texttt{unop}_{op}\ \ v_1\ \ v_2 \qquad c = unOp_{op}(R(v_2))}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto c]\rangle :: SF\rangle}$$

There are several ways of branching in Dalvik bytecode, and the first we will look at is `goto`:

$$\frac{m.instructionAt(pc) = \texttt{goto}\ \ pc'}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc', R\rangle :: SF\rangle}$$

where the program counter is simply updated in the new configuration to the one which was specified as the argument to the instruction. The new program counter must be an address within the current method, but this is checked by the bytecode verifier.

For conditional branching we use an auxiliary function:

$$relOp_{op}(c_1, c_2) = c_1\ op\ c_2$$

where $op \in \mathsf{RelOp}$. The result of the method is either true or false, depending of the relation between the two variables. An example implementation of this, for the operator *eq* is:

$$relOp_{eq}(c_1, c_2) = \begin{cases} true & \text{if } \ c_1 = c_2 \\ false & \text{otherwise} \end{cases}$$

The function is used in our two rules for `if`:

$$\frac{m.instructionAt(pc) = \texttt{if}\ \ op\ \ v_1\ \ v_2\ \ pc' \qquad relOp_{op}(R(v_1), R(v_2))}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc', R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{if}\ \ op\ \ v_1\ \ v_2\ \ pc' \qquad \neg relOp_{op}(R(v_1), R(v_2))}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

The first applies when $relOp_{op}$ is true, and the second when it is false. If the result is true, the program counter is updated to the new one given as an argument for the instruction. Otherwise, we move to the next instruction in the method. There is another similar instruction in Dalvik, `ifz`, where the comparison is always with zero. We have two similar rules for this instruction:

$$\frac{m.instructionAt(pc) = \texttt{ifz}\ \ op\ \ v\ \ pc' \qquad relOp_{op}(R(v), 0)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc', R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{ifz}\ \ op\ \ v\ \ pc' \qquad \neg relOp_{op}(R(v), 0)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

The primitive types `long`, `float` and `double` can be compared using the *cmp* instruction. Again we use an auxiliary function for the result. The function should return 0 if the values of the two arguments given are equal, 1 if the second is larger, or -1 if the first is larger. The "bias" is for floating point operations and indicate how `NaN` comparisons are treated: `g`-bias returns

1 for `NaN` comparisons, and `l`-bias returns -1. The actual implementation is not relevant for the semantics and would be trivial to implement from the documentation. We do not model special floating point values such as `NaN` but discuss it here because it is embedded in the syntax of the Dalvik instruction set like binary and unary operations. The instruction maps the destination register to the result from the auxiliary function:

$$\frac{m.instructionAt(pc) = \texttt{cmp}\ \ bias\ \ v_1\ \ v_2\ \ v_3 \qquad c = cmp_{bias}(R(v_2), R(v_3))}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto c]\rangle :: SF\rangle}$$

5.2 Objects

We now turn focus to the part of instructions concerned with objects. In Dalvik, objects are allocated on the heap, and beside defining how to allocate them we will also define rules for comparing object types, getting and setting field values on objects as well as for static fields.

To allocate objects on the heap, we use an auxiliary function:

$$\begin{array}{l} newObject\colon \mathsf{Heap} \times \mathsf{Class} \to \mathsf{Heap} \times \mathsf{Ref} \\ newObject(H, cl) = (H', loc) \end{array}$$

where $loc \notin \mathrm{dom}(H)$, $H' = H[loc \mapsto o]$, $o \in \mathsf{Object}$, and $o.class = cl$. The function takes an existing heap and a class, and returns a modified heap along with a reference to a new location for the allocated object. It is used in the rule for `new-instance` which is supplied with the class for which a new instance should be created and a destination register for the reference to the object:

$$\frac{m.instructionAt(pc) = \texttt{new-instance}\ \ v\ \ cl \qquad (H', loc) = newObject(H, cl)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc+1, R[v \mapsto loc]\rangle :: SF\rangle}$$

We have previously defined semantic rules for two variations of `const`, but Dalvik has one more variant that requires its own case: `const-string`. It handles constant strings defined at compile-time. The second argument to the instruction is a string, and the instruction converts this primitive string into an object of the `String` class on the heap, and maps the destination register to the new reference:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{const-string}\ \ v\ \ s \\ (H', loc) = newObject(H, \texttt{String}) \qquad o = H'(loc) \qquad H'[loc \mapsto o[value \mapsto s]] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc+1, R[v \mapsto loc]\rangle :: SF\rangle}$$

To check if a given reference is an instance of a specific type, the instruction `instance-of` can be used. It maps a destination register to either 1 or 0, depending on whether or not the reference-bearing register is of the specified type:

$$\frac{m.instructionAt(pc) = \texttt{instance-of}\ \ v_1\ \ v_2\ \ type \qquad loc = R(v_2) \qquad o = H(loc) \qquad c = \begin{cases} 1 & \text{if}\ \ o \in \mathsf{Object} \wedge o.class \preceq type \vee \\ & \quad o \in \mathsf{Array} \wedge o.type \preceq type \\ 0 & \text{otherwise} \end{cases}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto c]\rangle :: SF\rangle}$$

Notice how the referenced object must be in the Object or Array domain before the types can be compared. The reason for this is that arrays are also allocated on the heap. We describe arrays in more details in Section 5.4.

Fields on objects are defined as mappings to values, which makes it easy to update and reference them on object instances. For the instruction `iget` the referenced object is looked up on the heap, and the destination register is mapped to the field of the referenced object instance:

$$\frac{m.instructionAt(pc) = \texttt{iget}\ \ v_1\ \ v_2\ \ fld \qquad R(v_2) = loc \qquad o = H(loc)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto o.fld]\rangle :: SF\rangle}$$

For the `iput` instruction a similar object lookup is performed, but for this instruction we update the field mapping on the object to the value of the source register:

$$\frac{m.instructionAt(pc) = \texttt{iput}\ \ v_1\ \ v_2\ \ fld \qquad R(v_2) = loc \qquad o = H(loc)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H[loc \mapsto o[fld \mapsto R(v_1)]], \langle m, pc+1, R\rangle :: SF\rangle}$$

Unlike other instructions, put-instructions take the source register as the first argument to be symmetric with get-instructions.

Static fields are all placed on the static heap, S, which makes accessing and updating the fields trivial:

$$\frac{m.instructionAt(pc) = \texttt{sget}\ \ v\ \ fld}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v \mapsto S(fld)] :: SF\rangle\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{sput}\ \ v\ \ fld}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S[fld \mapsto R(v)], H, \langle m, pc+1, R :: SF\rangle\rangle}$$

5.3 Methods

We now look at instructions for invoking methods and returning from them. Dalvik, like Java, uses *dynamic dispatch*, where the implementation of a

method is looked up at runtime due to the Java class hierarchy. The various invoke instructions use different auxiliary functions to do the lookup at runtime. Before we formalise these auxiliary functions, we define a notation to indicate that a method signature $meth \in \mathsf{MethodSignature}$ is compatible with a given method $m \in \mathsf{Method}$. We write $meth \triangleleft m$ which is defined as:

$$\begin{array}{lll} meth \triangleleft m & \text{iff} & m.name = meth.name \wedge \\ & & m.argType = meth.argType \wedge \\ & & m.returnType = meth.returnType \end{array}$$

Furthermore, we define the number of arguments for a method to be the length, $|\cdot|$, of the *argType* domain sequence:

$$arity(meth) = |meth.argType|$$

To resolve the actual method that has to be called, we use the following function to search through the class hierarchy for virtual methods (all non-static methods that are overridable or defined in interfaces):

$$resolveMethod(meth, cl) = \begin{cases} \bot & \text{if } cl = \bot \\ m & \text{if } m \in cl.methods \wedge meth \triangleleft m \wedge \\ & \quad m.kind = \texttt{virtual} \\ resolveMethod(meth, cl.super) & \text{otherwise} \end{cases}$$

The semantic rule for `invoke-virtual` is then defined as:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = loc \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ n = arity(meth) \qquad m' = resolveMethod(meth, o.class) \\ R' = [0 \mapsto \bot, \ldots, m'.maxLocal \mapsto \bot, \\ \quad m'.maxLocal + 1 \mapsto v_1, \ldots, m'.maxLocal + arity(m') \mapsto v_n] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

The instruction receives n arguments and the signature of the method to invoke. The first argument v_1 is a reference to the object on which the method should be invoked. The location of the method is resolved using *resolveMethod* and this method is put into a new frame on top of the call stack, with the program counter set to 0. A new set of local registers, R', is created, where the first $m'.maxLocal$ registers are mapped to $\bot$ from Val such that they are initially undefined. The arguments are then mapped into the next registers.

Because we are not addressing exceptions yet, *resolveMethod* returning $\bot$ will result in a stuck configuration.

Invoking methods through an interface in Dalvik uses dynamic dispatch as `invoke-virtual`, and the semantic rules are the same because the actual implementation of an invoked method is found in the runtime class, or any of its superclasses:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-interface}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = loc \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ n = arity(meth) \qquad m' = resolveMethod(meth, o.class) \\ R' = [0 \mapsto \bot, \ldots, m'.maxLocal \mapsto \bot, \\ \quad m'.maxLocal + 1 \mapsto v_1, \ldots, m'.maxLocal + arity(m') \mapsto v_n] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

Non-static private methods and constructors are invoked using `invoke-direct`. Another auxiliary function is needed to resolve the method at runtime:

$$resolveDirectMethod(meth, cl) = \begin{cases} m & \text{if } m \in cl.methods \wedge meth \triangleleft m \wedge m.kind = \texttt{direct} \\ \bot & \text{otherwise} \end{cases}$$

it is similar to the one before, except that it should not recurse through the class hierarchy as direct methods are always present in the class of the object, the method is being invoked on. Other than that, the semantic rule is the same:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-direct}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = loc \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ n = arity(meth) \qquad m' = resolveDirectMethod(meth, o.class) \\ R' = [0 \mapsto \bot, \ldots, m'.maxLocal \mapsto \bot, \\ \quad m'.maxLocal + 1 \mapsto v_1, \ldots, m'.maxLocal + arity(m') \mapsto v_n] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

In Dalvik, a method in a parent (*super*) class is invoked with the `invoke-super` instruction. The method which has to be resolved at runtime is then located in *o.class.super* or above:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-super}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = loc \qquad loc \neq \texttt{null} \qquad o = H(loc) \qquad o.class.super \neq \bot \\ n = arity(meth) \qquad m' = resolveMethod(meth, o.class.super) \\ R' = [0 \mapsto \bot, \ldots, m'.maxLocal \mapsto \bot, \\ \quad m'.maxLocal + 1 \mapsto v_1, \ldots, m'.maxLocal + arity(m') \mapsto v_n] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

Static methods in Dalvik can be inherited, but since they are not called on objects, resolving the method can be done from merely the method itself.

We use the recursive auxiliary function *resolveStaticMethod* for this:

$$\textit{resolveStaticMethod}(\textit{meth}, \textit{cl}) = \begin{cases} \bot & \text{if } \textit{cl} = \bot \\ \textit{m} & \text{if } \textit{m} \in \textit{cl.methods} \wedge \textit{meth} \triangleleft \textit{m} \wedge \\ & \quad \textit{m.kind} = \texttt{static} \\ \textit{resolveStaticMethod}(\textit{meth}, \textit{cl.super}) & \text{otherwise} \end{cases}$$

Static methods in Dalvik are invoked in a similar way as the other type of methods we have looked at:

$$\frac{\begin{array}{c}\textit{m.instructionAt}(\textit{pc}) = \texttt{invoke-static } v_1 \ldots v_n \ \textit{meth} \\ n = \textit{arity}(\textit{meth}) \qquad \textit{m}' = \textit{resolveStaticMethod}(\textit{meth}, \textit{meth.class}) \\ R' = [0 \mapsto \bot, \ldots, \textit{m}'.\textit{maxLocal} \mapsto \bot, \\ \textit{m}'.\textit{maxLocal} + 1 \mapsto v_1, \ldots, \textit{m}'.\textit{maxLocal} + \textit{arity}(\textit{m}') \mapsto v_n]\end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

where *meth.class* is the class from where the method resolving should begin. With no object reference, the instruction is not guaranteed at least one argument to the static method, thus we need another rule for this case:

$$\frac{\begin{array}{c}\textit{m.instructionAt}(\textit{pc}) = \texttt{invoke-static } \varepsilon \ \textit{meth} \\ n = \textit{arity}(\textit{meth}) \qquad \textit{m}' = \textit{resolveStaticMethod}(\textit{meth}, \textit{meth.class}) \\ R' = [0 \mapsto \bot, \ldots, \textit{m}'.\textit{maxLocal} \mapsto \bot]\end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

When methods return in Dalvik, the top frame of the call stack is removed. This returns the control to the method of the frame now on top. When a method has no return value, the semantic rule is:

$$\frac{\textit{m.instructionAt}(\textit{pc}) = \texttt{return-void}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: \langle m', pc', R'\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', pc' + 1, R'\rangle :: SF\rangle}$$

When the method returns a value, the special register `retval` is mapped to the value in the frame now on top of the call stack:

$$\frac{\textit{m.instructionAt}(\textit{pc}) = \texttt{return } v}{\begin{array}{r}A \vdash \langle S, H, \langle m, pc, R\rangle :: \langle m', pc', R'\rangle :: SF\rangle \Longrightarrow \\ \langle S, H, \langle m', pc' + 1, R'[\texttt{retval} \mapsto R(v)]\rangle :: SF\rangle\end{array}}$$

The content of the special register is only available on the very next operation after something has been put in it. Therefore, the instruction `move-result` should be used right after `return` in order to move the result from the special register `retval` to another register that can be read at a later point:

$$\frac{\textit{m.instructionAt}(\textit{pc}) = \texttt{move-result } v}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc + 1, R[v \mapsto R(\texttt{retval})]\rangle :: SF\rangle}$$

5.4 Arrays

Dalvik supports Java arrays, including arrays of arrays. Arrays are allocated on the heap in a way similar to objects. We use the following function to update the heap and get a reference to the new array:

$$\begin{array}{l} \mathit{newArray}\colon \mathsf{Heap} \times \mathbb{N} \times \mathsf{ArrayType} \to \mathsf{Heap} \times \mathsf{Ref} \\ \mathit{newArray}(H, n, \mathit{type}) = (H', \mathit{loc}) \end{array}$$

where $\mathit{loc} \notin \mathrm{dom}(H)$, $H' = H[\mathit{loc} \mapsto a]$, $a \in \mathsf{Array}$, $a.\mathit{length} = n$, and $a.\mathit{type} = \mathit{type}$.

Creating a new array is done using the `new-array` instruction:

$$\frac{\begin{array}{c} \mathit{m.instructionAt}(\mathit{pc}) = \texttt{new-array}\ \ v_1\ \ v_2\ \ \mathit{type} \\ \mathit{type} \in \mathsf{ArrayType} \qquad n = R(v_2) \geq 0 \qquad (H', \mathit{loc}) = \mathit{newArray}(H, n, \mathit{type}) \end{array}}{A \vdash \langle S, H, \langle m, \mathit{pc}, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, \mathit{pc}+1, R[v_1 \mapsto \mathit{loc}]\rangle :: SF\rangle}$$

It can create arrays of all reference or primitive types.

The size of an array can be retrieved using the `array-length` instruction where the destination register is mapped to the length of the array:

$$\frac{\mathit{m.instructionAt}(\mathit{pc}) = \texttt{array-length}\ \ v_1\ \ v_2 \qquad R(v_2) = \mathit{loc} \qquad a = H(\mathit{loc})}{A \vdash \langle S, H, \langle m, \mathit{pc}, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, \mathit{pc}, R[v_1 \mapsto a.\mathit{length}]\rangle :: SF\rangle}$$

Retrieving and updating values in arrays requires the heap reference to the array, an index into the array and a destination or source. The values of the array are found in the function $a.\mathit{value}$ where $a \in \mathsf{Array}$:

$$\frac{\mathit{m.instructionAt}(\mathit{pc}) = \texttt{aget}\ \ v_1\ \ v_2\ \ v_3 \qquad R(v_2) = \mathit{loc} \qquad a = H(\mathit{loc}) \qquad i = R(v_3)}{A \vdash \langle S, H, \langle m, \mathit{pc}, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, \mathit{pc}+1, R[v_1 \mapsto a.\mathit{value}(i)]\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} \mathit{m.instructionAt}(\mathit{pc}) = \texttt{aput}\ \ v_1\ \ v_2\ \ v_3 \qquad R(v_2) = \mathit{loc} \qquad a = H(\mathit{loc}) \qquad i = R(v_3) \\ \mathit{value}' = a.\mathit{value}[i \mapsto R(v_1)] \qquad a' = a[\mathit{value} \mapsto \mathit{value}'] \end{array}}{A \vdash \langle S, H, \langle m, \mathit{pc}, R\rangle :: SF\rangle \Longrightarrow \langle S, H[\mathit{loc} \mapsto a'], \langle m, \mathit{pc}+1, R\rangle :: SF\rangle}$$

Dalvik supports the instruction `filled-new-array` which pre-fills a new array with given values. There cannot be more than five values, and the type of the array must be in the domain ArrayTypeSingle. Once the array has been allocated, and the values have been mapped in the array, the heap is updated with the modified array and the `retval` register is mapped to the location of the array:

$$\frac{\begin{array}{c} \mathit{m.instructionAt}(\mathit{pc}) = \texttt{filled-new-array}\ \ v_1 \ldots v_n\ \ \mathit{type} \\ \mathit{type} \in \mathsf{ArrayTypeSingle} \qquad 1 \leq n \leq 5 \\ (H', \mathit{loc}) = \mathit{newArray}(H, n, \mathit{type}) \qquad a = H'(\mathit{loc}) \\ \mathit{value}' = a.\mathit{value}[0 \mapsto R(v_1), \ldots, n-1 \mapsto R(v_n)] \qquad a' = a[\mathit{value} \mapsto \mathit{value}'] \end{array}}{A \vdash \langle S, H, \langle m, \mathit{pc}, R\rangle :: SF\rangle \Longrightarrow \langle S, H'[\mathit{loc} \mapsto a'], \langle m, \mathit{pc}+1, R[\texttt{retval} \mapsto \mathit{loc}]\rangle :: SF\rangle}$$

Since the instruction does not require content arguments, it is possible to create an empty array:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{filled-new-array}\ \ \varepsilon\ \ type \\ type \in \mathsf{ArrayTypeSingle} \qquad (H', loc) = newArray(H, 0, type) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc+1, R[\texttt{retval} \mapsto loc]\rangle :: SF\rangle}$$

The final instruction for arrays is `fill-array-data`. It fills an existing array with data found in a static table. The array will be filled from the beginning (index 0) and if there are fewer elements in the data table than the array provides space for, the rest of the array is left unchanged. The static table can be found using a program counter value in the current method, where the table $d \in \mathsf{ArrayData}$ and the data can be retrieved using a function *data*:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{fill-array-data}\ \ v\ \ pc' \\ R(v) = loc \qquad loc \neq \texttt{null} \qquad a = H(loc) \qquad d = m.tableAt(pc') \\ value' = a.value[0 \mapsto d.data(0), \ldots, d.size - 1 \mapsto d.data(d.size - 1)] \\ a' = a[value \mapsto value'] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H[loc \mapsto a'], \langle m, pc+1, R\rangle :: SF\rangle}$$

5.5 Switches

Dalvik has support for switch instructions with two types of switches: packed and sparse. The `sparse-switch` instruction simply looks up the received key in the target table while `packed-switch` calculates the table index to look at from the lookup key and the value of the first key in the table. In both cases, control is transferred to the instruction at the new program counter if a match is found. Otherwise, control falls through to the next instruction as in `nop`.

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{packed-switch}\ \ v\ \ pc' \qquad s = m.tableAt(pc') \\ i = R(v) - s.firstKey \qquad i \in dom(s.packedTargets) \qquad pc'' = s.packedTargets(i) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc'', R\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{packed-switch}\ \ v\ \ pc' \qquad s = m.tableAt(pc') \\ i = R(v) - s.firstKey \qquad i \notin dom(s.packedTargets) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{sparse-switch}\ \ v\ \ pc' \qquad s = m.tableAt(pc') \\ R(v) \in dom(s.sparseTargets) \qquad pc'' = s.sparseTargets(R(v)) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc'', R\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{sparse-switch}\ \ v\ \ pc' \qquad s = m.tableAt(pc') \\ R(v) \notin dom(s.sparseTargets) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

5.6 Exception Semantics

Exceptions can be thrown in two situations: Either explicitly using the `throw` instruction, or by the system in case of a runtime error, such as a null dereference. Many instructions may throw these runtime exceptions, but we have not modelled that in the previous sections to keep the complexity of the rules down. Without exception rules, the semantics will end in a stuck configuration in those cases. However, in this section we show how the semantics can be expanded with runtime exceptions. Since exceptions are used explicitly in almost all apps as seen in Section 3.4, and implicitly in *all* apps because of runtime exceptions, they are important to consider.

5.6.1 Exception Domains

Exception handlers belong to methods so we extend the Method domain with a function mapping from exception handler indices to the corresponding exception handlers. The indices are defined by the compiler which has the responsibility of finding the right order of the exception handlers corresponding to their catch types and placements in the source code, possibly nested within each other.

$$handlers\colon \mathbb{N}_0 \to \mathsf{ExcHandler}$$

An exception handler has a type for the exceptions it may catch, a program counter value pointing to the handler code, and program counter values defining the boundaries of the region covered by the exception handler:

$$\begin{aligned}\mathsf{ExcHandler} = {} & (catchType\colon \mathsf{Class}_\perp)\times \\ & (handlerAddr\colon \mathsf{PC})\times \\ & (startAddr\colon \mathsf{PC})\times \\ & (endAddr\colon \mathsf{PC})\end{aligned}$$

The $\perp$ element in the domain for *catchType* represents a Java `finally` clause. In Java bytecode, a `finally` block is stored as a subroutine, and instructions to jump to that subroutine are inserted at the end of the `try` block and at the end of each `catch` block. The subroutine is also registered as a catch-all exception handler used in case none of the others match. This means that the finally block will always be run. Dalvik works the same way, except that the finally blocks are inlined because it does not support

subroutines. This means that a Java `finally` block in Dalvik does not require special treatment and can simply be used as a normal `catch` block that catches all exceptions in addition to the places where its code is inlined.

Exceptions that are not handled in the method where it is thrown are put on the call stack for the next method's exception handlers to try to handle. To represent that, we modify the CallStack domain to make it possible for the top frame to be an exception frame:

$$\mathsf{CallStack} = (\mathsf{Frame} + \mathsf{ExcFrame}) \times \mathsf{Frame}^*$$

An exception frame contains the location of its corresponding exception object on the heap and the address of the instruction that threw the exception:

$$\mathsf{ExcFrame} = \mathsf{Location} \times \mathsf{Method} \times \mathsf{PC}$$

5.6.2 Exception Rules

First, we define the semantic rules for the instruction `throw` which can throw any subclass of `Throwable` as an exception[1]. There are two cases, one where an appropriate exception handler is found in the local method, and one where it is not:

$$\frac{R(v) = loc_E \qquad cl = H(loc_E).class \qquad cl \preceq \texttt{Throwable} \qquad m.instructionAt(pc) = \texttt{throw}\ v \qquad findHandler(m, pc, cl) = pc'}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc', R[\texttt{retval} \mapsto loc_E]\rangle :: SF\rangle}$$

$$\frac{R(v) = loc_E \qquad cl = H(loc_E).class \qquad cl \preceq \texttt{Throwable} \qquad m.instructionAt(pc) = \texttt{throw}\ v \qquad findHandler(m, pc, cl) = \bot}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle loc_E, m, pc\rangle :: SF\rangle}$$

We use the auxiliary function *findHandler* to find an exception handler matching the location in the method and the given exception class. Before we define this function, we define two other auxiliary functions. The *canHandle* function is able to determine if an exception handler (h) can handle the given exception (cl_E) for the specific location in the method, and if their types match:

$$\begin{aligned} canHandle(h, pc, cl_E) \quad \equiv \quad & h.startAddr \leq pc \leq h.endAddr \wedge \\ & (cl_E \preceq h.catchType \vee h.catchType = \bot) \end{aligned}$$

[1] By "exception", we always mean a subclass of `Throwable`.

where h is a handler in the domain ExcHandler, $pc \in \mathsf{PC}$ and $cl_E \in \mathsf{Class} \wedge cl_E \preceq \texttt{Throwable}$. The function is used by *isFirstHandler* which determines if the specified handler is the first applicable handler (the one with the highest index) in the given set:

$$\begin{aligned} \mathit{isFirstHandler}(\eta, i, pc, cl_E) \quad \equiv \quad & \mathit{canHandle}(\eta(i), pc, cl_E) \wedge \\ & (\forall j \leq i \colon \mathit{canHandle}(\eta(j), pc, cl_E)) \end{aligned}$$

where η is a function mapping indices to handlers for a specific method, i is the index for a specific handler, $pc \in \mathsf{PC}$ and $cl_E \in \mathsf{Class} \wedge cl_E \preceq \texttt{Throwable}$.

We can now define *findHandler*:

$$\mathit{findHandler}(m, pc, cl_E) = \begin{cases} \eta(i).\mathit{handlerAddr} & \text{if } \eta = m.\mathit{handlers} \wedge \mathrm{dom}(\eta) \neq \emptyset \wedge \\ & \quad \exists i \colon \mathit{isFirstHandler}(\eta, i, pc, cl_E) \\ \bot & \text{otherwise} \end{cases}$$

which returns the first exception handler for the given exception class in the method, or $\bot$ in case no handler was found.

Next, we need semantic rules for when an exception frame is on top of the stack. In this case, two things can happen. If the local method (the method frame below the exception frame) has an appropriate exception handler, the exception frame is removed from the stack, the program counter is updated to the location of the handler, and the `retval` register is mapped to the exception in the new top frame:

$$\frac{cl = H(loc_E).\mathit{class} \qquad \mathit{findHandler}(m, pc, cl) = pc'}{\begin{aligned} A \vdash \langle S, H, \langle loc_E, m_E, pc_E \rangle :: \langle m, pc, R \rangle :: SF \rangle \Longrightarrow \\ \langle S, H, \langle m, pc', R[\texttt{retval} \mapsto loc_E] \rangle :: SF \rangle \end{aligned}}$$

If the local method does not have an appropriate exception handler, the exception frame is removed from the top of the stack, so is the second one, and a new exception frame is added instead. The new exception frame has a method and a program counter corresponding to the method frame which was just removed, thus it is the same as re-throwing the exception to the next method in the call hierarchy.

$$\frac{cl = H(loc_E).\mathit{class} \qquad \mathit{findHandler}(m, pc, cl) = \bot}{A \vdash \langle S, H, \langle loc_E, m_E, pc_E \rangle :: \langle m, pc, R \rangle :: SF \rangle \Longrightarrow \langle S, H, \langle loc_E, m, pc \rangle :: SF \rangle}$$

In an exception handler, the first instruction must be `move-exception` if the exception object is needed by the handler. This instruction is similar to `move-result` except that the object must be an exception:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{move-exception}\ v \\ loc_E = R(v) \qquad H(loc_E).class \preceq \texttt{Throwable} \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v \mapsto R(\texttt{retval})]\rangle :: SF\rangle}$$

5.6.3 Runtime Exceptions

The previous section covered explicitly thrown exceptions, and now we will show how to handle runtime exceptions. The Dalvik instruction `check-cast` checks whether or not a type cast is possible. The instruction throws a `ClassCastException` if it is not. First, we define the case where the type cast is possible and no exception is thrown:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{check-cast}\ \ v\ \ type \\ loc = R(v) \qquad o = H(loc) \\ (o \in \mathsf{Object} \wedge o.class \preceq type) \vee (o \in \mathsf{Array} \wedge o.type \preceq type) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

If the given type is not a subtype of the instance type, the exception is thrown and again we have two cases: One where there is a local handler, and one where there is not:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{check-cast}\ \ v\ \ type \\ loc = R(v) \qquad o = H(loc) \\ (o \in \mathsf{Object} \wedge o.class \npreceq type) \vee (o \in \mathsf{Array} \wedge o.type \npreceq type) \\ findHandler(m, pc, \texttt{ClassCastException}) = pc' \\ (H', loc_E) = newObject(H, \texttt{ClassCastException}) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc', R[\texttt{retval} \mapsto loc_E]\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{check-cast}\ \ v\ \ type \\ loc = R(v) \qquad o = H(loc) \\ (o \in \mathsf{Object} \wedge o.class \npreceq type) \vee (o \in \mathsf{Array} \wedge o.type \npreceq type) \\ findHandler(m, pc, \texttt{ClassCastException}) = \bot \\ (H', loc_E) = newObject(H, \texttt{ClassCastException}) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle loc_E, m, pc\rangle :: SF\rangle}$$

Notice how we allocate a new exception object before the exception is either put in `retval` or in an exception frame.

Null-pointer exceptions are common errors in Java, and they can be thrown in several of the Dalvik instructions as well, including `throw`. We define two rules for `invoke-virtual` that handle the case where the given object reference is *null*. They are similar to the previous exception rules, as the first one fits when there is a local handler and the second one when there is not. They both allocate the `NullPointerException` on the heap before the exception is used:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = \texttt{null} \qquad findHandler(m, pc, \texttt{NullPointerException}) = pc' \\ (H', loc_E) = newObject(H, \texttt{NullPointerException}) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc', R[\texttt{retval} \mapsto loc_E]\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = \texttt{null} \qquad findHandler(m, pc, \texttt{NullPointerException}) = \bot \\ (H', loc_E) = newObject(H, \texttt{NullPointerException}) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle loc_E, m, pc\rangle :: SF\rangle}$$

Chapter 6

Flow Logic

In this chapter we define a control flow analysis based on the semantics for Dalvik from Chapter 5. The analysis is expressed as flow logic judgements [NNH99]. First, we introduce the necessary concepts and define abstract domains for the analysis.

6.1 Partial Order and Lattices

Flow logic is based on partial order and lattices. Before we define our analysis, the necessary concepts are introduced briefly. They are standard definitions discussed in [NNH99].

Partial order A *partial order* in a set S is a relation, $\sqsubseteq$, on S that is reflexive, anti-symmetric, and transitive:

- $\forall x \in S : x \sqsubseteq x$
- $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$
- $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$

A set S with a partial order $\sqsubseteq$ is called a partially ordered set and denoted $(S, \sqsubseteq)$.

Least upper bound For $X \subseteq S$, we say that $y \in S$ is an *upper bound* of X if $\forall x \in X : x \sqsubseteq y$. In addition, if $y \sqsubseteq z$ for all upper bounds z, then y

is the *least upper bound* of X, denoted $\bigsqcup X$. The binary least upper bound $\bigsqcup \{x, y\}$ is written $x \sqcup y$.

Greatest lower bound Similarly, $y \in S$ is a *lower bound* if $\forall x \in X : y \sqsubseteq x$, and the *greatest lower bound* $\bigsqcap X$ if $z \sqsubseteq y$ for all lower bounds z. The greatest lower bound $\bigsqcap \{x, y\}$ is written $x \sqcap y$.

Complete lattice A partially ordered set $(S, \sqsubseteq)$ where $S \neq \emptyset$ is a *complete lattice* if $\bigsqcup X$ and $\bigsqcap X$ exist for all $X \subseteq S$.

Powerset lattice A powerset $\mathcal{P}(S)$ over S forms a complete lattice ordered by subset inclusion where $\bigsqcup \mathcal{P}(S) = S = \top$ is called the top element, and $\bigsqcap \mathcal{P}(S) = \emptyset = \bot$ is the bottom element. Figure 6.1 illustrates a powerset lattice. We use powerset lattices to represent sets of values in abstract domains in the analysis.

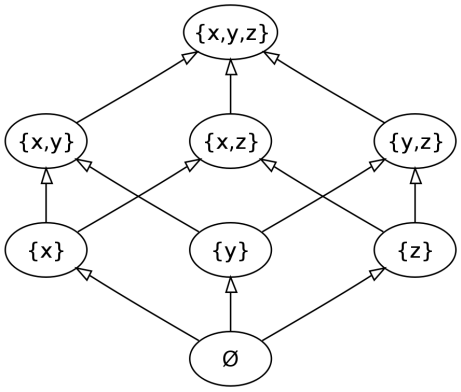


Figure 6.1: A powerset lattice over $\{x, y, z\}$ (from [KSm11]).

6.2 Abstract Domains

In this section, we define the abstract domains which are abstractions of the concrete semantic domains. They are defined in ways that are relevant for control flow analysis. The abstract domains are necessary to statically represent values that are only available at runtime.

Just like the semantic domains, the abstract domain for values will consist of primitive values, references, and classes:

$$\overline{\mathsf{Val}} = \overline{\mathsf{Prim}} + \overline{\mathsf{Ref}} + \overline{\mathsf{Class}}$$

As in [Han05], the overbar is used to distinguish abstract domains from semantic, or concrete, domains. Since the analysis is an over-approximation, we will need to be able to represent sets of possible values. The hat notation will be used to represent abstract domains that are complete lattices ordered by subset inclusion:

$$\widehat{\mathsf{Val}} = \mathcal{P}(\overline{\mathsf{Val}})$$

Primitive values are represented as integers, so there is no difference from the semantic domain:

$$\overline{\mathsf{Prim}} = \mathsf{Prim} = \mathbb{Z}$$

$$\widehat{\mathsf{Prim}} = \mathcal{P}(\overline{\mathsf{Prim}})$$

Strings and classes will be used in the same way:

$$\widehat{\overline{\mathsf{String}}} = \mathcal{P}(\overline{\mathsf{String}}) = \mathcal{P}(\mathsf{String})$$

$$\widehat{\overline{\mathsf{Class}}} = \mathcal{P}(\overline{\mathsf{Class}}) = \mathcal{P}(\mathsf{Class})$$

References can be object references or array references:

$$\overline{\mathsf{Ref}} = \overline{\mathsf{ObjRef}} + \overline{\mathsf{ArrRef}}$$

For objects, we will use the representation of modelling references as classes or `null`:

$$\overline{\mathsf{ObjRef}} = \mathsf{Class} \cup \{\texttt{null}\}$$

This means that objects of the same class cannot be distinguished. By using the Cartesian product of the class and the address of the instruction that

created the object, the analysis could become more precise, but we leave it out to keep the analysis simple.

Array references are modelled after the array type, and similarly to objects, the analysis could be improved by including the creation point of the array.

$$\overline{\mathsf{ArrRef}} = \mathsf{ArrayType} \cup \{\texttt{null}\}$$

Abstract values from these domains are written (Ref x), (ObjRef x), and (ArrRef x) for readability.

Abstract addresses will be represented as their corresponding semantic domain but with the addition of a special program counter value representing the end of control flow for methods:

$$\overline{\mathsf{Addr}} = \mathsf{Addr} + (\mathsf{Method} \times \{\mathsf{END}\})$$

The addition of this explicit value makes each method have easily referrable entry and exit points: $pc = 0$ and $pc = \mathsf{END}$, respectively.

For the abstract representation of the LocalReg domain, we could simply map $\mathsf{Register} \to \widehat{\mathsf{Val}}$, but for a more precise analysis inside methods, flow sensitivity can be added by keeping track of values in registers at all points in a method:

$$\widehat{\mathsf{LocalReg}} = \overline{\mathsf{Addr}} \to (\mathsf{Register} \cup \{\mathsf{END}\}) \to \widehat{\mathsf{Val}}$$

This means that for $\hat{R} \in \widehat{\mathsf{LocalReg}}, a \in \overline{\mathsf{Addr}}$, the value $\hat{R}(a)$ is a function mapping registers to abstract values. The token END is reused as a pseudo-register such that for $m \in \mathsf{Method}$, the expression $\hat{R}(m, \mathsf{END})$ is notation for $\hat{R}(m, \mathsf{END})(\mathsf{END})$. This is used in the judgements for method invocation instructions in Section 7.3 to pass return values to the `retval` register.

As with the semantic domains, the heap is split into the static and dynamic parts. The static heap maps fields to values:

$$\widehat{\mathsf{StaticHeap}} = \mathsf{Field} \to \widehat{\mathsf{Val}}$$

and the dynamic heap maps references to objects and arrays:

$$\widehat{\mathsf{Heap}} = \overline{\mathsf{Ref}} \to (\widehat{\mathsf{Object}} + \widehat{\mathsf{Array}})$$

The state of an object is the state of its (instance) fields:

$$\widehat{\mathsf{Object}} = \mathsf{Field} \to \widehat{\mathsf{Val}}$$

For a simple analysis, the length and structure of arrays can be ignored. Instead the array is simply represented as an unordered set of values:

$$\widehat{\mathsf{Array}} = \widehat{\mathsf{Val}}$$

6.2.1 Abstract Representation Function

For specifying the flow logic judgements of Dalvik instructions, a representation function is needed to map concrete semantic values into their corresponding abstract representations. Since our analysis represents all values as elements in sets of possible values, we can simply create a singleton set with the value:

$$\beta(c) = \{c\}$$

In the `const`, `cmp`, and `instance-of` instructions in Chapter 7, the representation function will map $\mathsf{Prim} \to \widehat{\mathsf{Prim}}$, for `const-string` it will map $\mathsf{String} \to \widehat{\mathsf{String}}$, and lastly for `const-class`, `new-instance`, `throw`, and `check-cast`, it will map $\mathsf{Class} \to \widehat{\mathsf{Class}}$.

6.3 Flow Logic Specification

The abstract representation of a semantic program configuration can now be specified from the abstract domains as the domain for the control flow analysis:

$$\widehat{\mathsf{Analysis}} = \widehat{\mathsf{StaticHeap}} \times \widehat{\mathsf{Heap}} \times \widehat{\mathsf{LocalReg}}$$

Since each component is a lattice, the $\widehat{\mathsf{Analysis}}$ domain is also a lattice with one element being smaller than another if each of the components of the first element is smaller than the corresponding component of the second element. This ordering is used to compare analysis results. When used between functions, the ordering is taken as a point-wise extension of the ordering on the codomain. For example, for $\hat{R} \in \widehat{\mathsf{LocalReg}}$ and $a_1, a_2 \in \widehat{\mathsf{Addr}}$:

$$\hat{R}(a_1) \sqsubseteq \hat{R}(a_2) \quad \text{iff} \quad \forall r \in \mathrm{dom}(\hat{R}(a_1)) : \hat{R}(a_1)(r) \sqsubseteq \hat{R}(a_2)(r)$$

For convenience we will extend the relation for functions to be able to compare all codomain values except those mapped to, from some specified set. In general, for functions F_1, F_2 and the set to exclude X:

$$F_1 \sqsubseteq_X F_2 \quad \text{iff} \quad \forall a \in \text{dom}(F_1) \setminus X : F_1(a) \sqsubseteq F_2(a)$$

The analysis specifies judgements that define when an analysis result (from the $\widehat{\mathsf{Analysis}}$ domain) is acceptable:

$$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon \texttt{instr}$$

where $(\hat{S}, \hat{H}, \hat{R}) \in \widehat{\mathsf{Analysis}}$ and `instr` is the instruction at pc in method m. In the next chapter we specify and explain the judgements for the Dalvik instructions.

Chapter 7

Flow Logic Judgements

As with the semantics, we will first treat Dalvik without exceptions and then add them in Section 7.6. The judgements in this chapter are also summarized in Appendix C.

7.1 Imperative Core

The semantics for the `nop` instruction state that its only effect is incrementing the program counter. Since this does not change the state of the registers, they are all copied into the analysis of the next program point:

$$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon \texttt{nop}$$
$$\text{iff} \quad \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc + 1)$$

The static and dynamic heaps do not depend on instruction addresses, so they are not copied each time the program counter changes. By "copy" actually we mean that the least upper bound of the old and new value is used as the new value, or, alternatively, that the union of the sets of possible values is used as the new set of possible values. A solution that satisfies this ordering and all of the following for each instruction in each method in the program will be a safe over-approximation of all possible values in every method.

After a `const` instruction, the contents of its destination register will be known at the next program point to be the abstract value of the argument. The rest of the registers are copied forward unchanged as signified

by the $\sqsubseteq_{\{v\}}$ relation:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{const}\ \ v\ \ c\\ \quad \text{iff} \quad \beta(c) \sqsubseteq \hat{R}(m,pc+1)(v)\\ \qquad\ \ \hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)\end{array}$$

The conditions of the judgements are connected by an implicit conjunction.

The instruction `const-class` works the same way:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{const-class}\ \ v\ \ cl\\ \quad \text{iff} \quad \beta(cl) \sqsubseteq \hat{R}(m,pc+1)(v)\\ \qquad\ \ \hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)\end{array}$$

The instruction `move` is like `const` except that the contents of a register is copied instead of a constant value:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{move}\ \ v_1\ \ v_2\\ \quad \text{iff} \quad \hat{R}(m,pc)(v_2) \sqsubseteq \hat{R}(m,pc+1)(v_1)\\ \qquad\ \ \hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)\end{array}$$

For a binary operation, the value to be copied forward to the destination register is determined by the binary abstract $\widehat{binOp}$ function:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{binop}_{op}\ \ v_1\ \ v_2\ \ v_3\\ \quad \text{iff} \quad \widehat{binOp}_{op}(\hat{R}(m,pc)(v_2),\hat{R}(m,pc)(v_3)) \sqsubseteq \hat{R}(m,pc+1)(v_1)\\ \qquad\ \ \hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)\end{array}$$

The simplest such function always returns $\top$ from the $\widehat{\mathsf{Prim}}$ domain, meaning all possible primitive values:

$$\widehat{binOp}_{op}(c_1,c_2) = \top_{\widehat{\mathsf{Prim}}}$$

This function can be improved for the benefit of a specific analysis. The case for binary operations with a literal value is similar:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{binop-lit}_{op}\ \ v_1\ \ v_2\ \ c\\ \quad \text{iff} \quad \widehat{binOp}_{op}(\hat{R}(m,pc)(v_2),\beta(c)) \sqsubseteq \hat{R}(m,pc+1)(v_1)\\ \qquad\ \ \hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)\end{array}$$

Unary operations are also similar:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{unop}_{op}\ \ v_1\ \ v_2\\ \quad \text{iff} \quad \widehat{unOp_{op}}(\hat{R}(m,pc)(v_2)) \sqsubseteq \hat{R}(m,pc+1)(v_1)\\ \qquad \hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)\end{array}$$

where

$$\widehat{unOp_{op}}(c) = \top_{\widehat{\mathsf{Prim}}}$$

The `goto` instruction unconditionally branches to its argument. This is done simply by copying all registers to the ones at new address:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{goto}\ \ pc'\\ \quad \text{iff} \quad \hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc')\end{array}$$

Branching with `if` is like `goto` except that there are two possibilities for the next program counter value. A simple and safe over-approximation is to assume both branches are taken, so register contents are simply copied to both program points:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{if}\ \ op\ \ v_1\ \ v_2\ \ pc'\\ \quad \text{iff} \quad \hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)\\ \qquad \hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc')\end{array}$$

The same applies for `ifz`:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{ifz}\ \ op\ \ v_1\ \ pc'\\ \quad \text{iff} \quad \hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)\\ \qquad \hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc')\end{array}$$

A more precise analysis could choose based on the sets of values in the registers and the given operator where above rules simply ignore them.

Comparison is similar to the `if` instructions, except that the result of the comparison affects the given destination register rather than the program counter. We over-approximate by storing all possible results from the comparison into the destination register:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{cmp}\ \ bias\ \ v_1\ \ v_2\ \ v_3\\ \quad \text{iff} \quad \beta(-1) \sqcup \beta(0) \sqcup \beta(1) \sqsubseteq \hat{R}(m,pc+1)(v_1)\\ \qquad \hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)\end{array}$$

7.2 Objects

When a new object is created, a reference to it is stored in the destination register of the `new-instance` instruction:

$$\begin{aligned}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{:}&\texttt{new-instance}\ \ v\ \ cl\\ \text{iff}\quad &\beta(cl) \sqsubseteq \hat{R}(m,pc+1)(v)\\ &\hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)\end{aligned}$$

Since object references are modelled as classes and since these are all known statically, there is no need to "create" a new object on the heap, and the class can be used directly as a reference.

The `const-string` instruction is simply a shortcut for creating a new `java.lang.String` instance and setting its value[1] to the given constant string. As with `new-instance`, the reference to the object is stored in the destination register:

$$\begin{aligned}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{:}&\texttt{const-string}\ \ v\ \ s\\ \text{iff}\quad &\beta(s) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ \texttt{String})(\mathit{value})\\ &(\mathsf{ObjRef}\ \texttt{String}) \sqsubseteq \hat{R}(m,pc+1)(v)\\ &\hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)\end{aligned}$$

As with `cmp`, the `instance-of` instruction simply stores the union/least upper bound of the possible results in the destination register:

$$\begin{aligned}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{:}&\texttt{instance-of}\ \ v_1\ \ v_2\ \ \mathit{type}\\ \text{iff}\quad &\beta(0) \sqcup \beta(1) \sqsubseteq \hat{R}(m,pc+1)(v_1)\\ &\hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)\end{aligned}$$

Unlike `new-instance`, the two instructions `iget` and `iput` access object *contents*, so they use the abstract dynamic heap. Since registers in the analysis contain multiple values, all of them must be used, but since registers may contain values of different types, we will use the notation ($\mathsf{ObjRef}\ cl$) to only match the object references and not the array references:

$$\begin{aligned}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{:}&\texttt{iget}\ \ v_1\ \ v_2\ \ \mathit{fld}\\ \text{iff}\quad &\forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m,pc)(v_2)\text{:}\\ &\quad \hat{H}(\mathsf{ObjRef}\ \ cl)(\mathit{fld}) \sqsubseteq \hat{R}(m,pc+1)(v_1)\\ &\hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)\end{aligned}$$

[1] In the Apache Harmony Java implementation used in Android, the String class uses the name "value" for the variable holding the character array representing the string.

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\text{: } \texttt{iput}\ \ v_1\ \ v_2\ \ \mathit{fld} \\
\quad \text{iff} \quad \forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m, pc)(v_2)\text{:} \\
\qquad\quad \hat{R}(m, pc)(v_1) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ \ cl)(\mathit{fld}) \\
\qquad \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc + 1)
\end{array}
$$

On the static heap the situation is simpler since fields are identified uniquely by name:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\text{: } \texttt{sget}\ \ v\ \ \mathit{fld} \\
\quad \text{iff} \quad \hat{S}(\mathit{fld}) \sqsubseteq \hat{R}(m, pc + 1)(v) \\
\qquad\quad \hat{R}(m, pc) \sqsubseteq_{\{v\}} \hat{R}(m, pc + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\text{: } \texttt{sput}\ \ v\ \ \mathit{fld} \\
\quad \text{iff} \quad \hat{R}(m, pc)(v) \sqsubseteq \hat{S}(\mathit{fld}) \\
\qquad\quad \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc + 1)
\end{array}
$$

7.3 Methods

The `invoke-virtual` instruction works as follows: For each possible object the method can be called on, the method is resolved (by dynamic dispatch using the *resolveMethod* function from the semantics), the arguments are transferred, and the `retval` register is updated with the return value unless the return type of the method is void:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\text{: } \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ \mathit{meth} \\
\quad \text{iff} \quad \forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m, pc)(v_1)\text{:} \\
\qquad\quad m' = \mathit{resolveMethod}(\mathit{meth}, cl) \\
\qquad\quad \forall 1 \leq i \leq n\text{: } \hat{R}(m, pc)(v_i) \sqsubseteq \hat{R}(m', 0)(m'.\mathit{maxLocal} + i) \\
\qquad\quad m'.\mathit{returnType} \neq \texttt{void} \Rightarrow \hat{R}(m', \mathsf{END}) \sqsubseteq \hat{R}(m, pc + 1)(\texttt{retval}) \\
\qquad \hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc + 1)
\end{array}
$$

As usual, all but the affected register are transferred untouched. The invoke instructions and `filled-new-array` are the only ones that set the contents of the `retval` register (until we add exceptions), and its value is only defined at instructions immediately following those. Therefore, we do not need to explicitly exclude it from being copied forward in other instructions.

There are no differences between the conditions for `invoke-virtual` and

`invoke-interface`:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{invoke-interface}\ \ v_1 \ldots v_n\ \ meth \\
\quad \text{iff} \quad \forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m,pc)(v_1)\text{:} \\
\qquad\quad m' = resolveMethod(meth, cl) \\
\qquad\quad \forall 1 \le i \le n\text{: } \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.maxLocal + i) \\
\qquad\quad m'.returnType \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)
\end{array}
$$

The judgement for direct methods is the same as for virtual methods except that the appropriate resolve function is used:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{invoke-direct}\ \ v_1 \ldots v_n\ \ meth \\
\quad \text{iff} \quad \forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m,pc)(v_1)\text{:} \\
\qquad\quad m' = resolveDirectMethod(meth, cl) \\
\qquad\quad \forall 1 \le i \le n\text{: } \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.maxLocal + i) \\
\qquad\quad m'.returnType \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)
\end{array}
$$

`invoke-super` is also the same as for virtual methods except that the superclass must be defined and that the resolving starts from the superclass of the given object:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{invoke-super}\ \ v_1 \ldots v_n\ \ meth \\
\quad \text{iff} \quad \forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m,pc)(v_1)\text{:} \\
\qquad\quad cl.super \neq \bot \\
\qquad\quad m' = resolveMethod(meth, cl.super) \\
\qquad\quad \forall 1 \le i \le n\text{: } \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.maxLocal + i) \\
\qquad\quad m'.returnType \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)
\end{array}
$$

Static methods are different since they are not invoked on objects, but otherwise the structure is the same:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{invoke-static}\ \ v_1 \ldots v_n\ \ meth \\
\quad \text{iff} \quad m' = resolveStaticMethod(meth) \\
\qquad\quad \forall 1 \le i \le n\text{: } \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.maxLocal + i) \\
\qquad\quad m'.returnType \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad\quad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)
\end{array}
$$

The judgement for methods with no arguments is even simpler since the copying of arguments is skipped:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon \texttt{invoke-static}\ \ \varepsilon\ \ meth \\
\quad \text{iff} \quad m' = \mathit{resolveStaticMethod}(meth) \\
\qquad m'.\mathit{returnType} \neq \texttt{void} \Rightarrow \hat{R}(m', \mathsf{END}) \sqsubseteq \hat{R}(m, pc+1)(\texttt{retval}) \\
\qquad \hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1)
\end{array}
$$

The `return` instruction sets the value at the END pseudo register to be read by the invoke instructions:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon \texttt{return}\ \ v \\
\quad \text{iff} \quad \hat{R}(m, pc)(v) \sqsubseteq \hat{R}(m, \mathsf{END})
\end{array}
$$

For void methods no constraints are generated by returning:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon \texttt{return-void} \\
\quad \text{iff} \quad \mathit{true}
\end{array}
$$

Finally, after a method has been invoked and its return value transferred to the `retval` register of the next instruction, the value can then be moved to a regular register:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon \texttt{move-result}\ \ v \\
\quad \text{iff} \quad \hat{R}(m, pc)(\texttt{retval}) \sqsubseteq \hat{R}(m, pc+1)(v) \\
\qquad \hat{R}(m, pc) \sqsubseteq_{\{v\}} \hat{R}(m, pc+1)
\end{array}
$$

7.4 Arrays

As with object references, array references are modelled after statically known information, so for the `new-array` instruction, the heap is not affected, but the array reference, which is simply the array type, is stored in the destination register. The array length is irrelevant since arrays are modelled as unordered sets of values:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon \texttt{new-array}\ \ v_1\ \ v_2\ \ \mathit{type} \\
\quad \text{iff} \quad (\mathsf{ArrRef}\ \mathit{type}) \sqsubseteq \hat{R}(m, pc+1)(v_1) \\
\qquad \hat{R}(m, pc) \sqsubseteq_{\{v_1\}} \hat{R}(m, pc+1)
\end{array}
$$

Since the array length is not tracked, the `array-length` instruction, like the $\widehat{binOp}$ and $\widehat{unOp}$ functions, is no more precise than $\top$ from the $\widehat{\mathsf{Prim}}$ domain:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{array-length}\ \ v_1\ \ v_2\\ \quad \text{iff} \quad \top_{\widehat{\mathsf{Prim}}} \sqsubseteq \hat{R}(m,pc+1)(v_1)\\ \qquad \hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)\end{array}$$

The array get instruction transfers each array reference to the destination register of the next instruction while ignoring the given array index:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{aget}\ \ v_1\ \ v_2\ \ v_3\\ \quad \text{iff} \quad \forall(\mathsf{ArrRef}\ \mathit{type}) \in \hat{R}(m,pc)(v_2)\colon \hat{H}(\mathsf{ArrRef}\ \mathit{type}) \sqsubseteq \hat{R}(m,pc+1)(v_1)\\ \qquad \hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)\end{array}$$

The array put instruction simply adds to the heap and does not affect the registers:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{aput}\ \ v_1\ \ v_2\ \ v_3\\ \quad \text{iff} \quad \forall(\mathsf{ArrRef}\ \mathit{type}) \in \hat{R}(m,pc)(v_2)\colon \hat{R}(m,pc)(v_1) \sqsubseteq \hat{H}(\mathsf{ArrRef}\ \mathit{type})\\ \qquad \hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)\end{array}$$

The `filled-new-array` instruction is a specialized instruction that constructs a new array of the given type and with the values from the argument registers. It stores a reference to the new array in the `retval` register as if it was a method call. Because of our simple representation of arrays, the given content arguments are simply stored in the array modelling the array reference:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{filled-new-array}\ \ v_1 \ldots v_n\ \ \mathit{type}\\ \quad \text{iff} \quad \{(\mathsf{ArrRef}\ \mathit{type})\} \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval})\\ \qquad \forall 1 \le i \le n\colon \hat{R}(m,pc)(v_i) \sqsubseteq \hat{H}(\mathsf{ArrRef}\ \mathit{type})\\ \qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)\end{array}$$

Since `filled-new-array` may be called with no arguments, thus creating a new array of length zero, we use the following judgement that is the same as the previous one except that no values are copied into the array on the heap:

$$\begin{array}{l}(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{filled-new-array}\ \ \varepsilon\ \ \mathit{type}\\ \quad \text{iff} \quad \{(\mathsf{ArrRef}\ \mathit{type})\} \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval})\\ \qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)\end{array}$$

The final array related instruction fills an existing array with data from a statically defined table referenced by a program counter value. Registers contain various values, so only array references are used, and each of the referenced arrays is filled with the data from the table looked up by the *tableAt* function:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\text{:}\texttt{fill-array-data}\ v\ \ pc' \\
\quad \text{iff} \quad \forall(\mathsf{ArrRef}\ type) \in \hat{R}(m, pc)(v)\text{:} \\
\qquad\quad d = m.tableAt(pc') \\
\qquad\quad \forall 0 \leq i \leq d.size - 1\text{:} \\
\qquad\qquad\quad d.data(i) \sqsubseteq \hat{H}(\mathsf{ArrRef}\ type) \\
\qquad\quad \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc + 1)
\end{array}
$$

7.5 Switches

The control flow for the two switch instructions follow the same structure: Look up the switch table using the *tableAt* function and, for each jump target, copy the current register values to that location and also to the next program counter for the situation where no case matches. The lookup key in the argument register is ignored.

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\text{:}\texttt{packed-switch}\ v\ \ pc' \\
\quad \text{iff} \quad s = m.tableAt(pc') \\
\qquad\quad \forall pc'' \in s.packedTargets\text{:} \\
\qquad\qquad\quad \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc'') \\
\qquad\quad \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\text{:}\texttt{sparse-switch}\ v\ \ pc' \\
\quad \text{iff} \quad s = m.tableAt(pc') \\
\qquad\quad \forall pc'' \in s.sparseTargets\text{:} \\
\qquad\qquad\quad \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc'') \\
\qquad\quad \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc + 1)
\end{array}
$$

7.6 Exception Flow Logic

In this section we show how an analysis can be defined to handle exceptions. First, we define the abstract domains necessary and then we define flow judgements for the exception semantics.

7.6.1 Abstract Exception Domains

Exceptions are objects of a subclass of `Throwable`. For clarity, we introduce an abstract domain for exception references:

$$\overline{\mathsf{ExcRef}} = \overline{\mathsf{ObjRef}}$$

Since we do not model the call stack directly in the analysis, we will also need a way to treat exceptions that cannot be handled in the method they are thrown. The exception cache tracks sets of exceptions in methods:

$$\widehat{\mathsf{ExcCache}} = \mathsf{Method} \to \mathcal{P}(\overline{\mathsf{ExcRef}})$$

When the exception cache is added to the previous analysis, we get the exception analysis domain:

$$\widehat{\mathsf{Analysis}_{\mathsf{EXC}}} = \widehat{\mathsf{StaticHeap}} \times \widehat{\mathsf{Heap}} \times \widehat{\mathsf{LocalReg}} \times \widehat{\mathsf{ExcCache}}$$

7.6.2 Exception Judgements

From the semantics in Section 5.6, we know that two things can happen when an exception is thrown: If a local handler exists, control is transferred to that handler with a reference to the exception object in the `retval` register. If no local handler exists, the method aborts and the exception is put on the call stack in an exception frame. The analysis will treat this situation with the following auxiliary predicate:

$$\begin{array}{l}\mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ cl_E),(m,pc)) \equiv \\ \quad \mathit{findHandler}(m,pc,cl_E) = pc' \Rightarrow \\ \quad\quad \beta(\mathsf{ExcRef}\ cl_E) \sqsubseteq \hat{R}(m,pc')(\texttt{retval}) \\ \quad\quad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc') \\ \quad \mathit{findHandler}(m,pc,cl_E) = \bot \Rightarrow \\ \quad\quad \beta(\mathsf{ExcRef}\ cl_E) \sqsubseteq \hat{E}(m)\end{array}$$

The predicate uses the *findHandler* function defined in the semantics to check if a handler for an exception of class cl_E that was thrown at pc in method m is available. If it is, the exception reference is stored in the `retval` register

of the program counter value of the handler (with β as previously defined) and the remaining registers are copied untouched. If not, the exception is stored in the exception cache to be handled by the invoke instructions.

We will first demonstrate HANDLE in the `throw` instruction. Each exception reference in the argument register is handled and since the instruction fails on a null reference, a `NullPointerException` is also handled:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)\text{: } \texttt{throw}\ \ v \\
\quad \text{iff} \quad \forall(\mathsf{ExcRef}\ \ cl_E) \in \hat{R}(m, pc)(v)\text{:} \\
\qquad \mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ \ cl_E), (m, pc)) \\
\qquad \mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ \texttt{NullPointerException}), (m, pc))
\end{array}
$$

The conditions for the `move-exception` instruction are identical to the ones from the `move-result` instruction:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)\text{: } \texttt{move-exception}\ \ v \\
\quad \text{iff} \quad \hat{R}(m, pc)(\texttt{retval}) \sqsubseteq \hat{R}(m, pc+1)(v) \\
\qquad \hat{R}(m, pc) \sqsubseteq_{\{v\}} \hat{R}(m, pc+1)
\end{array}
$$

7.6.3 Runtime Exceptions

The semantics of the `check-cast` instruction state that two things may happen: In case the cast is acceptable it works like `nop`, otherwise a `ClassCastException` is thrown:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)\text{: } \texttt{check-cast}\ \ v\ \ \mathit{type} \\
\quad \text{iff} \quad \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc+1) \\
\qquad \mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ \texttt{ClassCastException}), (m, pc))
\end{array}
$$

As the last instruction in the exception analysis, we present `invoke-virtual`. Compared to the non-exception aware version, the instruction now handles each unhandled exception from the exception cache of the method being called and it handles a `NullPointerException` in case the method had been called on a `null` reference:

$(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)$: `invoke-virtual` $v_1 \ldots v_n$ $meth$

iff $\forall(\mathsf{ObjRef}\ cl) \in \hat{R}(m, pc)(v_1)$:

$m' = resolveMethod(meth, cl)$

$\forall 1 \leq i \leq n$: $\hat{R}(m, pc)(v_i) \sqsubseteq \hat{R}(m', 0)(m'.maxLocal + i)$

$m'.returnType \neq \texttt{void} \Rightarrow \hat{R}(m', \mathsf{END}) \sqsubseteq \hat{R}(m, pc + 1)(\texttt{retval})$

$\forall(\mathsf{ExcRef}\ cl_E) \in \hat{E}(m') : \mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ cl_E), (m, pc))$

$\hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc + 1)$

$\mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ \texttt{NullPointerException}), (m, pc))$

Chapter 8

Future Work

In this chapter we look at the possibilities for future work in three areas: implementation of the analysis, expansion of the analysis to make it more precise, and expansion of the formalization to include concurrency and Java reflection. In addition, we look at some of the limitations of only analyzing Dalvik bytecode.

8.1 Implementation

With the operational semantic rules for Dalvik bytecode, and the control flow analysis specified in flow logic, the analysis is now ready be used on apps. However, without an automated tool to run the analysis it will be a very tedious task to do even for a small app. Instead, the flow judgements can be converted using a constraint generator into a set of constraints that can be solved by an automated solver such as Prolog [Dan12] or Datalog [Dat12]. This requires the bytecode to be available but since parts of the Android API is not written in Java, these parts should be analyzed by another method. By manually analysing often-used API methods this will additionally result in a more precise analysis. We have included the 20 most used library methods in our data set in Table 8.1, which shows that an analysis could clearly benefit from a precise manual inspection of string handling methods.

Calls	Method
1,278,534	java/lang/StringBuilder->append(String)StringBuilder
556,232	java/lang/StringBuilder->toString()String
442,197	java/lang/Object-><init>()V
354,901	java/lang/StringBuilder-><init>()V
284,448	java/lang/String->equals(Object)Z
194,952	java/lang/StringBuilder-><init>(String)V
183,849	java/lang/StringBuffer->append(String)StringBuffer
149,807	java/lang/Integer->valueOf(I)Integer
137,022	java/lang/StringBuilder->append(I)StringBuilder
107,464	java/lang/String->length()I
97,309	java/util/Iterator->hasNext()Z
96,830	java/util/Map->put(Object;Object)Object
96,785	java/util/Iterator->next()Object
85,157	java/util/HashMap->put(Object;Object)Object
81,411	java/lang/StringBuilder->append(Object)StringBuilder
72,085	android/widget/TextView->setText(CharSequence)V
67,802	java/util/List->add(Object)Z
67,123	java/lang/IllegalArgumentException-><init>(String)V
67,113	java/lang/String->valueOf(Object)String
63,289	java/util/ArrayList->add(Object)Z

Table 8.1: The 20 most used library methods in our data set.

8.2 Java Features

Due to time constraints, we have not included concurrency in our formalization of Dalvik, nor did we handle it in the control flow analysis. Our study in Section 3.4 showed that monitors are used in 88 % of all the studied apps, and further inspection showed that 90 % of the apps use the Java library for threading. These numbers indicate that an analysis should handle concurrency, possibly by expanding the formalization and analysis to include concurrency.

In Java, and thus Dalvik, it is possible to examine and modify classes, methods and fields at runtime through the Java Reflection API. The analysis we have developed will fail to follow flow through Java reflection, as it will in most cases be seen as a call to a library method for which the result is unknown. Other studies of Android apps [FCH+11] used static analysis to find overprivileged apps, and expanded their analysis to analyze reflective calls

to a depth of two method calls. Despite of this, they were not able to fully resolve 41 % of the reflective calls. They identified use of reflection in 61 % of their studied apps, while we found it in 73 % of the apps.

Reflection in Android apps is used for several things, such as accessing private or hidden methods and JSON parsing [FCH+11]. We performed manual inspections of the use of reflection in our data sample and confirmed these uses. Furthermore, our study shows that more than half of the reflection calls are in libraries. Android releases are not all backward compatible with all features. Some features are added or removed when new versions are released. Our samples indicate that this is one of the main reasons why reflection is used, such that the developer is able to release the same code for a larger set of Android versions. In addition, the Android documentation recommends use of reflection to increase backward compatibility [And11a]. The Android API has a large set of internal methods that have been hidden (removed from the public JAR library which contains the Android API) and are not guaranteed to be present. Accessing these hidden methods is one of the other common uses of reflection in Android. An expansion of our analysis should include not only formalization of semantics for reflective method calls, but could also try to identify the patterns described here.

The usage of `Runtime.exec()` in 19.53 % of the apps we have studied raises some concern, and should also be further investigated. We inspected some of these uses, and found execution of both the `su` and `logcat` programs which, if successful, give the app access to run commands as the super user on the platform or read logs from all applications, respectively.

8.3 Native Libraries

We identified the use of native libraries in 20.35 % of the apps. This means that these apps are able to run code which can not be analyzed by static analysis of Dalvik bytecode alone. Static analysis of the libraries would require expansion to analyze ARM assembly as well. We identified an increased use of the native libraries compared to previous studies. Vulnerabilities have been found in the Linux kernel used by Android, and has been exploited [Obe11]. Therefore, any expansion or implementation of our analysis should identify calls to these native libraries, and possibly analyze them as well.

Chapter 9

Conclusion

We found several studies that show the increasing problem with malicious apps and privacy problems on the Android platform. Android apps run in the Dalvik virtual machine that is a register based VM for Java.

We collected the 1,700 most popular free apps from Android Market, and through a study of these we found that: All the types of Dalvik instructions are used by the 1,700 apps, and most of them are used in all apps. Specialised Java features, such as multi-threading, reflection and execution of programs outside the VM are used by a large part of the apps. We confirmed the risk of information leakage from apps by looking at the permissions required of the apps. Android developers tend to obfuscate their code, use several hundreds of classes, and include a lot of third-party library code in their apps. A thorough analysis for Android apps should consider all of these findings and be able to analyze Dalvik bytecode as well as native ARM code.

We generalized the Dalvik instruction set and formalized it using operational semantics. We defined formal representation of apps, types and the necessary semantic domains. The semantic rules were defined without considering exceptions, but we demonstrated how they can be added. The semantic rules were based on the Dalvik documentation, inspection of the Dalvik VM source code, and manual testing.

We defined abstract domains based on powerset lattices for a control flow analysis based on the semantic rules. The analysis is a safe over-approximation of actual program behaviour defined by flow logic judgements for a simple control flow analysis that could form the basis of an analysis tool.

Bibliography

[Ali11] Alienvault Labs. Analysis of Trojan-SMS.AndroidOS.FakePlayer.a. http://labs.alienvault.com/labs/index.php/2010/analysis-of-trojan-sms-androidos-fakeplayer-a/, November 29th 2011.

[And11a] Android Developers. Backward Compatibility for Applications. http://developer.android.com/resources/articles/backward-compatibility.html, December 15th 2011.

[And11b] Android Developers. Security and Permissions. http://developer.android.com/guide/topics/security/security.html, November 29th 2011.

[And11c] Android Developers. What is Android? http://developer.android.com/guide/basics/what-is-android.html, November 29th 2011.

[And11d] Android Open Source Project. Android Security Overview. http://source.android.com/tech/security/index.html, November 29th 2011.

[And11e] Android Open Source Project. Bytecode for the Dalvik VM. http://source.android.com/tech/dalvik/dalvik-bytecode.html, December 13th 2011.

[And11f] Android Open Source Project. Downloading the Source Tree. http://source.android.com/source/downloading.html, December 14th 2011.

[Bru11] Brut.alll. android-apktool. http://code.google.com/p/android-apktool/, November 29th 2011.

[Dan12] Daniel Diaz. The GNU Prolog web site. http://www.gprolog.org, January 3rd 2012.

[Dat12] Datalog Educational System. Datalog Educational System. `http://www.fdi.ucm.es/profesor/fernan/DES/`, January 3rd 2012.

[EGC+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[EOMC11] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[Eri11] Eric Lafortune. ProGuard. `http://proguard.sourceforge.net`, December 13th 2011.

[FCH+11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[FFC+11] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[Goo11a] Google. Android Market. `https://market.android.com`, November 29th 2011.

[Goo11b] Google Inc. Android Debug Bridge — Android Developers. `http://developer.android.com/guide/developing/tools/adb.html`, December 18th 2011.

[Goo11c] Google Inc. ProGuard — Android Developers. `http://developer.android.com/guide/developing/tools/proguard.html`, December 13th 2011.

[Han05] René Rydhof Hansen. *Flow Logic for Language-Based Safety and Security*. PhD thesis, Technical University of Denmark, 2005.

[HHJ+11] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In

Proceedings of the 18th ACM conference on Computer and communications security, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[jes11] jesusfreke. smali - An assembler/disassembler for Android's dex format. `http://code.google.com/p/smali/`, November 29th 2011.

[Jon11] Jonathan Meyer. About Jasmin. `http://jasmin.sourceforge.net/about.html`, December 14th 2011.

[KSm11] KSmrq. Hasse diagram of powerset of 3. `http://commons.wikimedia.org/wiki/File:Hasse_diagram_of_powerset_of_3.svg`, December 15th 2011.

[Nie11] Nielsen Company, The. Android Market Shares Recent Acquires. `http://blog.nielsen.com/nielsenwire/?p=27418`, April 26th 2011.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[Obe11] Jon Oberhide. Don't Root Robots. `http://jon.oberheide.org/files/bsides11-dontrootrobots.pdf`, December 15th 2011.

[Rau11] Raunak. Marketplace crawler for Android Enthusiast. `http://code.google.com/p/android-marketplace-crawler/`, December 18th 2011.

[res11] research2guidance. Android Market Insights. `http://www.research2guidance.com/shop/index.php/android-market-insights-september-2011`, October 8th 2011.

[Siv04] Igor Siveroni. Operational Semantics of the Java Card Virtual Machine. *Journal of Logic and Algebraic Programming*, 58(1–2):3–25, January–March 2004.

Appendix A

Generalized Instruction Set

Opcode	Original instruction	Generalized instruction
00	nop	nop
01	move	move
02	move/from16	
03	move/16	
04	move-wide	
05	move-wide/from16	
06	move-wide/16	
07	move-object	
08	move-object/from16	
09	move-object/16	
0a	move-result	move-result
0b	move-result-wide	
0c	move-result-object	
0d	move-exception	move-exception
0e	return-void	return-void
0f	return	return
10	return-wide	
11	return-object	
12	const/4	const
13	const/16	
14	const	
15	const/high16	
16	const-wide/16	
17	const-wide/32	
18	const-wide	
19	const-wide/high16	
1a	const-string	const-string
1b	const-string/jumbo	
1c	const-class	const-class

Opcode	Original instruction	Generalized instruction
1d	monitor-enter	monitor-enter
1e	monitor-exit	monitor-exit
1f	check-cast	check-cast
20	instance-of	instance-of
21	array-length	array-length
22	new-instance	new-instance
23	new-array	new-array
24	filled-new-array	filled-new-array
25	filled-new-array/range	
26	fill-array-data	fill-array-data
27	throw	throw
28	goto	goto
29	goto/16	
2a	goto/32	
2b	packed-switch	packed-switch
2c	sparse-switch	sparse-switch
2d	cmpl-float	cmp
2e	cmpg-float	
2f	cmpl-double	
30	cmpg-double	
31	cmp-long	
32	if-eq	if
33	if-ne	
34	if-lt	
35	if-ge	
36	if-gt	
37	if-le	
38	if-eqz	ifz
39	if-nez	
3a	if-ltz	
3b	if-gez	
3c	if-gtz	
3d	if-lez	
3e..43	(unused)	
44	aget	aget
45	aget-wide	
46	aget-object	
47	aget-boolean	
48	aget-byte	
49	aget-char	
4a	aget-short	
4b	aput	aput
4c	aput-wide	
4d	aput-object	

Opcode	Original instruction	Generalized instruction
4e	aput-boolean	
4f	aput-byte	
50	aput-char	
51	aput-short	
52	iget	iget
53	iget-wide	
54	iget-object	
55	iget-boolean	
56	iget-byte	
57	iget-char	
58	iget-short	
59	iput	iput
5a	iput-wide	
5b	iput-object	
5c	iput-boolean	
5d	iput-byte	
5e	iput-char	
5f	iput-short	
60	sget	sget
61	sget-wide	
62	sget-object	
63	sget-boolean	
64	sget-byte	
65	sget-char	
66	sget-short	
67	sput	sput
68	sput-wide	
69	sput-object	
6a	sput-boolean	
6b	sput-byte	
6c	sput-char	
6d	sput-short	
6e	invoke-virtual	invoke-virtual
6f	invoke-super	invoke-super
70	invoke-direct	invoke-direct
71	invoke-static	invoke-static
72	invoke-interface	invoke-interface
73	(unused)	
74	invoke-virtual/range	invoke-virtual
75	invoke-super/range	invoke-super
76	invoke-direct/range	invoke-direct
77	invoke-static/range	invoke-static
78	invoke-interface/range	invoke-interface
79..7a	(unused)	

Opcode	Original instruction	Generalized instruction
7b	neg-int	unop
7c	not-int	
7d	neg-long	
7e	not-long	
7f	neg-float	
80	neg-double	
81	int-to-long	
82	int-to-float	
83	int-to-double	
84	long-to-int	
85	long-to-float	
86	long-to-double	
87	float-to-int	
88	float-to-long	
89	float-to-double	
8a	double-to-int	
8b	double-to-long	
8c	double-to-float	
8d	int-to-byte	
8e	int-to-char	
8f	int-to-short	
90	add-int	binop
91	sub-int	
92	mul-int	
93	div-int	
94	rem-int	
95	and-int	
96	or-int	
97	xor-int	
98	shl-int	
99	shr-int	
9a	ushr-int	
9b	add-long	
9c	sub-long	
9d	mul-long	
9e	div-long	
9f	rem-long	
a0	and-long	
a1	or-long	
a2	xor-long	
a3	shl-long	
a4	shr-long	
a5	ushr-long	
a6	add-float	

Opcode	Original instruction	Generalized instruction
a7	sub-float	
a8	mul-float	
a9	div-float	
aa	rem-float	
ab	add-double	
ac	sub-double	
ad	mul-double	
ae	div-double	
af	rem-double	
b0	add-int/2addr	binop
b1	sub-int/2addr	
b2	mul-int/2addr	
b3	div-int/2addr	
b4	rem-int/2addr	
b5	and-int/2addr	
b6	or-int/2addr	
b7	xor-int/2addr	
b8	shl-int/2addr	
b9	shr-int/2addr	
ba	ushr-int/2addr	
bb	add-long/2addr	
bc	sub-long/2addr	
bd	mul-long/2addr	
be	div-long/2addr	
bf	rem-long/2addr	
c0	and-long/2addr	
c1	or-long/2addr	
c2	xor-long/2addr	
c3	shl-long/2addr	
c4	shr-long/2addr	
c5	ushr-long/2addr	
c6	add-float/2addr	
c7	sub-float/2addr	
c8	mul-float/2addr	
c9	div-float/2addr	
ca	rem-float/2addr	
cb	add-double/2addr	
cc	sub-double/2addr	
cd	mul-double/2addr	
ce	div-double/2addr	
cf	rem-double/2addr	
d0	add-int/lit16	binop-lit
d1	rsub-int	
d2	mul-int/lit16	

Opcode	Original instruction	Generalized instruction
d3	div-int/lit16	
d4	rem-int/lit16	
d5	and-int/lit16	
d6	or-int/lit16	
d7	xor-int/lit16	
d8	add-int/lit8	
d9	rsub-int/lit8	
da	mul-int/lit8	
db	div-int/lit8	
dc	rem-int/lit8	
dd	and-int/lit8	
de	or-int/lit8	
df	xor-int/lit8	
e0	shl-int/lit8	
e1	shr-int/lit8	
e2	ushr-int/lit8	
e3..ff	(unused)	

Appendix B

Semantic Rules

$$\frac{m.instructionAt(pc) = \texttt{nop}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{const}\ \ v\ \ c}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v \mapsto c]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{const-class}\ \ v\ \ cl}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v \mapsto cl]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{move}\ \ v_1\ \ v_2}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto R(v_2)]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{move-result}\ \ v}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v \mapsto R(\texttt{retval})]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{binop}_{op}\ \ v_1\ \ v_2\ \ v_3 \qquad c = binOp_{op}(R(v_2), R(v_3))}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto c]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{binop-lit}_{op}\ \ v_1\ \ v_2\ \ c \qquad c' = binOp_{op}(R(v_2), c)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto c']\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{unop}_{op}\ \ v_1\ \ v_2 \qquad c = unOp_{op}(R(v_2))}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto c]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{goto}\ \ pc'}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc', R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{if}\ \ op\ \ v_1\ \ v_2\ \ pc' \qquad relOp_{op}(R(v_1), R(v_2))}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc', R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{if}\ \ op\ \ v_1\ \ v_2\ \ pc' \qquad \neg relOp_{op}(R(v_1), R(v_2))}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{ifz}\ \ op\ \ v\ \ pc' \qquad relOp_{op}(R(v), 0)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc', R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{ifz}\ \ op\ \ v\ \ pc' \qquad \neg relOp_{op}(R(v), 0)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{new-instance}\ \ v\ \ cl \qquad (H', loc) = newObject(H, cl)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc+1, R[v \mapsto loc]\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{const-string}\ \ v\ \ s \\ (H', loc) = newObject(H, \texttt{String}) \qquad o = H'(loc) \qquad H'[loc \mapsto o[value \mapsto s]] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc+1, R[v \mapsto loc]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{iget}\ \ v_1\ \ v_2\ \ fld \qquad R(v_2) = loc \qquad o = H(loc)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto o.fld]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{iput}\ \ v_1\ \ v_2\ \ fld \qquad R(v_2) = loc \qquad o = H(loc)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H[loc \mapsto o[fld \mapsto R(v_1)]], \langle m, pc+1, R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{sget}\ \ v\ \ fld}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v \mapsto S(fld)] :: SF\rangle\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{sput}\ \ v\ \ fld}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S[fld \mapsto R(v)], H, \langle m, pc+1, R :: SF\rangle\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{instance-of}\ \ v_1\ \ v_2\ \ type \\ loc = R(v_2) \qquad o = H(loc) \qquad c = \begin{cases} 1 & \text{if} \quad o \in \mathsf{Object} \wedge o.class \preceq type \vee \\ & \qquad o \in \mathsf{Array} \wedge o.type \preceq type \\ 0 & \text{otherwise} \end{cases} \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto c]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{cmp}\ \ bias\ \ v_1\ \ v_2\ \ v_3 \qquad c = cmp_{bias}(R(v_2), R(v_3))}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto c]\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = loc \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ n = arity(meth) \qquad m' = resolveMethod(meth, o.class) \\ R' = [0 \mapsto \bot, \ldots, m'.maxLocal \mapsto \bot, \\ m'.maxLocal + 1 \mapsto v_1, \ldots, m'.maxLocal + arity(m') \mapsto v_n] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-direct}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = loc \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ n = arity(meth) \qquad m' = resolveDirectMethod(meth, o.class) \\ R' = [0 \mapsto \bot, \ldots, m'.maxLocal \mapsto \bot, \\ m'.maxLocal + 1 \mapsto v_1, \ldots, m'.maxLocal + arity(m') \mapsto v_n] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-interface}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = loc \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ n = arity(meth) \qquad m' = resolveMethod(meth, o.class) \\ R' = [0 \mapsto \bot, \ldots, m'.maxLocal \mapsto \bot, \\ \quad m'.maxLocal + 1 \mapsto v_1, \ldots, m'.maxLocal + arity(m') \mapsto v_n] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-super}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = loc \qquad loc \neq \texttt{null} \qquad o = H(loc) \qquad o.class.super \neq \bot \\ n = arity(meth) \qquad m' = resolveMethod(meth, o.class.super) \\ R' = [0 \mapsto \bot, \ldots, m'.maxLocal \mapsto \bot, \\ \quad m'.maxLocal + 1 \mapsto v_1, \ldots, m'.maxLocal + arity(m') \mapsto v_n] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-static}\ \ v_1 \ldots v_n\ \ meth \\ n = arity(meth) \qquad m' = resolveStaticMethod(meth, meth.class) \\ R' = [0 \mapsto \bot, \ldots, m'.maxLocal \mapsto \bot, \\ \quad m'.maxLocal + 1 \mapsto v_1, \ldots, m'.maxLocal + arity(m') \mapsto v_n] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-static}\ \ \varepsilon\ \ meth \\ n = arity(meth) \qquad m' = resolveStaticMethod(meth, meth.class) \\ R' = [0 \mapsto \bot, \ldots, m'.maxLocal \mapsto \bot] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{return-void}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: \langle m', pc', R'\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', pc' + 1, R'\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{return}\ \ v}{\begin{array}{r} A \vdash \langle S, H, \langle m, pc, R\rangle :: \langle m', pc', R'\rangle :: SF\rangle \Longrightarrow \\ \langle S, H, \langle m', pc' + 1, R'[\texttt{retval} \mapsto R(v)]\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{new-array}\ \ v_1\ \ v_2\ \ type \\ type \in \mathsf{ArrayType} \qquad n = R(v_2) \geq 0 \qquad (H', loc) = newArray(H, n, type) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc + 1, R[v_1 \mapsto loc]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{array-length}\ \ v_1\ \ v_2 \qquad R(v_2) = loc \qquad a = H(loc)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc, R[v_1 \mapsto a.length]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{aget}\ \ v_1\ \ v_2\ \ v_3 \qquad R(v_2) = loc \qquad a = H(loc) \qquad i = R(v_3)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc + 1, R[v_1 \mapsto a.value(i)]\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{aput}\ \ v_1\ \ v_2\ \ v_3 \qquad R(v_2) = loc \qquad a = H(loc) \qquad i = R(v_3) \\ value' = a.value[i \mapsto R(v_1)] \qquad a' = a[value \mapsto value'] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H[loc \mapsto a'], \langle m, pc + 1, R\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{filled-new-array}\ \ v_1 \ldots v_n\ \ type \\ type \in \mathsf{ArrayTypeSingle} \qquad 1 \leq n \leq 5 \\ (H', loc) = newArray(H, n, type) \qquad a = H'(loc) \\ value' = a.value[0 \mapsto R(v_1), \ldots, n - 1 \mapsto R(v_n)] \qquad a' = a[value \mapsto value'] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H'[loc \mapsto a'], \langle m, pc + 1, R[\texttt{retval} \mapsto loc]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{filled-new-array}\ \ \varepsilon\ \ type \qquad type \in \mathsf{ArrayTypeSingle} \qquad (H', loc) = newArray(H, 0, type)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc+1, R[\texttt{retval} \mapsto loc]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{fill-array-data}\ \ v\ \ pc' \qquad R(v) = loc \qquad loc \neq \texttt{null} \qquad a = H(loc) \qquad d = m.tableAt(pc') \qquad value' = a.value[0 \mapsto d.data(0), \ldots, d.size - 1 \mapsto d.data(d.size - 1)] \qquad a' = a[value \mapsto value']}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H[loc \mapsto a'], \langle m, pc+1, R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{packed-switch}\ \ v\ \ pc' \qquad s = m.tableAt(pc') \qquad i = R(v) - s.firstKey \qquad i \in dom(s.packedTargets) \qquad pc'' = s.packedTargets(i)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc'', R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{packed-switch}\ \ v\ \ pc' \qquad s = m.tableAt(pc') \qquad i = R(v) - s.firstKey \qquad i \notin dom(s.packedTargets)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{sparse-switch}\ \ v\ \ pc' \qquad s = m.tableAt(pc') \qquad R(v) \in dom(s.sparseTargets) \qquad pc'' = s.sparseTargets(R(v))}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc'', R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{sparse-switch}\ \ v\ \ pc' \qquad s = m.tableAt(pc') \qquad R(v) \notin dom(s.sparseTargets)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{throw}\ \ v \qquad R(v) = loc_E \qquad cl = H(loc_E).class \qquad cl \preceq \texttt{Throwable} \qquad findHandler(m, pc, cl) = pc'}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc', R[\texttt{retval} \mapsto loc_E]\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{throw}\ \ v \qquad R(v) = loc_E \qquad cl = H(loc_E).class \qquad cl \preceq \texttt{Throwable} \qquad findHandler(m, pc, cl) = \bot}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle loc_E, m, pc\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{move-exception}\ \ v \qquad loc_E = R(v) \qquad H(loc_E).class \preceq \texttt{Throwable}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v \mapsto R(\texttt{retval})]\rangle :: SF\rangle}$$

$$\frac{cl = H(loc_E).class \qquad findHandler(m, pc, cl) = pc'}{A \vdash \langle S, H, \langle loc_E, m_E, pc_E\rangle :: \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc', R[\texttt{retval} \mapsto loc_E]\rangle :: SF\rangle}$$

$$\frac{cl = H(loc_E).class \qquad findHandler(m, pc, cl) = \bot}{A \vdash \langle S, H, \langle loc_E, m_E, pc_E\rangle :: \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle loc_E, m, pc\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{check-cast}\ \ v\ \ type \qquad loc = R(v) \qquad o = H(loc) \qquad (o \in \mathsf{Object} \wedge o.class \preceq type) \vee (o \in \mathsf{Array} \wedge o.type \preceq type)}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{check-cast}\ \ v\ \ type \\ loc = R(v) \qquad o = H(loc) \\ (o \in \mathsf{Object} \wedge o.class \not\preceq type) \vee (o \in \mathsf{Array} \wedge o.type \not\preceq type) \\ findHandler(m, pc, \texttt{ClassCastException}) = pc' \\ (H', loc_E) = newObject(H, \texttt{ClassCastException}) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc', R[\texttt{retval} \mapsto loc_E]\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{check-cast}\ \ v\ \ type \\ loc = R(v) \qquad o = H(loc) \\ (o \in \mathsf{Object} \wedge o.class \not\preceq type) \vee (o \in \mathsf{Array} \wedge o.type \not\preceq type) \\ findHandler(m, pc, \texttt{ClassCastException}) = \bot \\ (H', loc_E) = newObject(H, \texttt{ClassCastException}) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle loc_E, m, pc\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = \texttt{null} \qquad findHandler(m, pc, \texttt{NullPointerException}) = pc' \\ (H', loc_E) = newObject(H, \texttt{NullPointerException}) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc', R[\texttt{retval} \mapsto loc_E]\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = \texttt{null} \qquad findHandler(m, pc, \texttt{NullPointerException}) = \bot \\ (H', loc_E) = newObject(H, \texttt{NullPointerException}) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle loc_E, m, pc\rangle :: SF\rangle}$$

Appendix C

Flow Logic Judgements

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{nop}$$
$$\text{iff} \quad \hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{const}\ \ v\ \ c$$
$$\text{iff} \quad \beta(c) \sqsubseteq \hat{R}(m,pc+1)(v)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{const-class}\ \ v\ \ cl$$
$$\text{iff} \quad \beta(cl) \sqsubseteq \hat{R}(m,pc+1)(v)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{move}\ \ v_1\ \ v_2$$
$$\text{iff} \quad \hat{R}(m,pc)(v_2) \sqsubseteq \hat{R}(m,pc+1)(v_1)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{move-result}\ \ v$$
$$\text{iff} \quad \hat{R}(m,pc)(\texttt{retval}) \sqsubseteq \hat{R}(m,pc+1)(v)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{binop}_{op}\ \ v_1\ \ v_2\ \ v_3$$
$$\text{iff} \quad \widehat{binOp_{op}}(\hat{R}(m,pc)(v_2),\hat{R}(m,pc)(v_3)) \sqsubseteq \hat{R}(m,pc+1)(v_1)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{binop-lit}_{op}\ \ v_1\ \ v_2\ \ c$$
$$\text{iff} \quad \widehat{binOp_{op}}(\hat{R}(m,pc)(v_2),\beta(c)) \sqsubseteq \hat{R}(m,pc+1)(v_1)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{unop}_{op}\ \ v_1\ \ v_2$$
$$\text{iff}\quad \widehat{unOp_{op}}(\hat{R}(m,pc)(v_2)) \sqsubseteq \hat{R}(m,pc+1)(v_1)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{goto}\ \ pc'$$
$$\text{iff}\quad \hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc')$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{if}\ \ op\ \ v_1\ \ v_2\ \ pc'$$
$$\text{iff}\quad \hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)$$
$$\hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc')$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{ifz}\ \ op\ \ v_1\ \ pc'$$
$$\text{iff}\quad \hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)$$
$$\hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc')$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{cmp}\ \ bias\ \ v_1\ \ v_2\ \ v_3$$
$$\text{iff}\quad \beta(-1) \sqcup \beta(0) \sqcup \beta(1) \sqsubseteq \hat{R}(m,pc+1)(v_1)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{new-instance}\ \ v\ \ cl$$
$$\text{iff}\quad \beta(cl) \sqsubseteq \hat{R}(m,pc+1)(v)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{const-string}\ \ v\ \ s$$
$$\text{iff}\quad \beta(s) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ \texttt{String})(value)$$
$$(\mathsf{ObjRef}\ \texttt{String}) \sqsubseteq \hat{R}(m,pc+1)(v)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{instance-of}\ \ v_1\ \ v_2\ \ type$$
$$\text{iff}\quad \beta(0) \sqcup \beta(1) \sqsubseteq \hat{R}(m,pc+1)(v_1)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{iget}\ \ v_1\ \ v_2\ \ fld$$
$$\text{iff}\quad \forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m,pc)(v_2)\text{:}$$
$$\hat{H}(\mathsf{ObjRef}\ \ cl)(fld) \sqsubseteq \hat{R}(m,pc+1)(v_1)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{iput}\ \ v_1\ \ v_2\ \ fld$$
$$\text{iff}\quad \forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m,pc)(v_2)\text{:}$$
$$\hat{R}(m,pc)(v_1) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ \ cl)(fld)$$
$$\hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{sget}\ \ v\ \ \mathit{fld}$$
$$\text{iff}\quad \hat{S}(\mathit{fld}) \sqsubseteq \hat{R}(m,pc+1)(v)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{sput}\ \ v\ \ \mathit{fld}$$
$$\text{iff}\quad \hat{R}(m,pc)(v) \sqsubseteq \hat{S}(\mathit{fld})$$
$$\hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ \mathit{meth}$$
$$\text{iff}\quad \forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m,pc)(v_1)\text{:}$$
$$m' = \mathit{resolveMethod}(\mathit{meth}, cl)$$
$$\forall 1 \le i \le n\text{: } \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.\mathit{maxLocal}+i)$$
$$m'.\mathit{returnType} \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval})$$
$$\hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{invoke-direct}\ \ v_1 \ldots v_n\ \ \mathit{meth}$$
$$\text{iff}\quad \forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m,pc)(v_1)\text{:}$$
$$m' = \mathit{resolveDirectMethod}(\mathit{meth}, cl)$$
$$\forall 1 \le i \le n\text{: } \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.\mathit{maxLocal}+i)$$
$$m'.\mathit{returnType} \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval})$$
$$\hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{invoke-interface}\ \ v_1 \ldots v_n\ \ \mathit{meth}$$
$$\text{iff}\quad \forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m,pc)(v_1)\text{:}$$
$$m' = \mathit{resolveMethod}(\mathit{meth}, cl)$$
$$\forall 1 \le i \le n\text{: } \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.\mathit{maxLocal}+i)$$
$$m'.\mathit{returnType} \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval})$$
$$\hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{: } \texttt{invoke-super}\ \ v_1 \ldots v_n\ \ \mathit{meth}$$
$$\text{iff}\quad \forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m,pc)(v_1)\text{:}$$
$$cl.\mathit{super} \neq \bot$$
$$m' = \mathit{resolveMethod}(\mathit{meth}, cl.\mathit{super})$$
$$\forall 1 \le i \le n\text{: } \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.\mathit{maxLocal}+i)$$
$$m'.\mathit{returnType} \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval})$$
$$\hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)$$

$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)$: `invoke-static` $v_1 \ldots v_n$ $meth$
iff $m' = resolveStaticMethod(meth)$
$\forall 1 \leq i \leq n$: $\hat{R}(m, pc)(v_i) \sqsubseteq \hat{R}(m', 0)(m'.maxLocal + i)$
$m'.returnType \neq$ `void` $\Rightarrow \hat{R}(m', \mathsf{END}) \sqsubseteq \hat{R}(m, pc+1)(\texttt{retval})$
$\hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1)$

$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)$: `invoke-static` ε $meth$
iff $m' = resolveStaticMethod(meth)$
$m'.returnType \neq$ `void` $\Rightarrow \hat{R}(m', \mathsf{END}) \sqsubseteq \hat{R}(m, pc+1)(\texttt{retval})$
$\hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1)$

$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)$: `return` v
iff $\hat{R}(m, pc)(v) \sqsubseteq \hat{R}(m, \mathsf{END})$

$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)$: `return-void`
iff $true$

$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)$: `new-array` v_1 v_2 $type$
iff $(\mathsf{ArrRef}\ type) \sqsubseteq \hat{R}(m, pc+1)(v_1)$
$\hat{R}(m, pc) \sqsubseteq_{\{v_1\}} \hat{R}(m, pc+1)$

$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)$: `array-length` v_1 v_2
iff $\top_{\widehat{\mathsf{Prim}}} \sqsubseteq \hat{R}(m, pc+1)(v_1)$
$\hat{R}(m, pc) \sqsubseteq_{\{v_1\}} \hat{R}(m, pc+1)$

$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)$: `aget` v_1 v_2 v_3
iff $\forall(\mathsf{ArrRef}\ type) \in \hat{R}(m, pc)(v_2)$: $\hat{H}(\mathsf{ArrRef}\ type) \sqsubseteq \hat{R}(m, pc+1)(v_1)$
$\hat{R}(m, pc) \sqsubseteq_{\{v_1\}} \hat{R}(m, pc+1)$

$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)$: `aput` v_1 v_2 v_3
iff $\forall(\mathsf{ArrRef}\ type) \in \hat{R}(m, pc)(v_2)$: $\hat{R}(m, pc)(v_1) \sqsubseteq \hat{H}(\mathsf{ArrRef}\ type)$
$\hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc+1)$

$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)$: `filled-new-array` $v_1 \ldots v_n$ $type$
iff $\{(\mathsf{ArrRef}\ type)\} \sqsubseteq \hat{R}(m, pc+1)(\texttt{retval})$
$\forall 1 \leq i \leq n$: $\hat{R}(m, pc)(v_i) \sqsubseteq \hat{H}(\mathsf{ArrRef}\ type)$
$\hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1)$

$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)$: `filled-new-array` ε $type$
iff $\{(\mathsf{ArrRef}\ type)\} \sqsubseteq \hat{R}(m, pc+1)(\texttt{retval})$
$\hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1)$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{:}\ \texttt{fill-array-data}\ \ v\ \ pc'$$
$$\text{iff}\quad \forall(\mathsf{ArrRef}\ \mathit{type}) \in \hat{R}(m,pc)(v)\text{:}$$
$$d = m.tableAt(pc')$$
$$\forall 0 \le i \le d.size - 1\text{:}$$
$$d.data(i) \sqsubseteq \hat{H}(\mathsf{ArrRef}\ \mathit{type})$$
$$\hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{:}\ \texttt{packed-switch}\ \ v\ \ pc'$$
$$\text{iff}\quad s = m.tableAt(pc')$$
$$\forall pc'' \in s.packedTargets\text{:}$$
$$\hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc'')$$
$$\hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R}) \models (m,pc)\text{:}\ \texttt{sparse-switch}\ \ v\ \ pc'$$
$$\text{iff}\quad s = m.tableAt(pc')$$
$$\forall pc'' \in s.sparseTargets\text{:}$$
$$\hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc'')$$
$$\hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\text{:}\ \texttt{throw}\ \ v$$
$$\text{iff}\quad \forall(\mathsf{ExcRef}\ \mathit{cl}_E) \in \hat{R}(m,pc)(v)\text{:}$$
$$\mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ \mathit{cl}_E),(m,pc))$$
$$\mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ \texttt{NullPointerException}),(m,pc))$$

$$(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\text{:}\ \texttt{move-exception}\ \ v$$
$$\text{iff}\quad \hat{R}(m,pc)(\texttt{retval}) \sqsubseteq \hat{R}(m,pc+1)(v)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)$$

$$(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\text{:}\ \texttt{check-cast}\ \ v\ \ \mathit{type}$$
$$\text{iff}\quad \hat{R}(m,pc) \sqsubseteq \hat{R}(m,pc+1)$$
$$\mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ \texttt{ClassCastException}),(m,pc))$$

$$(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\text{:}\ \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ \mathit{meth}$$
$$\text{iff}\quad \forall(\mathsf{ObjRef}\ \mathit{cl}) \in \hat{R}(m,pc)(v_1)\text{:}$$
$$m' = resolveMethod(\mathit{meth},\mathit{cl})$$
$$\forall 1 \le i \le n\text{:}\ \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.maxLocal + i)$$
$$m'.returnType \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval})$$
$$\forall(\mathsf{ExcRef}\ \mathit{cl}_E) \in \hat{E}(m')\text{ : }\mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ \mathit{cl}_E),(m,pc))$$
$$\hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)$$
$$\mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ \texttt{NullPointerException}),(m,pc))$$