# Static Analysis of Dalvik Bytecode and Reflection in Android

**Erik Ramsgaard Wognsen and Henrik Søndberg Karlsen**
**Software Engineering, Aalborg University**
**Master's Thesis, 2012**

**Title:**
Static Analysis of Dalvik Bytecode and Reflection in Android

**Project period:**
Software Engineering
SW10, Spring 2012
Master's Thesis

**Project group:**
sw102f12

**Authors:**
Erik Ramsgaard Wognsen
Henrik Søndberg Karlsen

**Supervisors:**
René Rydhof Hansen
Mads Chr. Olesen

**Abstract:**

Malicious apps pose an important problem on Android, the world's most popular smartphone operating system. Android apps are typically written in Java and compiled to run on the register based Dalvik virtual machine. Static analysis can approximate program behaviour and this approximation can be used to find malicious behaviour, for example covert sending of expensive text messages.

We expand our original operational semantics for the Dalvik instruction set to more accurately model the Android implementation, and we update our control flow analysis with these changes and improve its precision to achieve useful results when analyzing real apps. The analysis is further expanded to include support for reflection and Javascript interfaces, two dynamic features that are used extensively in popular Android apps.

Finally, we implement a prototype of the analysis which is able to create call graphs and run on real-world apps.

**Number of printed copies:** 5

**Number of content pages:** 55

**Number of appendices:** 6

**Date of completion:** June 6th 2012

# Summary

Android is the most used operating system for smartphones and one of its popular features is the ability to install third-party applications. These apps may access the user's personal information and can cost the user money, for example by sending text messages. These abilities can be misused, and malicious apps are a relevant problem for Android. Static analysis can approximate program behaviour and this approximation can be used to find malicious behaviour in apps.

We define a static analysis for Android apps based on our previous formalization of the Dalvik instruction set that resides beneath Android apps. We expand the operational semantics to more accurately model the Android implementation, and we update our flow logic based analysis with these changes and improve its precision to achieve useful results when analyzing real apps.

Reflection is widely used in Android apps, and we expand the analysis further to include support for reflective calls. While reflection is a dynamic feature, and therefore usually not handled by static analyses, we define the analysis as a safe over-approximation that relies on constant strings specified in the app for the reflection.

We demonstrate how the analysis can be implemented as we develop a prototype in Python that generates Prolog clauses expressing the constraints imposed by the flow logic. The prototype can be used to generate call graphs that include reflective calls. Furthermore, we demonstrate how the prototype can determine if any text messages can be sent with hardcoded strings. We have tested the tool on known Android malware and on small real-world apps from Google Play.

# Contents

# Preface

Our work on formalizing and analyzing the Dalvik platform started in September 2011 on our $9^{th}$ semester at Aalborg University. It has been treated in [WK12] and parts of it have been presented at *Bytecode 2012, the Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation* in collaboration with René Rydhof Hansen and Mads Chr. Olesen [KWOH12]. Furthermore, parts of the contributions of this thesis will be submitted for the *Bytecode 2012 special issue* that is to appear in *Elsevier's Science of Computer Programming* as an extended version of [KWOH12].

As a service to the reader and to present the complete work in this thesis, we provide a summary of [WK12] and [KWOH12]. Chapters 1 and 2 borrow heavily from chapters from [WK12]. Chapter 2 also contains parts from [KWOH12] as well as new material on class transformation and Javascript interfaces. Chapters 3 and 4 sum up chapters from [WK12] but also discuss several of the many updates we have made to the material as well as our new treatment of concurrency. Section 5.1 presents our full treatment of reflection analysis in Dalvik which was begun in [KWOH12].

The work described in this thesis involves these technologies and fields:

- The Java programming language
- Bytecode
- The Android platform
- Operational semantics
- Static program analysis

Some concepts are introduced, but in general, the reader is assumed to be acquainted with these topics.

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

# Chapter 1

# Introduction

The Android operating system is the most used operating system for smartphones [Nie11]. One of its popular features is the ability to enhance the functionality of a smartphone through custom applications — apps. These apps are available through app markets such as Google Play [Goo12a]. Apps may access the user's personal information, and are often able to access the Internet, send text messages or even make phone calls. With more and more features available on the smartphone, and more and more private data, the potential profit of malicious apps rise. Examples of such malicious apps include some that steal private information from the users [EGC+10] and some that cost the users money by covertly sending overpriced text-messages [Ali11].

In this project, we focus on the problems with malicious apps for Android. We base our work on our formalization for the bytecode language run by Android's virtual machine, Dalvik, in [WK12] where operational semantics and a control flow analysis is specified, based upon a large study of the instructions and API usage in typical Android apps.

We expand the control flow analysis and the operational semantics with a formalization of some of the more dynamic features of Dalvik, namely reflection and Javascript interfaces. Furthermore, we improve the precision of the control flow analysis and implement it in a Prolog-based prototype. The prototype has been made to run on real-world Android apps, an approach which has led us to become aware of many details in the implementation, and has led to changes in the formalization to comply with these observations.

The rest of this chapter summarizes the background for this project: A short introduction to program analysis for Android including related work,

an overview of Android and how its permissions are enforced and finally an introduction to Dalvik. These sections are slightly updated versions of chapters found in [WK12].

## 1.1 Program Analysis

Program analysis can be used to determine behaviour of an application. In our case, we want to keep track of information flow within an application. This can for example be used to determine if leakage of private information can occur, and whether an application is able to misuse access to services that cost the user money. A typical example of where leakage can occur on Android 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. Several malicious apps have been found on Android Market [Aeg12], with the hidden purpose of sending text messages to premium numbers. Dynamic and static program analysis can be used to detect such leakage or malicious behaviour [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, if the goal is to determine whether leakage is possible, and not just whether it happens in a specific run, dynamic analysis requires a complete input domain for the application. 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 focus only 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 intermediate steps. It does not require execution of the application, but it is able to determine conservative approximations of the flow of control or data within the application. Using static analysis, it is possible to track the locations in the application where personal information can propagate, 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.

For protection of private data and the capabilities of the smartphone, [FJM+11] have made a background service and a tool that rewrites the Dalvik bytecode in apps to use the service instead of direct API calls. By re-routing all calls that require permissions or access personal information to their service, they are able to discover what personal information leaves the device, and to block or modify the information before it does so. The service then allows the user to set permissions that are more fine-grained than the standard Android permissions. However, this approach requires knowledge of each app to determine what permissions it should be granted by the service.

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 we base our analysis on our operational semantics for Dalvik bytecode in [WK12]. In this project, we update the semantics. In addition, a control flow analysis is needed as a basis for a detailed analysis of information flow. We expand the control flow analysis in [WK12] with additional precision and support for some dynamic features.

## 1.2 Android

Android is an operating system for mobile devices that ships with various middleware and pre-installed applications [And11c]. It is based on the Linux kernel and allows third-party developers to create custom apps that are distributed on Google Play [Goo12a] and other app markets for end-users to download. Each app runs 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. Android enforces permissions within the sandbox, such that apps can only access information they have been granted access to. The permissions are declared statically by the developer in an XML file called the Android Manifest [And11b]. They are presented at install time and if they are not accepted by the user, installation is aborted. There exists a large set of permissions, including access to the Internet, ability to read or write contact information, and to send or receive text messages.

Google Play contains paid as well as free apps, with more than 500,000 different apps as of May 2012 [Mob12]. The average Android user has 32 apps installed on their device [Cha12]. Publishing apps on the market requires a market account which can be bought for a small fee. Each app submitted to Google Play is automatically tested for known malware and analyzed for unwanted behaviour [Loc12], but we have not been able to find any details on how this process works.

## 1.3 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]. The Dalvik VM is based on a register architecture, while regular Java VMs are stack-based. Dalvik instructions use register arguments to indicate which data to work with instead of using an operand stack. The instruction set is affected by this and differs from Java bytecode where several instructions are used to explicitly move data to and from the operand stack. On the other hand, Dalvik uses some specialised instructions for accessing registers with numbers 16 and up, i.e., those that require more than 4 bits to address. We present some of the individual instructions in Chapter 3.

# Chapter 2

# Study of Android Apps

To formalize the most used and important features of the Android platform, we collected and examined 1,700 of the most popular free apps on Google Play [Goo12a] (then known as Android Market) in [WK12]. The data set consisted of the 50 most popular apps of each category on Android Market in November 2011. App sizes (`*.apk` file) ranged from 16 KB to 50 MB while the bytecode content (`classes.dex` file) ranged from 1.3 KB to 7.4 MB.

We generalized the Dalvik bytecode instruction set into 39 semantically different instructions, and the study showed that all types of instructions were used in most apps. This led us to formalize the full generalized instruction set. The generalization process itself was fairly straightforward and the result is shown in Appendix A.

The study also included an insight into the usage of special Java features and Android APIs that could affect the design of static analyses. We uncovered several of these that challenge static analysis of Android apps. Here, we extend the study of these features, using the same data set as in [WK12]. The topics in this chapter are slightly updated explanations from [KWOH12] and [WK12], except Class Transformation and Javascript Interfaces which have not been presented before.

**Threading,** as indicated by the use of monitors, corresponding to the Java `synchronized` keyword, was found in 88% of the apps. Furthermore, 90.18% of the apps include a reference to the `java/lang/Thread` library. These observations are not conclusive, but indicate that multi-threaded programming is widespread. We discuss concurrency further in Section 4.3.

**The Java method** `Runtime.exec()` is used to execute programs in a separate native process and is present in 19.53% of the apps. We manually inspected some of these uses. Most of them do not use a hardcoded string as the argument to `exec()`, but of those that do, we found execution of both the `su` and `logcat` programs which, if successful, give the app access to run programs as the super user on the platform or read logs (with private data [Wil11]) from all applications, respectively. Some apps also use the `pm install` command to install other apps at runtime.

**Class Loading** Of the studied apps 39.71% contain a reference to the class loader library, `java/lang/ClassLoader`, or a subclass (e.g. `dalvik/system/DexClassLoader`). However, only 13.1% of apps use the `loadClass()` or `defineClass()` methods to actually load or define classes at runtime. Class loading allows the loading of Dalvik executable (DEX) files and JAR files while class definition allows for programmatic definition of Java classes including from scripting languages such as Javascript. If the classes being loaded are not present, e.g., if they are downloaded from the Internet, the app cannot be analyzed statically before installation. Furthermore, if the classes being loaded are created dynamically from other languages, analyzing the use before installation would require the analysis tool to parse/analyze these languages. A simpler solution for handling the apps that use these features would be to analyze the class just before it is being loaded, on the device. However, we consider this as out of scope for this project.

**Class Transformation** allows developers to change behaviour of classes at runtime, before it is loaded by the VM. It is a Java feature, and is therefore also available in Android. The transformations allowed include adding new instructions and changing control flow. We found no apps in our data set that use this feature, and will therefore not return to this subject.

**Reflection** is used extensively in Android apps for accessing private and hidden classes, methods, and fields, for JSON and XML parsing, and for backward compatibility [FCH+11]. We confirmed these observations by manual inspection. Of the 940 apps studied in [FCH+11], 61% were found to use reflection, and using automated static analysis they were able to resolve the targets for 59% of the reflective calls.

73% of the apps in our data set use reflection. This indicates that a formalization of reflection in Dalvik is necessary to precisely analyze

most apps. Reflection resolves classes, methods, and fields from strings. When these are statically known, static analysis becomes possible. We treat this in Section 5.1.

**Javascript Interfaces** allow Javascript in a webpage embedded in an app to control that app. Android supports in-app loading of webpages, through the WebKit API [Goo12b] that provides a custom embedded web browser. This API includes the `addJavascriptInterface()` method whose purpose is to make the methods on a Java object available to Javascript code. The method is used in 39% of the apps in our data set. The interface allows webpages loaded by the app to call methods on the Java object. Previous studies have shown that advertisement and analytics libraries use this to give the third-party advertisement companies access to sensor information, such as location updates [LHD+11]. We confirmed this use through manual inspection, and furthermore discovered apps that were practically webpages, and where the Dalvik code merely loads the page and extends the browser functionality, e.g. by allowing the webpage to send text messages from the phone.

# Chapter 3

# Dalvik Semantics

In this section we describe the formalization of the Dalvik bytecode language using operational semantics. We summarize the formalization done in [WK12] and present semantic rules for a small selection of instructions and we discuss updates since that work was done.

The approach is inspired by a similar effort in formalizing the Java Card bytecode language [Siv04, Han05]. To ensure that the formalization correctly represents the informal Dalvik semantics, we based the formalization on the documentation for Dalvik [And11d], inspection of the source code for the Dalvik VM and Apache Harmony Java in Android [And11e], tests of handwritten bytecode, and experiments with disassembly of compiled Java code. We have made a number of generalizations, including an idealised program counter abstracting away the length of instructions. These simplify the semantic rules but do not make the semantics less powerful [WK12].

We have formalized the generalized instruction set, except for the instructions related to concurrency, but including exception handling. The descriptions in this chapter are updated versions of those from [WK12].

## 3.1 App Structure

To be able to formalize the semantics we will first need a formal definition of the structure of Android apps. In [WK12], we specified the full app structure, starting from the App domain. Here, we summarize the most important domains as well as the changes made since [WK12]. The complete set of domains can be found in Appendix B.

We use record notation [Siv04], which is a notation for domains with access functions. 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 $x$.

One of the central domains, Class, is specified with a class name, an app in which the class is defined, the Java package it belongs to, a superclass (where $\mathsf{Class}_\perp = \mathsf{Class} \cup \{\perp\}$ and the superclass of `java/lang/Object` is defined to be $\perp$), as well as sets of implemented methods, method declarations (for abstract classes), fields, access flags and implemented interfaces:

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

The access function *methodDeclarations* has been added since [WK12] such that the Class domain can also represent abstract classes.

In Dalvik, interfaces are represented as a special type of class that inherits from `java/lang/Object` and "implements" the interfaces that it itself extends. For simplicity, we have included it as a separate domain with names closer to their semantic purposes. Beside the methods specified in an interface (method declarations), it may also include the *implementation of* a class constructor (`clinit`) that initializes static fields. Interfaces also support multiple inheritance from other interfaces. In total:

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

The access function *clinit* has been added since [WK12], and the methods specified in an interface have been changed to method declarations, as shown above, since they are not concrete implementations.

A method signature specifies how a method can be called: The name, the class or interface where it is declared (though not necessarily implemented), and the types: A sequence of types for the arguments (a sequence $A^*$ meaning an element from the set $\{\emptyset, A, A \times A, A \times A \times A, \ldots\}$) and a return type:

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

Method declarations specify everything about a method except its implementation. They appear in interfaces and abstract classes and beside the method signature specify a kind (explained below), a set of access flags, and the checked exceptions the method can throw:

$$\begin{aligned}\mathsf{MethodDeclaration} = {} & (\mathit{methodSignature}\colon \mathsf{MethodSignature}) \times \\ & (\mathit{kind}\colon \mathsf{Kind}) \times \\ & (\mathit{accessFlags}\colon \mathcal{P}(\mathsf{AccessFlag})) \times \\ & (\mathit{exceptionTypes}\colon \mathcal{P}(\mathsf{Class}))\end{aligned}$$

The kind of a method can be `direct`, which is used for non-overridable methods, i.e., constructors and private or final methods, `static` for static methods (that are not `direct`), and `virtual` for normal, overridable methods including methods specified in interfaces.

An actual method specifies the same as a method declaration plus the implementation details: A function mapping locations in the method (program counter values) to instructions, the number of registers used for local variables, a set of exception handlers, and a function mapping locations of data tables in the bytecode to the content of these tables:

$$\begin{aligned}\mathsf{Method} = {} & (\mathit{methodDeclaration}\colon \mathsf{MethodDeclaration}) \times \\ & (\mathit{instructionAt}\colon \mathsf{PC} \to \mathsf{Instruction}) \times \\ & (\mathit{numLocals}\colon \mathbb{N}_0) \times \\ & (\mathit{handlers}\colon \mathbb{N}_0 \to \mathsf{ExcHandler}) \times \\ & (\mathit{tableAt}\colon \mathsf{PC} \to \mathsf{ArrayData} \cup \mathsf{PackedSwitch} \cup \mathsf{SparseSwitch})\end{aligned}$$

For methods, the class of the method signature specifies the class (or interface, for `clinit`) where the method is implemented.

For convenience, we use e.g. $m.name$ where $m \in \mathsf{Method}$ as a shortcut to refer to $m.methodDeclaration.methodSignature.name$.

In [WK12] we only distinguished between methods and method signatures but this was insufficient to correctly represent methods specified in abstract classes and interfaces as well as reflection of method declarations as we describe in Section 5.1.4.

The types we model in Dalvik are either reference types or primitive types, and reference types can be either references to classes or to arrays. We specify this using BNF notation:

$$\begin{aligned} \mathsf{Type} &::= \mathsf{RefType} \mid \mathsf{PrimType} \\ \mathsf{RefType} &::= \mathsf{Class} \mid \mathsf{ArrayType} \end{aligned}$$

The full type hierarchy is specified in Appendix B with the update made since [WK12] that a reference can not be an interface. Furthermore, the notion of subtyping between classes, interfaces, and array types is formalized as the subclass relation, $\preceq$, which we will not discuss further here.

The Instruction domain which the *instructionAt* function of the Method domain maps to represents the set of all the generalized Dalvik instructions. For example, an instruction is `const` $v$ $c$, where $v$ and $c$ belong to the semantic domains Register and Prim, respectively.

## 3.2 Semantic Domains

Here follows a summary of the semantic domains from [WK12]. Values in our representation of Dalvik programs are either primitive values or heap references: $\mathsf{Val} = \mathsf{Prim} + \mathsf{Ref}$. Compared to the version in [WK12], we have removed Class as a component domain to correctly reflect the way, we have found the `const-class` instruction to work ("class references" are references to appropriate `java/lang/Class` instances as we discuss in Section 3.4.3). The details of primitive values will not be relevant so they can simply be represented as integers: $\mathsf{Prim} = \mathbb{Z}$. References are abstract locations or the null reference. In Dalvik, 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.

The Dalvik VM uses registers for computation and storage of local variables. Registers contain values, or $\bot$ in the case of undefined register contents: $\mathsf{LocalReg} = \mathsf{Register} \to \mathsf{Val}_\bot$. Dalvik has $2^{16}$ numbered regular registers (which we will generalize to $\mathbb{N}_0$) and a special register for holding return values: $\mathsf{Register} = \mathbb{N}_0 \cup \{\texttt{retval}\}$.

For storing objects, arrays and static fields, Dalvik uses the heap. To simplify the representation, we use a static heap $S \in \mathsf{StaticHeap}$ which maps fields to values while the normal (dynamic) heap maps references to objects or arrays. Objects have a class and a mapping of fields to values while arrays have a type, a size and a mapping of indices to values:

$$\begin{aligned}
\mathsf{StaticHeap} &= \mathsf{Field} \to \mathsf{Val} \\
\mathsf{Heap} &= \mathsf{Ref} \to (\mathsf{Object} + \mathsf{Array}) \\
\mathsf{Object} &= (\mathit{class}\colon \mathsf{Class}) \times (\mathit{field}\colon \mathsf{Field} \to \mathsf{Val}) \\
\mathsf{Array} &= (\mathit{type}\colon \mathsf{ArrayType}) \times (\mathit{length}\colon \mathbb{N}_0) \times (\mathit{value}\colon \mathbb{N}_0 \to \mathsf{Val})
\end{aligned}$$

## 3.3 Program Configurations

A program counter value paired with a method gives an absolute address of an instruction in an app: $\mathsf{Addr} = \mathsf{Method} \times \mathsf{PC}$, where program counters are integer indices of instructions: $\mathsf{PC} = \mathbb{N}_0$. We can then define stack frames to contain a method and a program counter, i.e., an address, and the local registers: $\mathsf{Frame} = \mathsf{Method} \times \mathsf{PC} \times \mathsf{LocalReg}$.

This leads to the following definition of call stacks as a sequence of frames except that the top frame may be an exception frame representing an as yet unhandled exception: $\mathsf{CallStack} = (\mathsf{Frame} + \mathsf{ExcFrame}) \times \mathsf{Frame}^*$. An exception frame contains the location of its corresponding exception object on the heap and the address of the instruction that threw the exception: $\mathsf{ExcFrame} = \mathsf{Location} \times \mathsf{Method} \times \mathsf{PC}$. When referring to a call stack, we use the notation $\langle m, pc, R\rangle :: SF$, where $\langle m, pc, R\rangle$ represents the top stack frame (in this case a non-exception frame), the operator :: appends a stack frame to a call stack, and $SF$ represents the (possibly empty) rest of the stack.

The configuration that we base our semantic rules on consists of the heaps and a call stack: $\mathsf{Configuration} = \mathsf{StaticHeap} \times \mathsf{Heap} \times \mathsf{CallStack}$. The semantic rules are reductions 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 $SF, SF' \in \mathsf{CallStack}$.

Unlike normal Java programs, Android apps have no `main()` method but are a collection of classes, some of which have methods that may be called by the Android system. These include constructors, `onStart()` methods for Activities, Services, etc., and `onClick()` methods for GUI elements, and they are discussed in Section 6.6.3. The only requirement for an initial configuration is that static fields are initialized in $S$. Also, since Android apps are not generally expected to terminate, we have no termination state.

## 3.4 Examples of Semantic Rules

With the domains and the notation defined, we are now ready to define actual semantic rules. We present a selection of interesting instructions and highlight some of the changes since [WK12]. For the full set of semantic rules, see Appendix C.

### 3.4.1 Imperative Core

A simple instruction is the `move` instruction that copies content from one register to another. We use $m.instructionAt(pc)$, where $m \in \mathsf{Method}$, $pc \in \mathsf{PC}$, to identify the instruction we are working with. The `move` instruction updates the configuration with an incremented program counter in order to move to the next instruction, and updates the register valuation such that the destination register is mapped to the content of the source register:

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

In general, the first argument of an instruction is the destination register when one is relevant.

For conditional branching we use the function $relOp_{op}(c_1, c_2) = c_1\ op\ c_2$ where $op \in \mathsf{RelOp} = \{\texttt{eq}, \texttt{ne}, \texttt{lt}, \texttt{le}, \texttt{gt}, \texttt{ge}\}$. The result of the method is either true or false, depending of the relation between the two variables, and the implementation is trivial. The auxiliary function is used in our two rules for the `if` instruction:

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

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

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.

### 3.4.2 Method Invocation

We now look at the main instruction for invoking methods with *dynamic dispatch* where the implementation of a method is looked up at runtime in the Java class hierarchy. The resolution uses the notation $meth \triangleleft m$ to indicate that a method signature $meth \in \mathsf{MethodSignature}$ is compatible with a given method $m \in \mathsf{Method}$ when their names, argument types and return type are equal. To resolve the actual method that has to be called, we use the following function to search through the class hierarchy:

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

In [WK12], this was a set of functions that also matched the method kind depending on the type of `invoke` instruction it was used in, but that was not necessary since there cannot exist two methods in the same inheritance chain that differ only in kind.

The `invoke-virtual` 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 method is resolved using *resolveMethod* and 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 $m'.numLocals$ registers are mapped to $\bot_{\mathsf{Val}}$ such that they are initially undefined, and the arguments are then mapped into the following registers:

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

In [WK12], the *arity* function simply mapped a method signature to the length of the *argTypes* domain sequence. We have changed this to a function

that maps to the number of 32 bit register required for the arguments. This is the same as the number of arguments except that `long` and `double` primitive values take up two registers. This correctly reflects the way wide data type arguments are transferred. Also, we use the *numLocals* access function as the number of 32 bit registers used for local variables as it is encoded in Dalvik instead of the *maxLocal* function in [WK12].

If the object reference for `invoke-virtual` is null, a `NullPointerException` is pushed on the call stack:

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

Standard Java exceptions belong to the `java/lang` package but we leave this out for brevity. The rules for finding the relevant exception handler are specified in [WK12] and can also be seen in Appendix C. In the event that method resolution fails, another runtime exception is thrown:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1 \dots v_n\ \ meth \\ R(v_1) = loc \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ n = arity(meth) \qquad resolveMethod(meth, o.class) = \bot \\ (H', loc_e) = newObject(H, \texttt{NoSuchMethodError}) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle loc_e, m, pc\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

This particular situation only occurs if the app has been manipulated since compilation or at runtime. In general, many instructions have runtime exceptions but since they take up space and are trivial to add, we leave them out of the semantics here.

The other `invoke` instructions are similar. Differences include that `invoke-static` resolves from the class specified in the method signature because there is no object with a runtime class, and that `invoke-direct` does not resolve through the class hierarchy because direct methods are implemented in the class that the method is invoked on.

### 3.4.3 Instantiation

Finally, we turn focus to the part of instructions concerned with objects. To allocate objects on the heap, we use an auxiliary function:

$$\begin{array}{l} newObject\colon \mathsf{Heap} \times \mathsf{Class} \to \mathsf{Heap} \times \mathsf{Ref} \\ newObject(H, cl) = (H', loc) \\ \text{where } loc \notin \mathrm{dom}(H)\ ,\ H' = H[loc \mapsto o]\ ,\ o \in \mathsf{Object}\ ,\ o.class = cl \end{array}$$

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:

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

The `const-string` instruction is a specialized instruction that creates a `java/lang/String` instance with a value from a DEX file constant pool and maps the destination register to the new reference. In our representation, the constants are inlined for convenience such that the string can simply be referred to as $s$:

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

Here, the fully qualified class name is used as a shorthand for the actual semantic `String` class, and `value` is the name of the character array representing the string in the Apache Harmony Java implementation used in Android.

Another specialized instruction is `const-class` which we have updated since [WK12]. Instead of using classes as a separate type of reference, using regular object references to `java/lang/Class` instances is closer to the Dalvik implementation and will be necessary for our reflection analysis in Section 5.1. A `java/lang/Class` object has a field `name` which refers to a `java/lang/String` with the name of the class. The instruction creates a `java/lang/Class` and a `java/lang/String` instance and sets the `java/lang/String` field `value` to the name of the given class and a maps the field `name` on the `java/lang/Class` to the newly created string:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{const-class}\ v\ \ cl \\ (H', loc_c) = newObject(H, \texttt{java/lang/Class}) \\ (H'', loc_s) = newObject(H', \texttt{java/lang/String}) \\ o_c = H''(loc_c) \qquad o'_c = o_c[field \mapsto o_c.field[\texttt{name} \mapsto loc_s]] \\ o_s = H''(loc_s) \qquad o'_s = o_s[field \mapsto o_s.field[\texttt{value} \mapsto cl.name]] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H''[loc_c \mapsto o'_c, loc_s \mapsto o'_s], \langle m, pc+1, R[v \mapsto loc]\rangle :: SF\rangle}$$

We make no effort to formalize or verify the integrity of the bytecode, such as jump destinations and instruction widths. These are verified by the Dalvik bytecode verifier before the bytecode is executed. Furthermore, we have only included sanity checks, such as checks for `null` dereferences and subtypes,

in some of the semantic rules, and deliberately left them out for many of the rules as the essence of the rules would otherwise be clouded by error handling.

# Chapter 4

# Analysis

In this chapter we define the control flow analysis for Dalvik that we use in our prototype analysis tool described in Chapter 6. The control flow analysis is based on the semantics from Chapter 3 and the one defined in [WK12]. The analysis is expressed as flow logic judgements [NNH99] which is based on partial orders and lattices, for details refer to [WK12]. First, we present the abstract domains used in the control flow analysis and next we present flow logic judgements for a selection of instructions. The abstract domains are collected in Appendix D and the full set of flow logic judgements in Appendix E.

## 4.1 Abstract Domains

The static analysis cannot determine all the runtime values, and therefore it is necessary to represent some of these values as part of abstract domains. The abstract domains are abstractions of the concrete semantic domains presented in Section 3.2.

The abstract domain for values will consist of primitive values, references, and the `null` reference: $\overline{\mathsf{Val}} = \overline{\mathsf{Prim}} + \overline{\mathsf{Ref}} + \{\texttt{null}\}$. As with the semantic domain $\mathsf{Ref}$, $\overline{\mathsf{Ref}}$ is updated since [WK12] to no longer contain $\overline{\mathsf{Class}}$. The semantic $\mathsf{Ref}$ also contains `null`, but due to the way we model abstract references, it is now placed directly in $\overline{\mathsf{Val}}$.

The overbar distinguishes the abstract domains from the semantic domains. The analysis is an over-approximation, and we use a $\widehat{\text{hat}}$ to represent sets of

values: $\widehat{\mathsf{Val}} = \mathcal{P}(\overline{\mathsf{Val}})$. These abstract domains are complete lattices ordered by subset inclusion.

A reference can refer to either an object or an array: $\overline{\mathsf{Ref}} = \overline{\mathsf{ObjRef}} + \overline{\mathsf{ArrRef}}$. The analysis specified in [WK12] represented object references by a single reference for each class. To analyze reflection with a certain precision, as we discuss in Section 5.1, we expand the analysis such that all references include a creation point consisting of the method and program counter where the corresponding object was created: $\overline{\mathsf{ObjRef}} = \mathsf{Class} \times \mathsf{Method} \times \mathsf{PC}$. The same applies for arrays which were initially only represented by the array type: $\overline{\mathsf{ArrRef}} = \mathsf{ArrayType} \times \mathsf{Method} \times \mathsf{PC}$. This representation is known as textual object graphs [VHU92].

For readability, we provide an abstract domain for exception references: $\overline{\mathsf{ExcRef}} = \overline{\mathsf{ObjRef}}$. Values from the above domains are written $(\mathsf{ObjRef}\ (x, m, pc))$, $(\mathsf{ArrRef}\ (x, m, pc))$, and $(\mathsf{ExcRef}\ (x, m, pc))$.

As in the semantic domains, primitive values are represented as integers: $\widehat{\overline{\mathsf{Prim}}} = \mathcal{P}(\overline{\mathsf{Prim}}) = \mathcal{P}(\mathsf{Prim}) = \mathcal{P}(\mathbb{Z})$.

Addresses are represented as in the semantic domains, with the addition of a special program counter value to represent the end of control flow for methods: $\overline{\mathsf{Addr}} = \mathsf{Addr} + (\mathsf{Method} \times \{\mathsf{END}\})$. The entry and exit points of a method are then $pc = 0$ and $pc = \mathsf{END}$, respectively.

We use $\hat{R}(a)$, where $\hat{R} \in \widehat{\mathsf{LocalReg}}$ and $a \in \overline{\mathsf{Addr}}$, as a function mapping registers to abstract values: $\widehat{\mathsf{LocalReg}} = \overline{\mathsf{Addr}} \to (\mathsf{Register} \cup \{\mathsf{END}\}) \to \widehat{\mathsf{Val}}$. The use of $\overline{\mathsf{Addr}}$ means that the analysis is flow-sensitive within methods. To pass return values to the `retval` register we use the pseudo-register $\mathsf{END}$ such that for $m \in \mathsf{Method}$, the expression $\hat{R}(m, \mathsf{END})$ is notation for $\hat{R}(m, \mathsf{END})(\mathsf{END})$.

In the analysis, judgements specify the constraints that the presence of an instruction in a given location in an app impose on the analysis result using the relation $\sqsubseteq$, as we demonstrate in Section 4.2. For $\widehat{\mathsf{LocalReg}}$, we use the notation $\hat{R}(a_1) \sqsubseteq \hat{R}(a_2)$ to specify that we copy all register values from one address to another, i.e., as a short-hand for:

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

This means that the least upper bound of the old and new value is used as the new value, and equivalently, since we are working with sets, that the union of the sets of possible values is used as the new set of possible values

in the registers. Between any two access functions, we subscript the relation with a set of input values to exclude from the comparison:

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

The heap is separated into the static and dynamic heap as in the semantics where the static heap maps fields to values: $\widehat{\mathsf{StaticHeap}} = \mathsf{Field} \to \widehat{\mathsf{Val}}$, and the dynamic heap maps references — now with textual object graph representation — 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: $\widehat{\mathsf{Object}} = \mathsf{Field} \to \widehat{\mathsf{Val}}$. Array values are kept as an unordered set, thus we ignore the length and structure: $\widehat{\mathsf{Array}} = \widehat{\mathsf{Val}}$. Unlike in the semantics, the class and array type is not necessary in these domains since they are already present in the references.

We use an exception cache to track exceptions that are not handled locally in a method, such that the previous method in the call stack can try to handle it: $\widehat{\mathsf{ExcCache}} = \mathsf{Method} \to \mathcal{P}(\overline{\mathsf{ExcRef}})$.

In the flow logic judgements we use an abstract representation function to map concrete semantic values into their corresponding abstract representations. The function maps the value to a singleton set: $\beta(c) = \{c\}$.

Finally, the domain for the control flow analysis consists of the above domains as follows:

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

An analysis result (from this domain) is acceptable when it respects the flow logic judgements. The result $\top_{\widehat{\mathsf{Analysis}}}$ is always acceptable but not useful, so the interesting solutions are the ones that are as precise as possible.

## 4.2 Examples of Flow Logic Judgements

In this section we briefly summarize a selection of judgements from [WK12] and present updated judgements for those that now use the textual object graph references. For the full set of judgements, see Appendix E.

### 4.2.1 Imperative Core

The semantics for the `move` instruction state that the content of a source register is copied to a destination register. In the analysis, this is expressed by the content of the source register being available in the destination register at the next program point:

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

The destination register is the only one being changed, and thus the content of all other registers is copied into the next program point without any changes. There are no changes on the static or dynamic heap, or the exception cache since these do not depend on the instruction address.

A safe over-approximation for conditional branching is to branch to all possible destinations. For the `if` instruction, this means that we assume both branches are taken. In the analysis, the content of all registers is transferred to both the next program point and the program point $pc'$ that the `if` instruction would jump to:

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

### 4.2.2 Method Invocation

Method invocation in Dalvik uses one of the `invoke` instructions. The flow logic judgement for `invoke-virtual` specifies that: For each object reference in the first argument register (the object, the method is invoked on), a method matching the signature argument, *meth*, from the instruction must be resolved using dynamic dispatch from the class in the reference. All arguments for the invoked method, i.e., the content of registers $v_1$ to $v_n$, must be present at program counter 0 in the invoked method. If the invoked method returns a value, this value must also be present in the `retval` register at the next program point in the invoking method. The invoking method also tries to handle any unhandled exception from the invoked method using the HANDLE predicate as discussed below. Finally, as an example of a runtime exception, a `NullPointerException` is tried:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ meth \\
\quad \text{iff}\ \ \forall(\mathsf{ObjRef}\ (cl,m_t,pc_t)) \in \hat{R}(m,pc)(v_1)\colon \\
\qquad m' = \mathit{resolveMethod}(meth,cl) \\
\qquad \forall 1 \le i \le n\colon \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.\mathit{numLocals}-1+i) \\
\qquad m'.\mathit{returnType} \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad \forall(\mathsf{ExcRef}\ (cl_e,m_e,pc_e)) \in \hat{E}(m')\colon \\
\qquad\quad \mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ (cl_e,m_e,pc_e)),(m,pc)) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1) \\
\qquad \mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ (\texttt{NullPointerException},m,pc)),(m,pc))
\end{array}
$$

Since we take all branches in the analysis, the instruction succeeds while simultaneously throwing exceptions. We use the following auxiliary predicate in the flow logic judgements when an exception is thrown:

$$
\begin{array}{l}
\mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ (cl_e,m_e,pc_e)),(m,pc)) \equiv \\
\quad \mathit{findHandler}(m,pc,cl_e) = pc' \neq \bot \Rightarrow \\
\qquad \{\mathsf{ExcRef}\ (cl_e,m_e,pc_e)\} \subseteq \hat{R}(m,pc')(\texttt{retval}) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc') \\
\quad \mathit{findHandler}(m,pc,cl_e) = \bot \Rightarrow \\
\qquad \{\mathsf{ExcRef}\ (cl_e,m_e,pc_e)\} \subseteq \hat{E}(m)
\end{array}
$$

The predicate *findHandler* is used to determine if a local handler is present in the method. If there is a local handler, a reference to the exception is put into the `retval` register at the program point for the found handler. If no local handler is found, the exception cache is used to store the exception reference such that the previous method in the call stack may try to handle it as shown in the `invoke-virtual` judgement.

### 4.2.3 Instantiation

In [WK12], object references were simply classes, and the `new-instance` instruction was specified as:

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

To improve the precision of the analysis, we use the textual object graph representation in the updated $\overline{\mathsf{ObjRef}}$ domain such that the address of the `new-instance` instruction is now part of the object reference:

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

Semantically, the instruction allocates room for the object and updates the destination register with its reference. In the analysis we use the statically known information about its creation point (method and program counter) to identify the new object, and therefore have no need to update the heap until any information is actually put there.

As explained in Chapter 3, the `const-string` instruction is a specialized instruction that creates a `java/lang/String` instance. We model this in the analysis, and update the heap using the creation point, class and field with the abstract representation of the string:

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

A reference to the updated location on the heap is put into the destination register, and all other registers are transferred unchanged into the next program point.

The instruction `const-class` is similar, as it creates a `java/lang/String` and a `java/lang/Class` reference. The fields `value` and `name` are updated with the name of the given class and the reference to the string, respectively:

$$
\begin{aligned}
&(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc): \texttt{const-class}\ v\ \ cl \\
&\quad \text{iff} \quad \beta(cl.name) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ \texttt{java/lang/String},m,pc))(\texttt{value}) \\
&\qquad\quad \{\mathsf{ObjRef}\ (\texttt{java/lang/String},m,pc)\} \subseteq \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/Class},m,pc))(\texttt{name}) \\
&\qquad\quad \{\mathsf{ObjRef}\ (\texttt{java/lang/Class},m,pc)\} \subseteq \hat{R}(m,pc+1)(v) \\
&\qquad\quad \hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)
\end{aligned}
$$

## 4.3 Concurrency

There are two Dalvik instructions related to concurrency: `monitor-enter` and `monitor-exit`, and as described in Chapter 2 they are used in most apps. The instructions are generated by the compiler when the Java keyword `synchronized` is used. Threads are started using Java API calls, and

`volatile` is set as an AccessFlag on fields that are declared with the Java keyword `volatile`[1]. For Java the execution order of instructions is defined by the Java Memory Model, JSR-133 [Cor12]. The memory model formalizes how shared variables should be read and written, and how instructions can be re-ordered to execute as-if-serially when they are concurrent [MG04]. According to unofficial statements [Kry12], Dalvik tries to comply with the JSR-133 memory model, though there should be cases where it does not on versions prior to Android 3.0.

However, with the analysis specified in this project, we conjecture that the analysis is sound even for multi-threaded apps. The details of the memory model are irrelevant since all possible values are present on the heap (which is the only thing shared between threads) at all points in the program, regardless of instruction and method order. If the analysis was expanded to support the relevant Dalvik instructions, Java API methods and the `volatile` access flag, it might however be possible to improve the precision of the analysis.

[1] A `volatile` Java field should never be cached as the variable is meant to be modified by more than one thread.

# Chapter 5

# Dynamic Dalvik

The study described in Chapter 2 uncovered two commonly used dynamic features in Android: Reflection and Javascript interfaces. In this chapter we describe how these features are used in Android apps and show how they can be handled by a static analysis by specifying operational semantics and expanding the control flow analysis to include the central parts of the Java reflection API and Javascript interfaces.

## 5.1 Reflection

Reflection allows a program to access class information at runtime, and use this information to create new objects, invoke methods or otherwise change the control flow of the program. When reflection is used, the types involved are usually not known statically. Instead, they are retrieved dynamically from strings. The strings can come from sources such as user input, files included with the app, the Internet, or, in some cases, constant strings in the program. We found that several of the apps in our data set specify constant strings in the program.

The most used method from the Java reflection API is `Method.invoke()`. It is an instance method on the `Method` class used to invoke dynamically resolved methods. An example can be seen in Listing 5.1 where the method `bar(int)` on the class `pkg.examples.Foo` is invoked on an instance of the class with the argument 3.

```
1 Class<?> clazz = Class.forName("pkg.examples.Foo");
2 Method method = clazz.getMethod("bar", int.class);
3 Integer result = (Integer) method.invoke(clazz.newInstance(), 3);
```

Listing 5.1: A method invoked through reflection in Java.

A `Method` object can be retrieved using the instance method `getMethod()` on the Java standard class `Class`. The instance of `Class` does not have to represent a class that implements the method since it is resolved with dynamic dispatch like normal method calls. In Listing 5.1, the `Class` object is retrieved using the static method `Class.forName()` that, given a fully qualified class name, returns a reference to a `Class` instance for the specified class.

Another way to obtain a `Class` object is through the Dalvik instruction `const-class`. It is generated when the static field `class` which is found on all Java classes is accessed. An alternative to `Class.forName("pkg.examples.Foo")` at line 1 in Listing 5.1 would therefore be to use `Foo.class`, presuming that the example code is located in the same package as the `Foo` class and that the class can be found by the Java compiler.

The `Method` objects are mainly retrieved using the methods `getMethod()` and `getMethods()`. The latter returns an array of all public method declarations on a class while the former returns a single object that is found by specifying the name and parameter types of the desired method declaration. The methods `getMethod()` and `getMethods()` only find public method declarations and they both find the method declaration objects by traversing through the class hierarchy, starting at the class represented by the `Class` object and searching upwards through superclasses and interfaces. Developers can also use the `getDeclaredMethod()` and `getDeclaredMethods()` methods which only look in the specified class but also return private methods.

Once a `Method` object has been obtained, it can be used to retrieve information about the method declaration, for example access modifiers, name, and the checked exceptions it can throw. Accessing these requires no additional information beside the information that is known statically from the method declaration. To invoke the method, an instance of the class or subclass hereof is required, except for static methods. The receiver object can be any Java object created using the regular Java `new` statement or through the `newInstance()` method on a `Class` object. Beside creating a new instance, the `newInstance()` method calls the parameterless constructor for the class.

To use another constructor, an instance of the `Constructor` class from the reflection API must be used.

### 5.1.1 Usage Patterns

The use-cases for reflection vary from app to app, and Android developers use it for many different things. However, we have observed some patterns in usage, most of which we have found through manual inspection of the bytecode.

**Hidden API methods** are invoked. Certain features in Android is deliberately hidden by the Android developers, such that they are not present in the JAR file for the Android API that app developers use when compiling their Android apps. An example of this is found in the Bluetooth features on Android, most of which were hidden in the early releases of Android due to lack of support on some devices. Developers tend to use these features anyway, and use reflection to do so instead of precompiling their own JAR file for the Android API.

**Private API methods and fields** are accessed by bypassing access modifiers. Several features of the Android platform are placed in private methods and fields, such as the ability to create a list of text messages from raw SMS data.

**Backward-compatibility** as new versions of Android are often released with new features, developers tend to use reflection to check if certain methods/features exist and call these only when they do. This pattern is even encouraged by Google in the Android documentation [And11a].

**JSON and XML** is generated and parsed with the use of reflection. Some apps use JSON and XML that contain information about their Java objects, and through reflection generation and parsing can be automated.

**Libraries** for Android apps are widely available on the Internet, and some of these use reflection. In many apps that use reflection, it is only used by the included libraries.

An analysis of reflection in standard Java has been done in [LWL05] where they found some of the same patterns in large open source projects from SourceForge [Sou12], such as object serialization and portability/backward-compatibility. However, they found that reflection was mostly used to create new objects without invoking new methods on them. We found that in

Android, invoking methods is among the most common uses of reflection. Our findings do however comply with the ones described in [FCH+11] for Android.

### 5.1.2 Assumptions

Static analysis of the reflection API is not possible in all cases, e.g. if a class being used is not known. Therefore, we presume the following requirements are met:

- All classes used through reflection are known statically, such that its components can be analyzed. In other words, we assume that dynamic class loading is not used.
- The program does not use a non-default class loader, as this could change the behaviour of `Class.forName()` and similar methods. As mentioned in Chapter 2, class loaders is used in 13.1% of the apps to either load or define new classes. This raised other problems with regards to static analysis, and they were considered out of scope for this project.
- The strings used to obtain `Method` and `Class` objects used for reflection can be determined statically. This presumption only holds for some of the studied apps. A preliminary number from [KWOH12] showed that 18.9% of the studied apps only use locally defined constant strings for the methods `Class.forName()` and `Class.getMethod()`.

This last result was based on an intra-procedural analysis of `const-string` instructions, not the (limited) data flow capabilities of our control flow analysis. But even with the inter-procedural analysis, improving the number requires the ability to track strings across collection APIs such as `java/util/ArrayList` and follow string manipulation such as that of the `java/lang/StringBuilder` class. For the latter, existing string analyses [CMS03, KGG+09, SAH+] may prove useful.

The operational semantics specified so far all represent single Dalvik instructions. We now change focus and specify operational semantics and flow logic judgements to represent Java API method calls. One way to do this would be to specify the relevant Dalvik instructions that each API method consists of. However, our goal is not to specify the semantics precisely, but to use the semantics as a means to specify an analysis that is able to handle the reflection calls. Therefore, we have chosen to specify the operational semantics for the

API methods as if they each were single, although advanced, Dalvik instructions. We try to keep the semantics and analysis as close to the specification and implementation as possible, but have left out some details when they are not necessary for the analysis. An example of this is exceptions: Most of the reflection API calls can throw exceptions, but we only describe these in rare cases. Furthermore, when we create objects, we only specify the fields necessary for the analysis, and do not guarantee that the implementation does not use more fields to track information.

The operational semantics and flow logic judgements can also be found in Appendix F.

### 5.1.3 Class Objects

When the Java method `Class.forName(string)` is used, it generates the Dalvik instruction `invoke-static` with the signature

`Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class;`

but for readability in the semantics and judgements, we identify such specialized calls using $meth =$ `java/lang/Class->forName`.

The instruction takes one argument: a reference to a string that identifies the class or interface one wants to reference. On the heap a new `Class` object is allocated and the field `name` is updated to point to the string reference of the class name:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-static}\ v_1\ \ meth \\ meth = \texttt{java/lang/Class->forName} \qquad loc = R(v_1) \qquad o = H(loc) \\ o.class \preceq \texttt{java/lang/String} \qquad o.field(\texttt{value}) \in \mathsf{ClassName} \\ (H', loc_{cl}) = newObject(H, \texttt{java/lang/Class}) \qquad o_{cl} = H'(loc_{cl}) \\ o'_{cl} = o_{cl}[field \mapsto o_{cl}.field[\texttt{name} \mapsto loc]] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H'[loc_{cl} \mapsto o'_{cl}], \langle m, pc+1, R[\texttt{retval} \mapsto loc_{cl}]\rangle :: SF\rangle}$$

For the analysis, register $v_1$ may contain several values but we can safely ignore anything other than strings, as the API method will only accept a string as an argument. Every string reference in $v_1$ is transferred to a new location on the heap, into the field `name` on the object identified by the type `java/lang/Class`, the current method and program counter. The same `Class` reference is placed in the `retval` register:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)\colon \texttt{invoke-static}\ v_1\ \ meth \\
\quad \text{iff} \quad meth = \texttt{java/lang/Class->forName} \\
\qquad \forall(\mathsf{ObjRef}\ (\texttt{java/lang/String}, m', pc')) \in \hat{R}(v_1)\colon \\
\qquad\quad \{\mathsf{ObjRef}\ (\texttt{java/lang/String}, m', pc')\} \subseteq \\
\qquad\qquad \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m, pc))(\texttt{name}) \\
\qquad \{\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m, pc)\} \subseteq \hat{R}(m, pc+1)(\texttt{retval}) \\
\qquad \hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1)
\end{array}
$$

### 5.1.4 Method Objects

A `Method` object represents a method declaration, i.e., an element in the MethodDeclaration domain. This means that `Class.getMethod()` finds a method declaration resolved from the class or interface represented by the `Class` object. For the semantics, we use two auxiliary functions: *resolvePublicMethodDeclaration* and *newMethodObject*. The first takes the value from the class name `String` from the `Class` object, the value from the method name `String` referenced in argument $v_2$ and argument types in the array referenced in $v_3$. The function searches through the class and interface hierarchy for a matching MethodDeclaration. The function can only find a method that is defined as public and is not a constructor. The function *newMethodObject* is given the method declaration and the existing heap and returns the updated heap and the location of the `Method` object where the relevant fields have been initialized. In fact it creates three new objects: a `Method`, a `Class`, and a `String`, because the field `declaringClass` on the `Method` object references a `Class` where the field `name` references a `String` with the actual class or interface name:

$$
\frac{\begin{array}{c}
m.instructionAt(pc) = \texttt{invoke-virtual}\ v_1\ v_2\ v_3\ meth \\
meth = \texttt{java/lang/Class->getMethod} \\
clname_o = H(R(v_1)).field(\texttt{name}) \qquad clname = H(clname_o).field(\texttt{value}) \\
mname = H(R(v_2)).field(\texttt{value}) \qquad types = H(R(v_3)).field(\texttt{value}) \\
m = resolvePublicMethodDeclaration(clname, mname, types) \\
m \neq \bot \qquad (H', loc_m) = newMethodObject(H, m)
\end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc+1, R[\texttt{retval} \mapsto loc_m]\rangle :: SF\rangle}
$$

The search order of the *resolvePublicMethodDeclaration* function is defined as: The current class, the superclasses of the current class and finally the superinterface hierarchy of the current class. The interface hierarchy of the superclasses are not searched, despite that methods declared in this part of the hierarchy would be found if reflection was not used. This behaviour is consistent with the Java documentation [Ora12] and the behaviour in Android 2.3. However, the search order has been changed in Android 4.0 to

be consistent with the expected behaviour where interfaces of superclasses are searched as well. This change in behaviour is undocumented and we have reported this as a bug [GC12] that is yet to be resolved. Regardless of search order, the function is able to find more than one applicable method declaration due to covariant return types. In such cases, the one with the most specific return type is returned, and if a single return type is not more specific than the others, an arbitrary method declaration is returned.

In the analysis, we define *mref* for readability to be the reference to the new `Method` object. For all references to `Class` objects in $v_1$, there are one or more class names referenced by a `String` on the heap in the field `name`. The set of class names is saved as *clnames* and for each of the references, `String` references from $v_2$ are found and the string values (method names) are saved as *mnames*. The set of method names is also put in the field `name` on the heap at *mref*. Furthermore, we use *resolvePublicMethodDeclarationsFromNames* to do a search through the class and interface hierarchy for valid method declarations, similar to the semantic *resolvePublicMethodDeclaration*, but for sets of class and method names. However, it does not take argument types into account since we do not model arrays precisely enough to do a reasonable comparison of the argument types. For each of the resulting method declarations ($m'$): the class name of the method declaration is created as a `String`, the string reference is put into a new `Class` object and a reference to the `Class` object is put in the field `declaringClass` for *mref* on the heap. Finally, a reference to the method object is present in the `retval` register:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ \mathit{meth} \\
\quad \text{iff}\quad \mathit{meth} = \texttt{java/lang/Class->getMethod} \\
\qquad \mathit{mref} = (\mathsf{ObjRef}\ (\texttt{java/lang/reflect/Method}, m, pc)) \\
\qquad \forall(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m_c, pc_c)) \in \hat{R}(m,pc)(v_1)\colon \\
\qquad\quad \forall(\mathsf{ObjRef}\ (\texttt{java/lang/String}, m_o, pc_o)) \in \\
\qquad\qquad \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m_c, pc_c))(\texttt{name})\colon \\
\qquad\quad \mathit{clnames} = \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/String}, m_o, pc_o))(\texttt{value}) \\
\qquad\quad \forall(\mathsf{ObjRef}\ (\texttt{java/lang/String}, m_s, pc_s)) \in \hat{R}(m,pc)(v_2)\colon \\
\qquad\qquad \{\mathsf{ObjRef}\ (\texttt{java/lang/String}, m_s, pc_s)\} \subseteq \hat{H}(\mathit{mref})(\texttt{name}) \\
\qquad\qquad \mathit{mnames} = \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/String}, m_s, pc_s))(\texttt{value}) \\
\qquad\qquad \forall m' \in \mathit{resolvePublicMethodDeclarationsFromNames}(\mathit{mnames}, \mathit{clnames})\colon \\
\qquad\qquad\quad \beta(m'.\mathit{class.name}) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ \texttt{java/lang/String}, m, pc)(\texttt{value}) \\
\qquad\qquad\quad \{\mathsf{ObjRef}\ (\texttt{java/lang/String}, m, pc)\} \subseteq \\
\qquad\qquad\qquad \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m, pc))(\texttt{name}) \\
\qquad\qquad\quad \{\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m, pc)\} \subseteq \hat{H}(\mathit{mref})(\texttt{declaringClass}) \\
\qquad \{\mathit{mref}\} \subseteq \hat{R}(m, pc+1)(\texttt{retval}) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1)
\end{array}
$$

### 5.1.5 Instantiation

To instantiate new objects through reflection, the API method `Class.newInstance()` is used. It requires a `Class` object representing the class one wants to create a new instance of, and the class must be a regular class, not an interface, abstract class, primitive type or array class. In such cases, an exception is thrown (but this is left out of the semantics and analysis for simplicity). The `Class` object has a reference to a `String` with the class name in the `name` field, and we use the auxiliary function *lookupClass* to find the corresponding class in the semantic Class domain. Next, the new instance is created on the heap using the same function *newObject* as in the regular `new-instance` instruction. Unlike the regular `new-instance` instruction, `Class.newInstance()` also calls the default constructor for the class being instantiated. We use an auxiliary function *lookupDefaultConstructor* to find this constructor, and if none exists the function will return $\perp$ and an exception should be thrown. The constructor is given registers where the argument register has been initialized to a reference to the newly allocated object. Control is transferred to the constructor by adding a new stack frame, just like regular method invocation, but a reference to the newly allocated object is also put into the `retval` register on the stack frame for the current method. A constructor cannot return a value, and therefore this reference cannot be replaced before control is returned to the current method:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ v_1\ \ meth \\ meth = \texttt{java/lang/Class->newInstance} \\ loc_{cl} = R(v_1) \neq \texttt{null} \qquad o_{cl} = H(loc_{cl}) \\ o_n = H(o_{cl}.field(\texttt{name})) \qquad cl = lookupClass(o_n.field(\texttt{value})) \\ (H', loc) = newObject(H, cl) \qquad m' = lookupDefaultConstructor(cl) \neq \perp \\ R' = [0 \mapsto \perp, \ldots, m'.numLocals - 1 \mapsto \perp, m'.numLocals \mapsto H'(loc)] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m', 0, R'\rangle :: \langle m, pc+1, R[\texttt{retval} \mapsto loc]\rangle :: SF\rangle}$$

The flow logic judgement specifies that for all the `Class` references in register $v_1$, `String` references are on the heap to specify the class name, and for each of these class names (*clname*) the semantic class must be found using the function *lookupClass*. A reference for each of these classes is put into the `retval` register for the current method, a default constructor is found and the new object reference is placed as an argument to the constructor:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{invoke-virtual}\ \ v_1\ \ meth \\
\quad \text{iff}\quad meth = \texttt{java/lang/Class->newInstance} \\
\qquad \forall(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m', pc')) \in \hat{R}(v_1)\colon \\
\qquad\quad \forall(\mathsf{ObjRef}\ (\texttt{java/lang/String}, m_s, pc_s)) \in \\
\qquad\qquad \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m', pc'))(\texttt{name})\colon \\
\qquad\qquad\quad \forall clname \in \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/String}, m_s, pc_s))(\texttt{value})\colon \\
\qquad\qquad\qquad cl = lookupClass(clname) \\
\qquad\qquad\qquad \{\mathsf{ObjRef}\ (cl, m, pc)\} \subseteq \hat{R}(m, pc+1)(\texttt{retval}) \\
\qquad\qquad\qquad m' = lookupDefaultConstructor(cl) \\
\qquad\qquad\qquad \{\mathsf{ObjRef}\ (cl, m, pc)\} \subseteq \hat{R}(m', 0)(m'.numLocals) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1)
\end{array}
$$

### 5.1.6 Method Invocation

Once a `Method` object is created it can be used to invoke the method it represents. The API method `Method.invoke()` takes two arguments beside the `Method` object itself: An object reference ($v_2$) for the receiver object on which the method should be invoked, and an array of arguments ($v_3$). The receiver object should be `null` if the method is static, and the method implementation will then be resolved from the declaring class in the `Method` object. We do not formalize the invocation on static methods as this is a straightforward modification of the case with a receiver object. We use the auxiliary function *methodSignature* to extract information from a `Method` object to create a corresponding signature in the semantic MethodSignature domain. The actual method to invoke is resolved using *resolveMethod*, just like in the regular `invoke-virtual` instruction:

$$
\frac{\begin{array}{c}
m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ meth \\
meth = \texttt{java/lang/reflect/Method->invoke} \qquad R(v_1) = loc_1 \neq \texttt{null} \\
o_1 = H(loc_1) \qquad o_1.class \preceq \texttt{java/lang/reflect/Method} \\
meth' = methodSignature(H, o_1) \qquad R(v_2) = loc_2 \neq \texttt{null} \qquad o_2 = H(loc_2) \\
R(v_3) = loc_3 \qquad a = H(loc_3) \in \mathsf{Array} \qquad m' = resolveMethod(meth', o_2.class) \\
a' = unboxArgs(a, m'.argTypes, H) \qquad bf = getBoxingFrame(m'.returnType) \\
R' = [0 \mapsto \bot, \ldots, m'.numLocals - 1 \mapsto \bot, \\
m'.numLocals \mapsto a'.value(0), \ldots, m'.numLocals + a'.length - 1 \mapsto a'.value(a'.length - 1)]
\end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: bf :: \langle m, pc, R\rangle :: SF\rangle}
$$

Before the arguments are transferred to the resolved method registers they may have to be unboxed: The API method receives the arguments in an array with elements of type `Object` (Java `varargs`). This means that if the invoked method has any formal arguments of primitive types, the API method unboxes the primitive values that were boxed before the call occurred. The

primitive values are extracted from the box object based on the argument types of the resolved method. We use an auxiliary function, *unboxArgs*, to unbox all the relevant arguments and return an array with the correctly typed values. These values are then transferred into the relevant registers that are put into a new stack frame along with the method to invoke. The unboxed array, $a'$, is longer than $a$ if any of the unboxed values are of wide data types, i.e., `long` or `double`.

The API method always returns a value of type `Object`, and if the invoked method returns a primitive value it must therefore be boxed by the API method. The return value is not available until the invoked method returns, and therefore we cannot yet box the value. Instead, we add an additional stack frame with a method to be run after the invoked method. We use an auxiliary function *getBoxingFrame* to generate this frame. The function takes the return type of the invoked method as an argument, such that the boxing method is able to determine if the return value should be boxed, and what class it should be boxed in. If boxing is to occur, it boxes the return value from the `retval` register and replaces it with a reference to the boxed value.

In the analysis, for all the `Method` object references in $v_1$, we use the auxiliary function *methodSignatures* to extract and create all possible method signatures that correspond with the information on the heap for the given `Method` object. All these method signatures must be resolved on all the object references for receiver objects in $v_2$. We do not store the order of the arguments in the array referenced in $v_3$, and therefore we cannot determine which of the arguments that must be unboxed. Instead, we transfer all values as they were, as well as unbox all arguments that are object references, if the class ($cl_o$) is a class that can be unboxed. The latter is determined by the auxiliary function *isBoxClass*. Depending on the return type of the invoked method, the return value of `Method.invoke()` is either `null` (if void), unchanged (if it is already of a reference type) or boxed (if it is of a primitive type). The function *primToBoxClass* translates a return type to the corresponding boxing class, e.g. `int` to `Integer`, and the return value of the method is then boxed by putting the value in the field `value` on the heap for the found class and the current method and program counter. In addition, the same object reference is put in the `retval` register for the next program counter in the current method. Finally, we handle any exceptions that are referenced in the exception cache since the invoked method might have thrown an exception:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\text{: } \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ meth \\
\quad \text{iff}\quad meth = \texttt{java/lang/reflect/Method->invoke} \\
\qquad \forall(\mathsf{ObjRef}\ (\texttt{java/lang/reflect/Method}, m_m, pc_m)) \in \hat{R}(m,pc)(v_1)\text{:} \\
\qquad\quad \forall meth' \in methodSignatures(\hat{H}, \mathsf{ObjRef}\ (\texttt{java/lang/reflect/Method}, m_m, pc_m))\text{:} \\
\qquad\qquad \forall(\mathsf{ObjRef}\ (cl_r, m_r, pc_r)) \in \hat{R}(m,pc)(v_2)\text{:} \\
\qquad\qquad\quad m' = resolveMethod(meth', cl_r) \\
\qquad\qquad\quad \{\mathsf{ObjRef}\ (cl_r, m_r, pc_r)\} \subseteq \hat{R}(m',0)(m'.numLocals) \\
\qquad\qquad\quad \forall 1 \leq i \leq arity(meth')\text{:} \\
\qquad\qquad\qquad \forall(\mathsf{ArrRef}\ (a, m_a, pc_a)) \in \hat{R}(m,pc)(v_3)\text{:} \\
\qquad\qquad\qquad\quad \hat{H}(\mathsf{ArrRef}\ (a, m_a, pc_a)) \sqsubseteq \hat{R}(m',0)(m'.numLocals + i) \\
\qquad\qquad\qquad\quad \forall(\mathsf{ObjRef}\ (cl_o, m_o, pc_o)) \in \hat{H}(\mathsf{ArrRef}\ (a, m_a, pc_a))\text{:} \\
\qquad\qquad\qquad\qquad isBoxClass(cl_o) \Rightarrow \\
\qquad\qquad\qquad\qquad\quad \hat{H}(\mathsf{ObjRef}\ (cl_o, m_o, pc_o))(\texttt{value}) \sqsubseteq \hat{R}(m',0)(m'.numLocals + i) \\
\qquad\qquad\quad m'.returnType = \texttt{void} \Rightarrow \beta(\texttt{null}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad\qquad\quad m'.returnType \in \mathsf{RefType} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad\qquad\quad m'.returnType \in \mathsf{PrimType} \Rightarrow \\
\qquad\qquad\qquad cl_b = primToBoxClass(m'.returnType) \\
\qquad\qquad\qquad \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ (cl_b, m, pc))(\texttt{value}) \\
\qquad\qquad\qquad \{\mathsf{ObjRef}\ (cl_b, m, pc)\} \subseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad\qquad\quad \forall(\mathsf{ExcRef}\ (cl_e, m_e, pc_e)) \in \hat{E}(m')\text{:} \\
\qquad\qquad\qquad \mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ (cl_e, m_e, pc_e)), (m,pc)) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)
\end{array}
$$

## 5.2 Javascript Interfaces

Java instance methods can be exposed and invoked from Javascript through Javascript interfaces. This feature is a part of WebKit [Goo12b], and, as described in Chapter 2, it is used in 39% of the apps in our data set. We found the two main uses to be in advertisement libraries and small apps where the main functionality is provided by a webpage. The Java object's methods are exposed to an in-app custom browser using the API method `android/webkit/WebView->addJavascriptInterface()`. In the API documentation, a warning states that it can be a dangerous security issue, and it should not be used unless all of the HTML in the loaded webpage is controlled by the developer. We found a violation of this recommendation: An app that gave access to send text messages through the Javascript interface allowed pages from eBay [eI12] to be loaded through links from its own webpage.

As the feature is widely used we now demonstrate how a call to the method can be handled in our static analysis. We do not specify operational semantics

for the method, as this would require a formalization of how Dalvik and Android exposes the individual Java objects through a `WebView`, and we argue that a sound analysis can be performed, simply by looking at what happens from a Dalvik point of view.

The API method receives three arguments: A `WebView` instance, the `Object` to expose and a `String` with a name to identify the object in Javascript. The name and the `WebView` instance are irrelevant for our analysis since Javascript cannot affect them. On the exposed `Object` it is only possible to access public methods but not constructors or methods that take reference types as arguments except for strings.

The analysis is safe because we assume that every method reachable through the interface can be called with any argument. This includes not just the methods on the Javascript interface class but also methods inherited from its superclasses. For simplicity, we assume that all virtual methods on the interface object are called independently of reference types in their argument list. For all the methods declared in the ancestry of the interface object's class ($cl_i$), we use the *resolveMethod* function to find only the ones that can be resolved from this class. For these methods ($m'$) we say that the value of all the arguments is $\top$, except for the first argument which is the original reference given in $v_2$:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)\colon \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ meth \\
\quad \text{iff} \quad meth = \texttt{android/webkit/WebView->addJavascriptInterface} \\
\qquad \forall(\mathsf{ObjRef}\ (cl_i, m_i, pc_i)) \in \hat{R}(m, pc)(v_2)\colon \\
\qquad\quad \forall cl' \in cl_i.super^* \cup \{cl_i\}\colon \\
\qquad\qquad \forall m' \in \{m' \in cl'.methods \mid m'.kind = \texttt{virtual}\}\colon \\
\qquad\qquad\quad m' = resolveMethod(m', cl_i) \Rightarrow \\
\qquad\qquad\qquad \{\mathsf{ObjRef}\ (cl_i, m_i, pc_i)\} \subseteq \hat{R}(m', 0)(m'.numLocals) \\
\qquad\qquad\qquad \forall 1 \leq i < arity(m')\colon \\
\qquad\qquad\qquad\quad \top \sqsubseteq \hat{R}(m', 0)(m'.numLocals + i) \\
\qquad \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc + 1)
\end{array}
$$

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

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

In the flow judgement we use $\top$ but in practice, the only types that can be received from Javascript are primitive types and strings.

# Chapter 6

# Prototype

The control flow analysis specified as flow logic judgements is itself enough to analyze Dalvik bytecode apps, but without some form of automation, the analysis would be a lengthy process. Our prototype translates Android apps to constraints expressed using Prolog clauses based on the flow logic. It combines several existing tools with our Python parser and constraint generator as shown in Figure 6.1. First, *apktool* extracts the bytecode content of an app and, leveraging another existing tool, *baksmali*, translates the bytecode to *smali*, a human readable format akin to assembly languages with instruction mnemonics, inlined constants and various annotations. We feed this output to our parser which builds lists of classes, methods, instructions, etc., and a tree representing the type hierarchy in the app. Our constraint generator traverses the lists and emits Prolog rules for method resolution, exception handlers, entry points, etc., as well as rules for each instance of each Dalvik instruction in the program. The Prolog program can then be queried for any information that the analysis specifies. This can for example happen interactively or as part of a more specific, programmed analysis. As an example of the latter, we generate call graphs with a special query and further process the output to visualize the call graph of an app.

The source code for the prototype is available at:
`https://bitbucket.org/erw/dalvik-bytecode-analysis-tool`

Section 6.1 explains the smali format and our parser, Section 6.2 explains the background of using Prolog as the constraint solver, and Sections 6.3 through 6.6 detail the constraint generator itself. Section 6.7 discusses running the analysis on real apps.

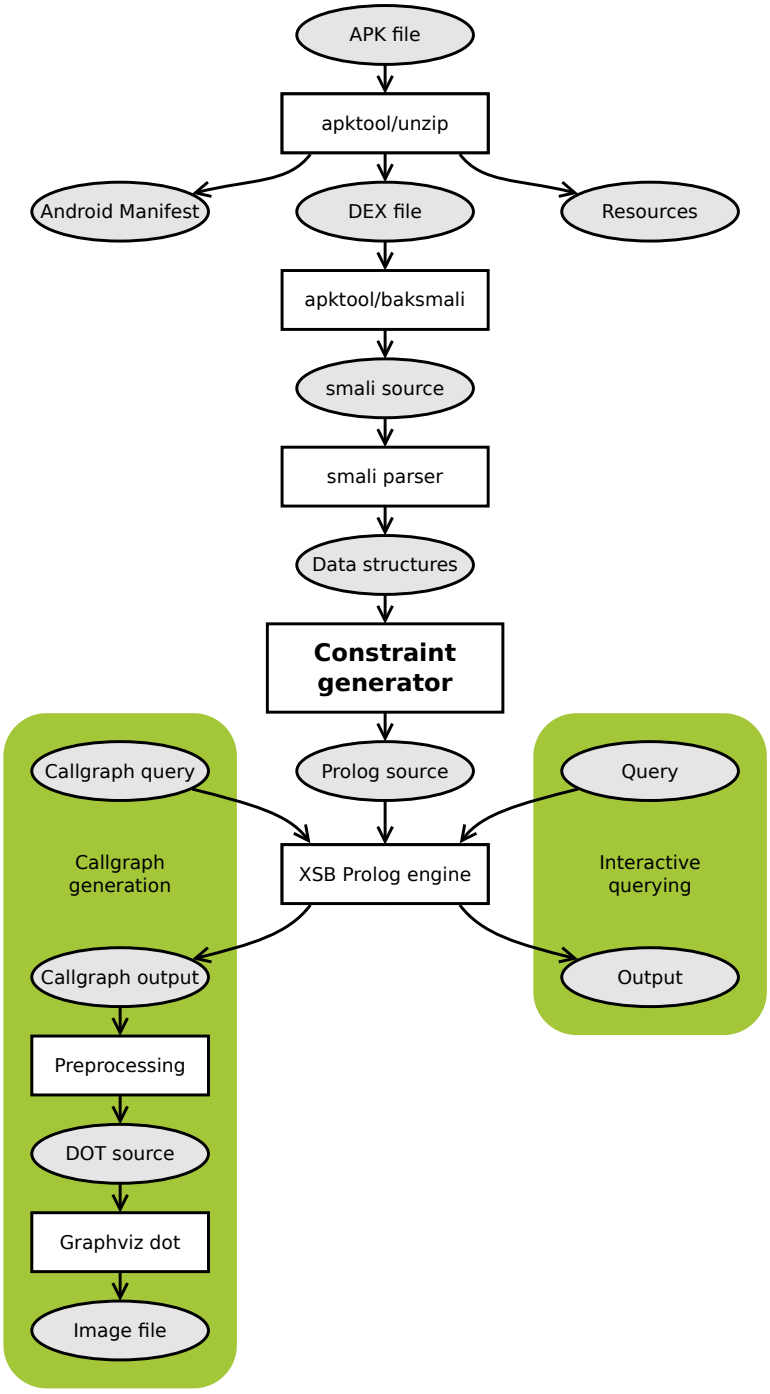


Figure 6.1: Diagram of the prototype of the analysis tool. Rectangles represent data processors and ellipses represent data.

## 6.1 Smali

Smali is both the name of a mnemonic language for Dalvik bytecode and an assembler for this language [jes11]. The corresponding disassembler is named *baksmali*, and we use it to disassemble DEX files into source code that our parser can process. We use it as part of *apktool* [Bru11] which is another open-source utility that streamlines the process of assembling and disassembling complete android apps. Baksmali creates a smali file for each class in the app. The structure of such a file is:

```
1  .class modifiers... Lsome/package/SomeClass;
2  .super Lsome/package/SuperClass;
3  .implements Lsome/package/ISomeInterface;
4
5  .method modifiers... methodName(Larg/types;)Lreturn/type;
6      .locals ...
7      instruction ...
8      instruction ...
9      instruction ...
10     ...
11 .end method
12
13 .method ...
14     ...
15 .end method
16
17 ...
```

Nested classes in Java are renamed with a $ sign in the class name for each nesting level so no classes contain within them other classes. Smali contains other structures such as switch tables and array data but these are not nested either. For each file, our parser records the class metadata and passes over the instructions while keeping track of the method they appear in and the metadata of methods including its declaration information and the number of registers used.

## 6.2 Prolog, Tabling and Termination

A part of the prototype is implemented in XSB Prolog [XSB12], a variant of the logic programming language Prolog [Dan12]. Logic programming is a

way to express mathematical logic, and it is easy to express the flow logic judgements with it. A Prolog program consists of rules and facts that specify relations. A program can be queried to determine if a relation is true, or for which values are able to make a relation true.

The data structures in Prolog are called terms. They can all be categorized as one of four types: atoms, numbers, variables or compound terms (composed of a functor and a list of arguments).

Listing 6.1 shows a Prolog program, where a small family tree has been encoded: `peter` and `sarah` are parents to `james`, and `sophia` is the child of `james` and `catherine`.

```
1 parent(peter, james).
2 parent(sarah, james).
3 parent(james, sophia).
4 parent(catherine, sophia).
5 ancestor(A, C) :- parent(A, C).
6 ancestor(A, C) :- parent(A, X), ancestor(X, C).
```

Listing 6.1: Prolog example.

The `parent` relation is specified using the facts in line 1–4. The names `peter, james, sarah, catherine` and `sophia` are atoms, note that they must start with a lowercase letter, or alternatively be surrounded by singlequotes. We have also created a relation called `ancestor` in line 5–6 to specify when `A` is an ancestor to `C`. `A` and `C` are variables (starts with an uppercase letter). The relation is specified as two rules: At line 5, the right side of `:-` (called the rule body) specifies that the left side (rule head) is true if `A` is `C`'s parent, that is, if a relationship exists such that `parent(A, C)`. The rule body in line 6 states that the rule head is true, only if there exists a relation where `A` is a parent to `X`, who must be an ancestor of `C`. The comma between the two goals indicate that they both must be true.

The program can be queried to determine if something is true, e.g.

```
parent(peter, james).
parent(peter, sarah).
```

would yield "yes" and "no", respectively. Variables can be used in the queries, such that

```
parent(P, james).
```

would yield both `P = peter` and `P = sarah`. Furthermore, the rules can be

queried such that

```
ancestor(Y, sophia).
```

would yield `james`, `catherine`, `peter`, and `sarah` as possible values of `Y`.

Prolog queries are evaluated by a Prolog engine, and the evaluation of regular Prolog uses a depth-first search through trees built from the facts and rules. However, if rules are left-recursive it will result in programs never terminating. For example, if the recursive rule of the `ancestor` relation was specified as

```
ancestor(A, C) :- ancestor(X, C), parent(A, X).
```

regular Prolog implementations will recurse until they run out of stack space.

We use XSB Prolog because it is able to use a different evaluation strategy called SLG that involves tabling (memoization). This results in faster evaluation, and ensures termination of our program analysis, even though we specify rules which are left-recursive (back-edges in the program). For a more detailed description of XSB Prolog and its evaluation strategy, refer to [SW+11b] and [SW+11a].

## 6.3 Examples of Instructions

Here we demonstrate the conversion of flow logic judgements to Prolog source code. For example, the judgement for the `const` instruction is:

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

For a `const` instruction at pc 48 in method `m1` in some app, these two Prolog clauses will be generated:

```
1 % 48: const v5, 0x1
2 hatR(m1, 49, 5, 0x1).
3 hatR(m1, 49, V, Y) :-
4         hatR(m1, 48, V, Y),
5         V \= 5.
```

As can be seen, this conversion and instantiation is fairly straightforward. The Python expression that generates the above looks as follows:

```
clause(hatR(address.next(), reg_num(address, dest), value))
+ transfer_exclude(address, reg_num(address, dest))
```

where `dest` and `value` are the arguments to the instruction as strings, and `address` is an object representing the current address. The Python function `hatR()` inserts an address, a register number, and a value into a Prolog `hatR/4` expression. The function `reg_num()` converts a register name (e.g., `v5` or `p2`) to a register number in the context of an address (`5` and `address.method.num_locals + 2`). The function `clause()` simply joins its argument Prolog expressions into a clause with the first argument being the head of the rule, or simply a fact if it is the sole argument. The function `transfer_exclude()` itself calls `clause()` and `hatR()` to generate a rule to transfer all registers except one to the next program point.

An example of a slightly more advanced instruction is `iput` which sets an instance field to a given value on a given object, provided the object's class matches the one that is part of the (fully qualified) field name. Its flow logic judgement is:

$$
\begin{aligned}
&(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)\colon \texttt{iput}\ \ v_1\ \ v_2\ \ \mathit{fld} \\
&\quad \text{iff}\quad \forall(\mathsf{ObjRef}\ (cl, m', pc')) \in \hat{R}(m, pc)(v_2)\colon \\
&\qquad cl \preceq \mathit{fld}.\mathit{class} \Rightarrow \\
&\qquad\quad \hat{R}(m, pc)(v_1) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ (cl, m', pc'))(\mathit{fld}) \\
&\qquad \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc + 1)
\end{aligned}
$$

As an instantiation of the Prolog code for `iput`, we show one from the method `Lru/watabou/moon3d/MoonView;-><init>(Landroid/content/Context;)V` which we abbrivate to `m2`:

```
% 3: iput v1, p0, Lru/watabou/moon3d/MoonView;->targetSunAngle:F
hatH((CL, CLAddrMID, CLAddrPC), 'targetSunAngle', Y) :-
        hatR(m2, 3, 2, (CL, CLAddrMID, CLAddrPC)),
        subclass(CL, 'Lru/watabou/moon3d/MoonView;'),
        hatR(m2, 3, 1, Y).
hatR(m2, 4, V, Y) :-
        hatR(m2, 3, V, Y).
```

The references from register `p0` (register 2) at the current program counter are extracted. The subclass condition check is implicit: If the `subclass/2` goal fails, the `hatH/3` relation does not hold for those particular arguments and another reference from register `p0` can be tried. If it succeeds, the variable `Y` is bound to each value from the source register in turn. Also, all registers are transferred to the next program point.

## 6.4 Prolog Relations

This section gives an overview of the Prolog relations our output programs consist of. Some relations represent the analysis itself while others represent program structure, method resolution, and auxiliary functions. Some facilitate interactive querying and one is used for the specific call graph analysis. Some are introduced to simplify the representation or to improve the efficiency of the analysis.

### 6.4.1 Analysis

The relations `hatS/3`, `hatH/3`, `hatR/4`, and `hatE/2` represent the values of the abstract domains that make up the analysis, $\hat{S}$, $\hat{H}$, $\hat{R}$, and $\hat{E}$. The numbers specify the arity, i.e., the number of terms the relations relate.

To represent methods and classes, we use their fully qualified names in the Java `Class.getName()` notation, i.e., `Lfully/qualified/name;`. Instead of using fully qualified field names as in Dalvik and our semantics, we split them into fully qualified class names and simple field names to facilitate querying all fields on a specified class. Therefore, the `hatS/3` relations looks as follows, compared to its corresponding abstract domain:

$$\widehat{\mathsf{StaticHeap}} = \mathsf{Field} \to \widehat{\mathsf{Val}} \qquad \texttt{hatS(class, field, value)}$$

where `class`, `field`, and `value` represent Prolog terms.

For simplicity, we have flattened the dynamic heap and the mapping of objects and arrays to their values into a single relation. Since names of classes and array types cannot collide (the latter always start with the square bracket character, '`[`'), it is not a problem to mix object and array references:

$$\begin{aligned} \widehat{\mathsf{Heap}} &= \overline{\mathsf{Ref}} \to (\widehat{\mathsf{Object}} + \widehat{\mathsf{Array}}) \\ \widehat{\mathsf{Object}} &= \mathsf{Field} \to \widehat{\mathsf{Val}} \\ \widehat{\mathsf{Array}} &= \widehat{\mathsf{Val}} \end{aligned} \qquad \texttt{hatH(ref, field, value)}$$

To model arrays as objects, we introduce an artificial field named `values` to store the values. Because of the textual object graph representation, the `ref` term will be a triple representing the class or array type and the textual creation point.

The register contents relation follows the $\widehat{\mathsf{LocalReg}}$ domain with the method and program counter value specified directly:

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

```
hatR(method, pc, reg, value)
```

For efficiency, we replace the use of the special program counter value and register END with the `return/2` relation which specifies return values from methods:

```
return(method, value)
```

Similarly, when methods are invoked and arguments are transferred in $\hat{R}$ to *pc* 0 of the invoked method, we use the special `invoke/3` relation:

```
invoke(method, argnum, val)
```

This allows us to specify argument numbers instead of register numbers such that the mapping between these numbers can be calculated statically once instead of at Prolog runtime each time a method is invoked. Thus, for each method, rules like the following will be generated. Here, the method `m` takes three arguments in registers 6, 7 and 8 (registers 0 to 5 are used for local variables):

```
hatR(m, 0, 6, Y) :- invoke(m, 0, Y).
hatR(m, 0, 7, Y) :- invoke(m, 1, Y).
hatR(m, 0, 8, Y) :- invoke(m, 2, Y).
```

The final analysis relation is the exception cache:

$\widehat{\mathsf{ExcCache}} = \mathsf{Method} \to \mathcal{P}(\overline{\mathsf{ExcRef}})$ `hatE(method, class)`

The contrast between values and sets of values in the analysis is especially visible in this last comparison of domain and relation, but it is present in all the above relations. Sets appear implicitly in our Prolog representation since the same terms may appear in several relationships. For example, if a field named `myField` on an object with reference `r` may contain the values from the set $\{0, 1, 2\}$, the following relationships all hold:

```
hatH(r, 'myField', 0).
hatH(r, 'myField', 1).
hatH(r, 'myField', 2).
```

### 6.4.2 Program Structure

We only represent the parts of the program structure that are necessary for the analysis. The `super/2` relation specifies pairs of superclasses and their

immediate subclasses. It is used among other places in the template for the rules for the `invoke-super` instruction. Some facts of this relation are based on the app under analysis while those that pertain to Java appear in the Prolog output for every app. The fact

```
super(bot, 'Ljava/lang/Object;').
```

follows from our definition of the superclass of `Object`. Here, `bot` is simply a Prolog atom representing $\bot$.

For use in exception handling, we model the standard Java exception type hierarchy:

```
super('Ljava/lang/Object;', 'Ljava/lang/Throwable;').
super('Ljava/lang/Throwable;', 'Ljava/lang/Exception;').
super('Ljava/lang/Throwable;', 'Ljava/lang/Error;').
super('Ljava/lang/Exception;', 'Ljava/lang/RuntimeException;').
...
```

The relation `method/7` links methods to their method declaration information which is used by querying helper relations (see Section 6.4.4).

The remaining program structure is not needed or otherwise present in the Prolog program. Notably, the *instructionAt* access function is not modelled explicitly since the constraints represented by each instruction are present in the program directly as Prolog rules.

### 6.4.3 Auxiliary Functions

The relations `unop/3` and `binop/4` are straightforward:

| | |
|---|---|
| $\widehat{unOp}_{op}(c)$ | `unop(op, c, result)` |
| $\widehat{binOp}_{op}(c_1, c_2)$ | `binop(op, c1, c2, result)` |

In the current implementation, we simply let any calculation return $\top_{\widehat{\mathsf{Prim}}}$ with the following two facts. They use the Prolog anonymous variable, `_`, to ignore their arguments:

```
unop(_, _, top_prim).  binop(_, _, _, top_prim).
```

Method resolution corresponding to the use of *resolveMethod*$(meth, cl)$ involves the relations `resolveFact/5` and `resolve/5` and is discussed in Section 6.5.

Our auxiliary function *super*$^*$ metioned in Section 5.2 is represented by the `ancestor/2` relation which relates pairs of classes as the transitive closure of the `super/2` relation (similar to Listing 6.1).

Our current version of the representation function, $\beta$, which maps values into singleton sets, has no representation in our Prolog programs due to the implicit representation of sets.

The remaining auxiliary functions concern exception handling. `handler/6` specifies the existence of an exception handler. The relations `canHandle/4`, `isFirstHandler/4`, and `findHandler/4` correspond to the predicates and the function of the same names in [WK12].

The HANDLE predicate of Section 4.2.2 expands to the following rules for an exception with reference `(cl_e, m_e, pc_e)` handled at address (`m`, `pc`):

```
1  % If a handler is found, transfer the exception reference to retval at the
       handler's location
2  hatR(m, HandlerPC, retval, (cl_e, m_e, pc_e)) :-
3          findHandler(m, pc, cl_e, HandlerPC),
4          HandlerPC \= bot.
5
6  % If a handler is found, transfer other registers unchanged
7  hatR(m, HandlerPC, V, Y) :-
8          findHandler(m, pc, cl_e, HandlerPC),
9          HandlerPC \= bot,
10         hatR(m, pc, V, Y),
11         V \= retval.
12
13 % If no handler is found, add the exception reference to the cache
14 hatE(m, (cl_e, m_e, pc_e)) :-
15         findHandler(m, pc, cl_e, HandlerPC),
16         HandlerPC = bot.
```

### 6.4.4 Querying

Basic queries use one of the analysis relations, for example `hatR/4`:

```
1  | ?- hatR('Lsome/package/FooClass;->barMethod()V', 27, 4, Y).
2  Y = 0
3  Y = 75
4  no
```

To make it easy to get an overview of the data flow in a method, we created the `methodContent/1` relation which, provided with a method, prints the set of contents of each register containing a value at each program point in the method:

```
1  | ?- methodContent('Lmy/pkg/MyActivity;->onCreate(Landroid/os/Bundle;)V').
2  NumLocals: 8
3
4  PC 0:
5  invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
6  v8: [(Lmy/pkg/MyActivity;  ',' _h243 ',' _h244)]
7  v9: [(Landroid/os/Bundle;  ',' android ',' 0)]
8
9  PC 1:
10 :try_start_0
11 const-string v4, "my.pkg.ClassB"
12 ...
```

It also prints labels and the smali bytecode mnemonics as shown above to provide an easy overview. Unlike other Prolog relations, the output is not related to the given method name in the relation but is instead printed as a side effect.

The terms `_h243` and `_h244` on line 6 are unbound variables from the Prolog engine which means that the method is called on any object of type `MyActivity` independently of creation point. The (`android`, 0) artificial creation point seen on line 7 stems from the argument being created by Android, i.e., instantiated outside of the app under analysis. We treat the topic of entry points in Section 6.6.3.

### 6.4.5 Call Graph Analysis

The `methodCall/3` relation relates two methods if the first calls the second. The third argument denotes whether the call is reflective or normal:

`methodCall(caller, callee, calltype)`

For each Dalvik `invoke` instruction, we generate a rule of the form

```
1  methodCall(m, Callee, regular) :-
2          hatR(m, pc, reg_obj, (Class, _, _)),
3          resolve(Class, method_name, arg_types, return_type, Callee).
```

where (`m`, `pc`) is the address of the `invoke` instruction, `reg_obj` is the register containing the references of the objects the method is invoked on (e.g., `v4` for `invoke-virtual {v4, v2, v13}, L...`), and `method_name`, `arg_types`, and `return_type` have the appropriate values for the method signature specified with the `invoke` instruction.

For `invoke-direct` and `invoke-static`, a `methodCall/3` fact is generated instead of a rule because of static method resolution as we discuss in Section 6.5.

To run the analysis and extract the call graph information, we query the auxiliary relation `printMethodCalls/0` which calls `methodCall/3` and prints a line for each method call. With some further processing, this becomes DOT source code which can be rendered to an image by the Graphviz *dot* graph visualizer [Gra12]. An example of such an image for a small test app is shown in Figure 6.2 on the following page. The call type is illustrated with reflective calls as dashed arrows. Methods that we handle as special cases of Dalvik `invoke` instructions are placed in the API area.

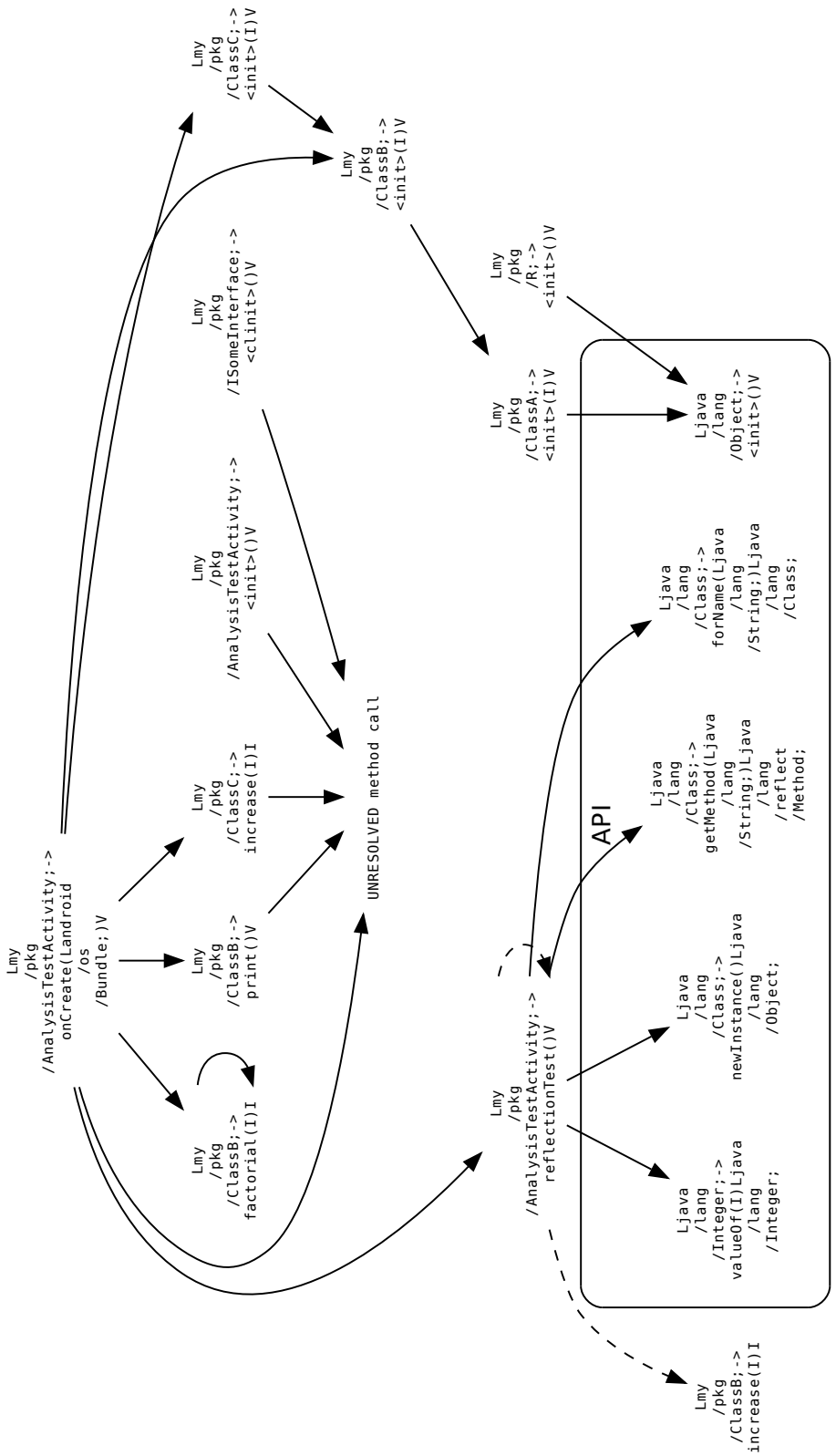


Figure 6.2: Example of a call graph. Dashed arrows represent reflective calls.

## 6.5 Method Resolution

We created the `resolve/5` relation to model the *resolveMethod*(*meth*, *cl*) semantic function:

```
resolve(class, method_name, arg_types, return_type, method)
```

but instead of implementing it recursively like *resolveMethod*, we create facts that can be looked up quickly. Each smali file represents a class and contains a `.super` pseudo-instruction. During the parsing we record pairs of classes and their superclasses, and from this, we build a tree of the class hierarchy in the app. Then, for each method in the app, we walk the tree down from the method's class to find all subclasses where resolving the method signature would lead to that method, i.e., those that do not implement the method themselves. For each of these classes, we generate a fact of the `resolveFact/5` relation, which relates the same arguments as `resolve/5`. For example, if classes `A` through `D` form an inheritance chain and `A` and `C` implement the method `int foo()`, these facts would be generated:

```
1 resolveFact('Lmy/pkg/A;', 'foo', [], 'I', 'Lmy/pkg/A;->foo()I').
2 resolveFact('Lmy/pkg/B;', 'foo', [], 'I', 'Lmy/pkg/A;->foo()I').
3 resolveFact('Lmy/pkg/C;', 'foo', [], 'I', 'Lmy/pkg/C;->foo()I').
4 resolveFact('Lmy/pkg/D;', 'foo', [], 'I', 'Lmy/pkg/C;->foo()I').
```

The difference between `resolveFact/5` and `resolve/5` is that the former represents resolutions that will succeed while the latter fails with the string `'UNRESOLVED method call'` when a method that is not implemented in the app itself is invoked:

```
1 resolve(Class, MethodName, ArgTypes, ReturnType, Method) :-
2         (resolveFact(Class, MethodName, ArgTypes, ReturnType, Method), !) ;
3         Method = 'UNRESOLVED method call'.
```

The cut ensures that a method will not be both resolved and unresolved. We discuss the consequences of unresolved methods in Section 6.6.1.

For `invoke-static`, the method resolution does not depend on an object and can therefore be performed statically. For `invoke-direct` the method is implemented in the class in the method signature and there is no resolution at all, only a lookup to see if the implementation exists. For these instructions the usual invoke rules using `resolve/5` are replaced by ones that transfer the arguments and return value directly in and out of the statically resolved method.

## 6.6 Modelling Java and Android

This section discusses features and program components that are not implemented in the apps and must therefore be modelled separately.

### 6.6.1 API Methods

Methods that are called but not implemented in an app may be from Java standard classes, Android APIs, and from other apps signed by the same developer key. They can also come from classes loaded at runtime but apps that do this are not amenable to static analysis before installation in the first place as discussed in Chapter 2.

Without handling external methods in some way, it is not possible to resolve the implemented methods. The potential effects of a single method are far-reaching. Using reflection, any method, including private ones, can be called, and anything reachable on the heap from the references given to the unknown method can be changed, again including private and final fields. It is even possible that debugging or diagnostics APIs allow programmatic access to the full heap.

In a specialized analysis it would be useful to be able to trust Java and Android API methods not to do anything malicious, for example by just setting their return values to top, but due to the possibilities of affecting the heap, every API method would require some inspection to determine its effects on the heap. APIs can also be handled as we have done with parts of the reflection API by modelling the methods with individual flow judgements, or they could be compiled to Dalvik bytecode and analyzed along with the app. This last approach might impact the running time of the analysis by increasing the effective size of the app considerably and would only work for the parts of the Java standard classes that are implemented in Java.

### 6.6.2 Java Features

We have already covered the modelling of the `java/lang/Object` class and of exceptions in Section 6.4.2.

Another Java feature we needed to represent is the `java/lang/Class` instances that represent primitive types. They are stored as static fields named `TYPE` on the corresponding box classes, so an `sget` instruction is

produced by the Dalvik compiler instead of `const-class`. For example, the `java/lang/Class` instance representing the `int` type is stored as `Integer.TYPE`. We model this with two Prolog facts like the following for each of the eight primitive Java types plus `void`:

```
hatS('Ljava/lang/Integer;', 'TYPE:Ljava/lang/Class;',
        ('Ljava/lang/Class;', java, 1)).
hatH(('Ljava/lang/Class;', java, 1), 'name', 'int').
```

Here, (`java`, 1) is introduced as the creation point of the `java/lang/Class` instance. The program counter values from 1 to 9 are used for the 9 objects.

### 6.6.3 Entry Points

As discussed in Section 3.3, Android apps are simply a collection of classes with methods that can be called by the Android system.

We have identified 1,695 `on`*`SomeEvent`*`()` methods in the Android API, for example `onLocationChanged()` or `onPictureTaken()`. Some are on interfaces, other on classes. We generate Prolog facts to simulate calls to the methods on all classes in the app that implement one of the relevant API interfaces and on all subclasses of the relevant API classes. We also call the constructors on subclasses of the four main app components that are instantiated by Android: activities, services, broadcast receivers, and content providers. The remaining many minor app components are listeners that are instantiated by the app itself before they are registered such that Android can call them.

All of these entry point methods are instance methods and as a simple overapproximation, we invoke the methods on all object references of the class, the method is implemented in, using the anonymous variable to ignore the creation point, e.g.:

```
(Lmy/pkg/MyActivity$1;->onClick(Landroid/view/View;)V, _, _)
```

The arguments passed to the entry point methods are $\top_{\widehat{\mathsf{Prim}}}$ for primitive arguments and otherwise objects with the artificial creation point (`android`, 0) to show that the argument was created by the Android system.

Class constructors (`<clinit>`) also form entry points but they do not need to be called because they do not depend on arguments. All instructions generate constraints whether the method they reside in is called or not.

## 6.7 Analyzing Real Apps

We have tested the prototype with many forms of interactive querying and with call graph generation on several apps from our data set. For real apps, extracting the call graph may take seconds to hours depending on the size. Every Dalvik instruction is converted to a number of Prolog rules, and some apps are too big to analyze even on a server with 68 GB RAM. Since we have implemented and improved the analysis simultaneously, readability and debuggability of the Prolog code has been important to us. We expect that with a different approach it would be possible to improve the efficiency at the cost of the readability of the output.

As an example of an approach to malware detection, we have examined apps that send text messages to see which numbers are used as destination. In many cases, it was not possible to know the numbers because they come from API methods that we do not support. The typical pattern for legitimate apps that send text messages is to retrieve numbers from the database with the user's contact list which requires a large number of API calls, some to connect to the database, some to read the content, and some to store and retrieve the results from Java collections.

A typical pattern for malware is to send messages to hardcoded numbers. Some apps that use hardcoded numbers are specialized and store different numbers for different countries in XML files which require a number of API calls to parse and extract. We found an app where our analysis determined that the string `1277` was the only possible value given as the destination argument of `sendTextMessage()`, but the description of the app (*Find Taxa Danmark*) on Google Play states that the app sends a premium message, so it cannot be classified as malicious even though it seems to make money on inattentive users.

We also analyzed a known malicious Russian app [Ali11] posing as a movie player, and our analysis could confirm this by querying the destination argument of `sendTextMessage()`:

```
invoke('Landroid/telephony/SmsManager;->sendTextMessage(Ljava/lang/String;
    Ljava/lang/String;Ljava/lang/String;Landroid/app/PendingIntent;
    Landroid/app/PendingIntent;)V', 1, ('Ljava/lang/String;', M, PC)),
    hatH(('Ljava/lang/String;', M, PC), 'value', Y).
```

This query yielded the numbers 3353 and 3354 for which each text message may cost around €4–8 [SP12a, SP12b].

# Chapter 7

# Conclusion

We have defined a formal semantics and control flow analysis of Android apps with support for dynamic features including reflection. The level of detail in the analysis enables it to yield useful results for real-world Android apps. This includes textual object graphs and handling of details close to the actual implementation of specific features on Android.

During our earlier comprehensive study of features used in Android, we discovered that reflection is used extensively by Android developers. We have expanded the study of these features and specified operational semantics for central parts of the Java reflection API, as well as expanded the control flow analysis to demonstrate how these dynamic features can be analyzed statically prior to app installation. Furthermore, we have expanded the analysis with support for WebKit Javascript interfaces, a feature that allows exposing Java objects to Javascript control in a built-in browser.

To fully analyze an app, all of the APIs it uses need to be handled as a specialized part of the analysis, e.g., in the same way that we have modelled the reflection API. Our analysis is a safe over-approximation that can form the grounds of such an analysis that includes the Java and Android APIs.

We have developed a prototype implementation of the analysis that generates Prolog clauses based on the constraints specified in the analysis. The generated Prolog program can be queried for information such as the content of registers in specific methods and fields on specific objects. This can for example be used to determine which phone numbers are used in the API method for sending text messages, which in turn could be used in analyzing malware. The prototype is also able to generate call graphs for apps, including calls made through reflection.

# Bibliography


[Aeg12] AegisLab. Security Alert 2011-05-11: New SMS Trojan "zsone" was Took Away from Google Market. http://blog.aegislab.com/index.php?op=ViewArticle&articleId=112&blogId=1, May 7th 2012.

[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. Bytecode for the Dalvik VM. http://source.android.com/tech/dalvik/dalvik-bytecode.html, December 13th 2011.

[And11e] 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.

[Cha12] Charles Stephens. The Average iPhone User Has 44 Apps on Their Device Versus Only 32 Apps For Android Smartphone Users. `http://velositor.com/2012/02/27/the-average-iphone-user-has-44-apps-on-their-device-versus-only-32-apps-for-android-smartphone-users/`, May 1st 2012.

[CMS03] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[Cor12] Oracle Corporation. The java community process(sm) program - jsrs: Java specification requests - detail jsr# 133. `http://jcp.org/en/jsr/detail?id=133`, May 29th 2012.

[Dan12] Daniel Diaz. The GNU Prolog web site. `http://www.gprolog.org`, 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.

[eI12] eBay Inc. ebay. `http://www.ebay.com`, May 23rd 2012.

[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.

[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.

[FJM+11] Jeffrey S. Foster, Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Nikhilesh Reddy, Yixin Zhu, and Todd Millstein. Dr. android and mr. hide: Fine-grained security policies on unmodied android. Technical Report CS-TR-5006, University of Maryland, Department of Computer Science, December 9th 2011.

[GC12] Android Issue Tracker Google Code. Issue 31485: Class.getmethod - unclear search order. `http://code.google.com/p/android/issues/detail?id=31485)`, May 23rd 2012.

[Goo12a] Google. Google Play. `https://play.google.com/store`, May 4th 2012.

[Goo12b] Google Inc. android.webkit — Android Developers. `http://developer.android.com/reference/android/webkit/package-summary.html`, May 8th 2012.

[Gra12] Graphviz. Graphviz - graph visualization software. `http://www.graphviz.org/`, May 30th 2012.

[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.

[KGG+09] Adam Kieżun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for string constraints. In *ISSTA 2009, Proceedings of the 2009 International Symposium on Software Testing and Analysis*, Chicago, IL, USA, July 21–23, 2009.

[Kry12] Alexey Kryshen. Dalvik vm & java memory model (concurrent programming on android). `http://stackoverflow.com/questions/6973667/dalvik-vm-java-memory-model-concurrent-programming-on-android`, May 29th 2012.

[KWOH12] Henrik Søndberg Karlsen, Erik Ramsgaard Wognsen, Mads Chr. Olesen, and René Rydhof Hansen. Study, Formalisation, and Analysis of Dalvik Bytecode. In *Bytecode 2012, the Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2012. Informal proceedings.

[LHD+11] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on WebView in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 343–352. ACM, 2011.

[Loc12] Hiroshi Lockheimer. Android and security. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`, Feb 2nd 2012.

[LWL05] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. Technical report, Stanford University, 2005.

[MG04] Jeremy Manson and Brian Goetz. Jsr 133 (java memory model) faq. `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html`, February 2004.

[Mob12] Mobile World Live. Google Play passes 500,000 apps — Mobile Business Briefing. `http://www.mobilebusinessbriefing.com/articles/google-play-passes-500-000-apps/23751`, May 9th 2012.

[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.

[Ora12] Oracle. Class (java platform) - getmethod. `http://docs.oracle.com/javase/6/docs/api/java/lang/Class.html#getMethod(java.lang.String,java.lang.Class...)`, May 23rd 2012.

[SAH+] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript.

[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.

[Sou12] SourceForge. SourceForge. `http://sourceforge.net/`, May 19th 2012.

[SP12a] SMS-Price.ru. Korotkij SMS nomer 3353. `http://sms-price.ru/number/3353/`, June 5th 2012.

[SP12b] SMS-Price.ru. Korotkij SMS nomer 3354. `http://sms-price.ru/number/3354/`, June 5th 2012.

[SW+11a] Terrance Swift, David S. Warren, et al. The xsb system version 3.3 volume 2: Libraries, interfaces and package. `http://xsb.sourceforge.net/manual2/manual2.pdf`, May 23rd 2011.

[SW+11b] Terrance Swift, David S. Warren, et al. The xsb system version 3.3 volume1: Programmer's manual. `http://xsb.sourceforge.net/manual1/manual1.pdf`, June 10th 2011.

[VHU92] Jan Vitek, R. Nigel Horspool, and James S. Uhl. Compile-time analysis of object-oriented programs. In *Proceedings of the 4th International Conference on Compiler Construction (CC)*, pages 236–250, Paderborn, Germany, October 1992.

[Wil11] William Enck and Damien Octeau and Patrick McDaniel and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium, August, 2011. San Francisco, CA.*, 2011.

[WK12] Erik Ramsgaard Wognsen and Henrik Søndberg Karlsen. Study, formalization and analysis of dalvik bytecode, January 5th 2012. 9th semester report, software engineering.

[XSB12] XSB. Xsb. `http://xsb.sourceforge.net`, May 24th 2012.

# 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 Domains

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

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

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

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

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

$$\begin{aligned}\mathsf{MethodDeclaration} = {} & (\mathit{methodSignature}\colon \mathsf{MethodSignature})\times \\ & (\mathit{kind}\colon \mathsf{Kind})\times \\ & (\mathit{accessFlags}\colon \mathcal{P}(\mathsf{AccessFlag}))\times \\ & (\mathit{exceptionTypes}\colon \mathcal{P}(\mathsf{Class}))\end{aligned}$$

$$\begin{aligned}\mathsf{Method} = {} & (\mathit{methodDeclaration}\colon \mathsf{MethodDeclaration})\times \\ & (\mathit{instructionAt}\colon \mathsf{PC}\to\mathsf{Instruction})\times \\ & (\mathit{numLocals}\colon \mathbb{N}_0)\times \\ & (\mathit{handlers}\colon \mathbb{N}_0\to\mathsf{ExcHandler})\times \\ & (\mathit{tableAt}\colon \mathsf{PC}\to\mathsf{ArrayData}\cup\mathsf{PackedSwitch}\cup\mathsf{SparseSwitch})\end{aligned}$$

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

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

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

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

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

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

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

$$\begin{aligned}
\mathsf{Type} &::= \mathsf{RefType} \mid \mathsf{PrimType} \\
\mathsf{PrimType} &::= \mathsf{PrimSingle} \mid \mathsf{PrimDouble} \\
\mathsf{PrimSingle} &::= \texttt{boolean} \mid \texttt{char} \mid \texttt{byte} \mid \texttt{short} \mid \texttt{int} \mid \texttt{float} \\
\mathsf{PrimDouble} &::= \texttt{long} \mid \texttt{double} \\
\mathsf{RefType} &::= \mathsf{Class} \mid \mathsf{ArrayType} \\
\mathsf{ArrayType} &::= \mathsf{ArrayTypeSingle} \mid \mathsf{ArrayTypeDouble} \\
\mathsf{ArrayTypeSingle} &::= \texttt{array}\ \ (\mathsf{RefType} \mid \mathsf{PrimSingle}) \\
\mathsf{ArrayTypeDouble} &::= \texttt{array}\ \ \mathsf{PrimDouble}
\end{aligned}$$

$$\begin{aligned}
\mathsf{StaticHeap} &= \mathsf{Field} \to \mathsf{Val} \\
\mathsf{Heap} &= \mathsf{Ref} \to (\mathsf{Object} + \mathsf{Array}) \\
\mathsf{Object} &= (\mathit{class}\colon \mathsf{Class}) \times (\mathit{field}\colon \mathsf{Field} \to \mathsf{Val}) \\
\mathsf{Array} &= (\mathit{type}\colon \mathsf{ArrayType}) \times (\mathit{length}\colon \mathbb{N}_0) \times (\mathit{value}\colon \mathbb{N}_0 \to \mathsf{Val})
\end{aligned}$$

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

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

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

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

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

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

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

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

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

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

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

$$\mathsf{Configuration} = \mathsf{StaticHeap} \times \mathsf{Heap} \times \mathsf{CallStack}$$

**Subtyping**

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

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

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

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

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

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

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

# Appendix C

# Semantic Rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$newObject$: $\mathsf{Heap} \times \mathsf{Class} \to \mathsf{Heap} \times \mathsf{Ref}$
$newObject(H, cl) = (H', loc)$
where $loc \notin \mathrm{dom}(H)$ , $H' = H[loc \mapsto o]$ , $o \in \mathsf{Object}$ , $o.class = cl$

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

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

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{const-class}\ \ v\ \ cl \\ (H', loc_c) = newObject(H, \texttt{java/lang/Class}) \\ (H'', loc_s) = newObject(H', \texttt{java/lang/String}) \\ o_c = H''(loc_c) \qquad o'_c = o_c[field \mapsto o_c.field[\texttt{name} \mapsto loc_s]] \\ o_s = H''(loc_s) \qquad o'_s = o_s[field \mapsto o_s.field[\texttt{value} \mapsto cl.name]] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H''[loc_c \mapsto o'_c, loc_s \mapsto o'_s], \langle m, pc+1, R[v \mapsto loc]\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{iget}\ \ v_1\ \ v_2\ \ fld \\ R(v_2) = loc \neq \texttt{null} \qquad o = H(loc) \\ o.class \preceq fld.class \qquad R' = R[v_1 \mapsto o.field(fld)] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R'\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{iput}\ \ v_1\ \ v_2\ \ fld \\ R(v_2) = loc \neq \texttt{null} \qquad o = H(loc) \\ o.class \preceq fld.class \qquad o' = o[field \mapsto o.field[fld \mapsto R(v_1)]] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H[loc \mapsto o'], \langle m, pc+1, R\rangle :: SF\rangle}$$

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

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

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

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

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

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

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

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

Throwing of runtime exceptions for the remaining `invoke` instructions is not shown but is similar to that for `invoke-virtual`.

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

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

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

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

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

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

The function *newArray* is similar to *newObject* but allocates an array of the specified type and length and initializes the fields to 0 or `null` depending on the type.

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

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

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

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

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

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

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

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

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

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

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

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

$$canHandle(h, pc, cl_e) \equiv h.startAddr \leq pc \leq h.endAddr \wedge (cl_e \preceq h.catchType \vee h.catchType = \bot)$$

$$isFirstHandler(\eta, i, pc, cl_e) \equiv canHandle(\eta(i), pc, cl_e) \wedge (\forall j \leq i\colon canHandle(\eta(j), pc, cl_e))$$

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

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

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

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

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

# Appendix D

# Abstract Domains

$$\overline{\mathsf{Val}} = \overline{\mathsf{Prim}} + \overline{\mathsf{Ref}} + \{\mathtt{null}\}$$

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

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

$$\overline{\mathsf{ObjRef}} = \mathsf{Class} \times \mathsf{Method} \times \mathsf{PC}$$

$$\overline{\mathsf{ArrRef}} = \mathsf{ArrayType} \times \mathsf{Method} \times \mathsf{PC}$$

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

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

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

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

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

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

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

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

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

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

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

**Notation and Functions**

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

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

$$\begin{array}{l}
\mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ (cl_e, m_e, pc_e)), (m, pc)) \equiv \\
\quad \mathit{findHandler}(m, pc, cl_e) = pc' \neq \bot \Rightarrow \\
\quad\quad \{\mathsf{ExcRef}\ (cl_e, m_e, pc_e)\} \subseteq \hat{R}(m, pc')(\texttt{retval}) \\
\quad\quad \hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc') \\
\quad \mathit{findHandler}(m, pc, cl_e) = \bot \Rightarrow \\
\quad\quad \{\mathsf{ExcRef}\ (cl_e, m_e, pc_e)\} \subseteq \hat{E}(m)
\end{array}$$

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

# Appendix E

# Flow Logic Judgements

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

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

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

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

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

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

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

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

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

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

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

$$(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{new-instance}\ \ v\ \ cl$$
$$\text{iff}\quad \{\mathsf{ObjRef}\ (cl,m,pc)\} \subseteq \hat{R}(m,pc+1)(v)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)$$

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

$$(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{const-class}\ \ v\ \ cl$$
$$\text{iff}\quad \beta(cl.name) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ \texttt{java/lang/String},m,pc))(\texttt{value})$$
$$\{\mathsf{ObjRef}\ (\texttt{java/lang/String},m,pc)\} \subseteq \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/Class},m,pc))(\texttt{name})$$
$$\{\mathsf{ObjRef}\ (\texttt{java/lang/Class},m,pc)\} \subseteq \hat{R}(m,pc+1)(v)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)$$

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

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{iget}\ \ v_1\ \ v_2\ \ \mathit{fld} \\
\quad \text{iff} \quad \forall(\mathsf{ObjRef}\ (cl,m',pc')) \in \hat{R}(m,pc)(v_2)\colon \\
\qquad cl \preceq \mathit{fld}.\mathit{class} \Rightarrow \\
\qquad\quad \hat{H}(\mathsf{ObjRef}\ (cl,m',pc'))(\mathit{fld}) \sqsubseteq \hat{R}(m,pc+1)(v_1) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)
\end{array}
$$

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

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

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

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ \mathit{meth} \\
\quad \text{iff} \quad \forall(\mathsf{ObjRef}\ (cl,m_t,pc_t)) \in \hat{R}(m,pc)(v_1)\colon \\
\qquad m' = \mathit{resolveMethod}(\mathit{meth},cl) \\
\qquad \forall 1 \le i \le n\colon \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.\mathit{numLocals}-1+i) \\
\qquad m'.\mathit{returnType} \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad \forall(\mathsf{ExcRef}\ (cl_e,m_e,pc_e)) \in \hat{E}(m')\colon \\
\qquad\quad \mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ (cl_e,m_e,pc_e)),(m,pc)) \\
\quad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1) \\
\quad \mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ (\texttt{NullPointerException},m,pc)),(m,pc))
\end{array}
$$

Handling of the exceptions from the cache is not shown in the following `invoke` instructions but is identical to that in `invoke-virtual`. Throwing of runtime exceptions is also absent but trivial to add.

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

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

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

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

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

$$(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{return-void}$$
$$\text{iff}\quad true$$

$$(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{new-array}\ \ v_1\ \ v_2\ \ type$$
$$\text{iff}\quad \beta(0) \sqsubseteq \hat{H}(\mathsf{ArrRef}\ (type, m, pc))$$
$$\{\mathsf{ArrRef}\ (type, m, pc)\} \subseteq \hat{R}(m,pc+1)(v_1)$$
$$\hat{R}(m,pc) \sqsubseteq_{\{v_1\}} \hat{R}(m,pc+1)$$

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

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

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

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

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

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

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

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

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

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

# Appendix F

# Reflection

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-static}\ \ v_1\ \ meth \\ meth = \texttt{java/lang/Class->forName} \qquad loc = R(v_1) \qquad o = H(loc) \\ o.class \preceq \texttt{java/lang/String} \qquad o.field(\texttt{value}) \in \mathsf{ClassName} \\ (H', loc_{cl}) = newObject(H, \texttt{java/lang/Class}) \qquad o_{cl} = H'(loc_{cl}) \\ o'_{cl} = o_{cl}[field \mapsto o_{cl}.field[\texttt{name} \mapsto loc]] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H'[loc_{cl} \mapsto o'_{cl}], \langle m, pc+1, R[\texttt{retval} \mapsto loc_{cl}]\rangle :: SF\rangle}$$

$$\begin{array}{l} (\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)\text{: } \texttt{invoke-static}\ \ v_1\ \ meth \\ \quad \text{iff} \quad meth = \texttt{java/lang/Class->forName} \\ \qquad \forall(\mathsf{ObjRef}\ (\texttt{java/lang/String}, m', pc')) \in \hat{R}(v_1)\text{:} \\ \qquad\quad \{\mathsf{ObjRef}\ (\texttt{java/lang/String}, m', pc')\} \subseteq \\ \qquad\qquad \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m, pc))(\texttt{name}) \\ \qquad \{\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m, pc)\} \subseteq \hat{R}(m, pc+1)(\texttt{retval}) \\ \qquad \hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1) \end{array}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ meth \\ meth = \texttt{java/lang/Class->getMethod} \\ clname_o = H(R(v_1)).field(\texttt{name}) \qquad clname = H(clname_o).field(\texttt{value}) \\ mname = H(R(v_2)).field(\texttt{value}) \qquad types = H(R(v_3)).field(\texttt{value}) \\ m = resolvePublicMethodDeclaration(clname, mname, types) \\ m \neq \bot \qquad (H', loc_m) = newMethodObject(H, m) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc+1, R[\texttt{retval} \mapsto loc_m]\rangle :: SF\rangle}$$

$(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)$: `invoke-virtual` $v_1\ \ v_2\ \ v_3\ \ meth$
iff $meth = \texttt{java/lang/Class->getMethod}$
$mref = (\text{ObjRef } (\texttt{java/lang/reflect/Method}, m, pc))$
$\forall(\text{ObjRef } (\texttt{java/lang/Class}, m_c, pc_c)) \in \hat{R}(m, pc)(v_1)$:
$\forall(\text{ObjRef } (\texttt{java/lang/String}, m_o, pc_o)) \in$
$\hat{H}(\text{ObjRef } (\texttt{java/lang/Class}, m_c, pc_c))(\texttt{name})$:
$clnames = \hat{H}(\text{ObjRef } (\texttt{java/lang/String}, m_o, pc_o))(\texttt{value})$
$\forall(\text{ObjRef } (\texttt{java/lang/String}, m_s, pc_s)) \in \hat{R}(m, pc)(v_2)$:
$\{\text{ObjRef } (\texttt{java/lang/String}, m_s, pc_s)\} \subseteq \hat{H}(mref)(\texttt{name})$
$mnames = \hat{H}(\text{ObjRef } (\texttt{java/lang/String}, m_s, pc_s))(\texttt{value})$
$\forall m' \in resolvePublicMethodDeclarationsFromNames(mnames, clnames)$:
$\beta(m'.class.name) \sqsubseteq \hat{H}(\text{ObjRef } \texttt{java/lang/String}, m, pc)(\texttt{value})$
$\{\text{ObjRef } (\texttt{java/lang/String}, m, pc)\} \subseteq$
$\hat{H}(\text{ObjRef } (\texttt{java/lang/Class}, m, pc))(\texttt{name})$
$\{\text{ObjRef } (\texttt{java/lang/Class}, m, pc)\} \subseteq \hat{H}(mref)(\texttt{declaringClass})$
$\{mref\} \subseteq \hat{R}(m, pc+1)(\texttt{retval})$
$\hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1)$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1\ \ meth \\ meth = \texttt{java/lang/Class->newInstance} \\ loc_{cl} = R(v_1) \neq \texttt{null} \qquad o_{cl} = H(loc_{cl}) \\ o_n = H(o_{cl}.field(\texttt{name})) \qquad cl = lookupClass(o_n.field(\texttt{value})) \\ (H', loc) = newObject(H, cl) \qquad m' = lookupDefaultConstructor(cl) \neq \bot \\ R' = [0 \mapsto \bot, \ldots, m'.numLocals - 1 \mapsto \bot, m'.numLocals \mapsto H'(loc)] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m', 0, R'\rangle :: \langle m, pc+1, R[\texttt{retval} \mapsto loc]\rangle :: SF\rangle}$$

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

iff $meth = \texttt{java/lang/Class->newInstance}$

$\forall(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m', pc')) \in \hat{R}(v_1)$:

$\forall(\mathsf{ObjRef}\ (\texttt{java/lang/String}, m_s, pc_s)) \in$

$\hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m', pc'))(\texttt{name})$:

$\forall clname \in \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/String}, m_s, pc_s))(\texttt{value})$:

$cl = lookupClass(clname)$

$\{\mathsf{ObjRef}\ (cl, m, pc)\} \subseteq \hat{R}(m, pc+1)(\texttt{retval})$

$m' = lookupDefaultConstructor(cl)$

$\{\mathsf{ObjRef}\ (cl, m, pc)\} \subseteq \hat{R}(m', 0)(m'.numLocals)$

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

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ meth \\ meth = \texttt{java/lang/reflect/Method->invoke} \qquad R(v_1) = loc_1 \neq \texttt{null} \\ o_1 = H(loc_1) \qquad o_1.class \preceq \texttt{java/lang/reflect/Method} \\ meth' = methodSignature(H, o_1) \qquad R(v_2) = loc_2 \neq \texttt{null} \qquad o_2 = H(loc_2) \\ R(v_3) = loc_3 \qquad a = H(loc_3) \in \mathsf{Array} \qquad m' = resolveMethod(meth', o_2.class) \\ a' = unboxArgs(a, m'.argTypes, H) \qquad bf = getBoxingFrame(m'.returnType) \\ R' = [0 \mapsto \bot, \ldots, m'.numLocals - 1 \mapsto \bot, \\ m'.numLocals \mapsto a'.value(0), \ldots, m'.numLocals + a'.length - 1 \mapsto a'.value(a'.length - 1)] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: bf :: \langle m, pc, R\rangle :: SF\rangle}$$

$(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)$: $\texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ meth$

iff $meth = \texttt{java/lang/reflect/Method->invoke}$

$\forall(\mathsf{ObjRef}\ (\texttt{java/lang/reflect/Method}, m_m, pc_m)) \in \hat{R}(m, pc)(v_1)$:

$\forall meth' \in methodSignatures(\hat{H}, \mathsf{ObjRef}\ (\texttt{java/lang/reflect/Method}, m_m, pc_m))$:

$\forall(\mathsf{ObjRef}\ (cl_r, m_r, pc_r)) \in \hat{R}(m, pc)(v_2)$:

$m' = resolveMethod(meth', cl_r)$

$\{\mathsf{ObjRef}\ (cl_r, m_r, pc_r)\} \subseteq \hat{R}(m', 0)(m'.numLocals)$

$\forall 1 \leq i \leq arity(meth')$:

$\forall(\mathsf{ArrRef}\ (a, m_a, pc_a)) \in \hat{R}(m, pc)(v_3)$:

$\hat{H}(\mathsf{ArrRef}\ (a, m_a, pc_a)) \sqsubseteq \hat{R}(m', 0)(m'.numLocals + i)$

$\forall(\mathsf{ObjRef}\ (cl_o, m_o, pc_o)) \in \hat{H}(\mathsf{ArrRef}\ (a, m_a, pc_a))$:

$isBoxClass(cl_o) \Rightarrow$

$\hat{H}(\mathsf{ObjRef}\ (cl_o, m_o, pc_o))(\texttt{value}) \sqsubseteq \hat{R}(m', 0)(m'.numLocals + i)$

$m'.returnType = \texttt{void} \Rightarrow \beta(\texttt{null}) \sqsubseteq \hat{R}(m, pc + 1)(\texttt{retval})$

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

$m'.returnType \in \mathsf{PrimType} \Rightarrow$

$cl_b = primToBoxClass(m'.returnType)$

$\hat{R}(m', \mathsf{END}) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ (cl_b, m, pc))(\texttt{value})$

$\{\mathsf{ObjRef}\ (cl_b, m, pc)\} \subseteq \hat{R}(m, pc + 1)(\texttt{retval})$

$\forall(\mathsf{ExcRef}\ (cl_e, m_e, pc_e)) \in \hat{E}(m')$:

$\mathsf{HANDLE}_{(\hat{R}, \hat{E})}((\mathsf{ExcRef}\ (cl_e, m_e, pc_e)), (m, pc))$

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