Informatics

Static Analysis of Low-Level Code

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

Ilya Grishchenko
Registration Number 01652947

to the Faculty of Informatics

at the TU Wien

Advisor: Matteo Maffei
Second advisor: Georg Weissenbacher

External reviewers:
Andrei Sabelfeld. Chalmers University of Technology and Gothenburg University, Sweden.
Karthikeyan Bhargavan. INRIA Paris, Prairie Research Institute, MPRI, France.

Vienna, 10th August, 2020

Ilya Grishchenko   Matteo Maffei

Declaration of Authorship

Ilya Grishchenko

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 10th August, 2020

Ilya Grishchenko

To my mother Olga

Acknowledgements

I am deeply grateful to my supervisor Matteo Maffei. His illuminating vision and sparking ideas were guiding me throughout my PhD-journey.

I extend my deepest gratitude to the reviewers of this work, Andrei Sabelfeld and Karthikeyan Bhargavan, for their time and effort examining this work. It is a great honour to receive their insightful and inspiring comments.

I am very grateful to Stefano Calzavara and Adrien Koutsos for their collaboration, elucidating flow-sensitivity discussions, and great help in establishing proof frameworks. I am deeply indebted to Clara Schneidewind for our long-lasting collaboration, and her original ideas on static analysis and its formalisations, detailed comments on this work, and warm encouragement. Additionally, my gratitude goes to both Clara Schneidewind and Markus Scherer, for keeping the spirit of HoRSt alive no matter what. Special thanks to Marco Squarcina, Mauro Tempesta, and Pedro Moreno-Sánchez, for their constructive comments on this thesis, important suggestions, and warm-hearted encouragement. Many thanks to Kees van Berkel whose everlasting support and meticulous feedback were so helpful to refine this work.

My gratitude also goes to the LogiCS doctoral college, in particular, to Armin Biere, Georg Weissenbacher, Martin Nöllenburg, and Anna Prianichnikova. I am honoured to have been part of this famous and well-established program. Additionally, I want to thank CISPA, where my PhD-journey began, and the Max Planck Institute for Informatics, whose scholarship enabled me to study in Germany.

I am thankful to my friends at TU Wien for the great atmosphere and the many sugar-shocks we had together. Finally, I would like to express my very special thanks to my mom Olga and my friends all around the world for staying close all these years.

Abstract

The Android platform is undoubtedly the most popular platform for smartphones, with thousands of new applications becoming available daily and billions of app installations each year. Ethereum is the most popular smart contract platform, with thousands of applications on the blockchain serving as trading platforms and providing other functionalities. Due to these platforms' popularity, security issues in their applications may reach a catastrophic scale with ease. Several prominent automated techniques help to reveal security problems in applications at the early stages of expansion. One such technique is static analysis. This thesis focuses on the design of static analysis techniques for Android apps and smart contracts distributed in the form of low-level code (bytecode).

After installation, an Android app may get access to a set of sensitive information sources (e.g., location data). Unfortunately, exposure of such information to third parties has led in the past to several cases of privacy breach, and continues to be a serious threat. In this thesis, we tackle information flow propagation in the bytecode of Android applications by sound Horn-clause based abstraction techniques. This work will be the first to use Horn-clause based techniques in the context of security analysis. Moreover, we prove that our approach is sound, that is, our approach provides guarantees for its results. As a consequence, it can be used to show the absence of explicit data leaks in an app. Furthermore, Horn-clause based abstraction techniques are not limited to information propagation tasks, that is, our techniques can be used to show any kind of program property expressed as a reachability property. In addition, our Horn-clause based techniques scale to large codebases, benefit from the advancements in Satisfiability Modulo Theory solving, and allow for favorable performance with respect to the state-of-the-art.

We instantiate the principles that were obtained while developing the analysis techniques for Android applications in the context of Ethereum smart contracts distributed in the form of Ethereum Virtual Machine (EVM) bytecode. Smart contracts are programs mainly used to perform financial operations (e.g., auctions) on cryptocurrency blockchains (e.g., Ethereum). Recent attacks demonstrate that certain vulnerabilities in smart contracts might cause severe money loss and an overall decrease of trust in the technology. Therefore, security analysis of EVM bytecode is in the focus of the research community. This thesis presents two results which establish the foundations for sound security analysis of EVM bytecode. First, the semantics of EVM bytecode is mechanized for the first time and tested against the official Ethereum test suite. This result facilitates both the design of analysis techniques and establishing their correctness properties. Second, this thesis provides the first sound Control Flow Graph reconstruction solution for EVM bytecode, that is, our analysis guarantees that reachable parts of the code are never pruned. This guarantee is required by a number of security properties for smart contracts. We also develop a tool implementing our analysis and successfully evaluate it on a big collection of real-world contracts.

Contents

Preface

The results of this thesis are based on the research work conducted during my Ph.D. studies.

Chapter 2 and Chapter 3 encompass the results of the collaboration with Stefano Calzavara, and Matteo Maffei presented at the 1st IEEE European Symposium on Security and Privacy (EuroS&P) [CGM16] in March 2016.

Chapter 4 and Chapter 5 are based on the joint work with Stefano Calzavara, Adrien Koutsos, and Matteo Maffei presented at the 30th IEEE Computer Security Foundations Symposium (CSF) [CGKM17] in August 2017.

[CGM16] and [CGKM17] describe two static analysis frameworks for Android applications: flow-insensitive and flow-sensitive for the heap elements. The author of the present thesis is the main contributor to these works, with Adrien Koutsos contributing significantly to the development of the theory backing the flow-sensitive part.

Chapter 6 presents the outcomes of the work with Clara Schneidewind and Matteo Maffei accepted for publication at the 7th International Conference on Principles of Security and Trust (POST) [GMS18a] and presented in April 2018. In this chapter we contribute by performing a formalisation in F* of a substantial fragment of EVM semantics. This result enables verification solutions based on F* encoding and facilitates establishing machine-checked proofs for analysis techniques. Moreover, we compile F* into OCaml and evaluate the executable semantics against the official EVM test suite. This work received the EAPLS Best Paper Award at ETAPS 2018.

Finally, Chapter 7 and Chapter 8 focus on the results of the extensive research performed with Clara Schneidewind, Markus Scherer, and Matteo Maffei accepted for publication and expected to be presented in November 2020 at the 27th ACM Conference on Computer and Communications Security (CCS) [SGSM20]. In this work we contribute by defining a static analysis for EVM bytecode's CFG reconstruction. Also we provide a proof of soundness for our analysis technique, its implementation and further extensive evaluation on real smart contracts.

CHAPTER 1

Introduction

Due to Android's growing popularity and its massive user base, vulnerabilities in Android applications can have catastrophic effects on the privacy of users and the security of their data. For instance, sensitive information of millions of users may leak to a third party. In order to block vulnerable apps before they reach their users, the official market requires the apps to go through a vetting procedure, i.e., apps are analyzed before they are accepted. Static analysis techniques play a crucial role in vetting. The idea behind static analysis is simple: it checks whether the app under analysis satisfies a property (e.g., the absence of a particular flow) without executing it, but rather by looking at static information, such as the app's code.

Android applications are shipped to application markets as low-level code, namely, Dalvik bytecode. This thesis presents two analysis frameworks for Dalvik bytecode that provide different trade-offs between performance and precision. In contrast to the state-of-the-art, our frameworks are the first to enjoy the soundness property, i.e., our analyses can be used to show important security properties of Android apps. An example of such a property is the absence of data leaks in the application. In particular, the task of the analysis is to answer whether sensitive information can reach a place in the code where it is leaked to a third party.

Like Android applications, Ethereum smart contracts also require analysis before they become publicly available. Furthermore, Ethereum smart contracts are also distributed as bytecode, in this case Ethereum Virtual Machine (EVM) bytecode. Smart contracts are programs that operate on cryptocurrency blockchains. Often, they perform financial operations, thus flaws in them may lead to money loss. For instance, flaws in the DAO contract [thea] recently caused damage of sixty million dollars. Since smart contracts cannot be patched after they are added to the blockchain, they must be thoroughly analyzed before they are published. The scientific community has proposed several static analysis techniques for smart contracts. Many of these techniques base their results on the control flow graph (CFG) reconstruction. This reconstruction requires a specific analysis that reveals which parts of the code are reachable due to the design of EVM bytecode. A correct CFG is essential for the analysis of many security properties of smart contracts, e.g., the one that captures the severe bug in the aforementioned DAO contract. However, none of the existing CFG reconstruction solutions have a correctness property, hence currently we cannot establish guarantees for any of their subsequent analysis results. In order to bridge this gap, this thesis introduces the first sound static analysis technique for CFG reconstruction of EVM bytecode. The soundness of our analysis technique is shown with respect to the first mechanized concrete semantics of EVM bytecode also introduced in this work.

We base our static analysis techniques for low-level code on *Horn-clause based abstraction*, employed as an approximation technique for verification algorithms over recursive Horn clauses. Intuitively, the goal is to build on state-of-the-art Satisfiability Modulo Theory (SMT) and Datalog solvers [DMB08a, JSS16] to check reachability properties by means of Horn clause resolution.

1.1 Summary of Contributions

The first part of this thesis is dedicated to the static analysis of the bytecode of Android applications. We follow a principled approach to the analysis of bytecode. First, we formalize a concrete semantics to model the behaviour of applications. Second, we define an abstraction of the concrete semantics to make the security analysis possible, as the underlying reachability property (e.g., the absence of data leaks) is undecidable. Then, we prove the soundness of our abstraction. Finally, we implement the resulting static analysis and evaluate its performance.

Android applications have non-trivial semantics that reflects the reactive nature of the application lifecycle (e.g., restarting and resuming an application can be performed arbitrarily many times). Consequently, the number and order of updates are not available statically for the heap elements shared among several lifecycle processes. For example, the process happening on the app's pause can share a heap element with a process happening on the app's resume, and since the user can perform a pause/resume sequence any time, correctly tracking the state of the shared heap element is challenging. To address the issue of non-deterministic heap updates, we consider two possible abstractions.

In our first abstraction, we employ the flow-insensitive treatment for all heap elements. A flow-insensitive heap abstraction cannot forget the information about the heap. For instance, once marked as containing sensitive information, a specific heap element will remain marked as such forever. Although flow-insensitivity for heap elements helps our first abstraction to compute correct heap approximations effectively, it comes at the cost of precision. To illustrate the necessity of a more precise abstraction, let us consider a simple app that writes private data to the heap and then overwrites it with public data before sending the heap's data to a third party: since all the heap elements are flow-insensitive, our first abstraction falsely concludes that the private data reaches the third party.

In our second abstraction, we address this precision problem via partial flow-sensitivity: for some elements of the heap, the second abstraction will perform strong updates, that is, eliminating the heap elements' previous values. Achieving a correct flow-sensitive heap abstraction is particularly challenging due to the reactive nature of Android applications. For instance, the heap elements manipulated by the lifecycle processes must still be updated weakly, i.e., the analysis still does not forget the previous values of these heap elements after the update.

As we base our static analysis techniques on *Horn-clause based abstraction*, we specify both abstractions (flow-insensitive and flow-sensitive for the heap elements) in terms of Horn clauses and show that the abstractions are sound. Our abstractions are the first ones that are sound, while supporting reasoning about arbitrary security properties of Android applications expressed in terms of reachability, that is, our solutions are not limited to pnly catching the aforementioned privacy leaks. For instance, our frameworks can support correctness properties for arithmetic operations that may be used to reveal overflows causing memory corruption and sensitive information disclosure. We implement two tools that differ in the heap's abstraction, namely, HornDroid (flow-insensitive) and fsHornDroid (flow-sensitive). Furthermore, we evaluate our tools against two targets. The first evaluation is performed on a benchmark designed by the static analysis community to compare the performance of various analyzers in terms of precision and recall. During this evaluation our solutions demonstrate favorable performance with respect to the state-of-the-art. The second evaluation considers real-life applications to show the scalability and applicability of the proposed analysis techniques.

The second part of this thesis is devoted to the problem of statically analysing Ethereum smart contracts. The principles discovered while developing the framework for Android applications in the first part of the thesis are applied in the context of Ethereum Virtual Machine (EVM) bytecode. Nevertheless, applying such principles is not without certain challenges, such as formalising the concrete semantics and reconstructing the control flow graph (CFG).

Defining concrete semantics for Ethereum smart contracts has turned out to be demanding, e.g., even the official specification [Woo14] suffered from several inconsistencies. For this reason we introduce the first semantic framework for EVM bytecode that provides a mechanized version of the concrete semantics. The mechanization of the concrete EVM semantics in the F* proof assistant allows us to test our semantics against the suite provided by the Ethereum foundation.

In addition to non-trivial semantics, EVM bytecode is also characterised by problematic control flow operations. In particular, EVM jump instructions transfer the control flow to the targets extracted from the stack, in other words, the previous computations affect the targets' values. This makes control flow graph reconstruction for EVM bytecode a challenging task. On the one hand, the analysis needs to be precise because heavy over-approximations of the control flow lead to the code becoming intractable for the analysis itself and any further analysis. For instance, an over-approximation may even introduce nested cycles in a bytecode without loops. On the other hand, the analysis needs to be sound, i.e., the analysis should not prune reachable parts of the code since they may introduce security issues, such as the one of the DAO contract mentioned earlier. We address the CFG reconstruction challenge for EVM bytecode with the first sound algorithm which is proven correct with respect to the previously mentioned concrete semantics. Furthermore, we implement our algorithm and evaluate it on a massive corpus of real smart contracts, achieving a 98% success rate, and demonstrating an improvement on the performance of the state-of-the-art tools.

1.2 Structure of the Thesis

This thesis is organized into two parts:

- Chapter 2 presents the first sound framework for the static analysis of Android applications based on a flow-insensitive heap abstraction which comes with a proof of soundness described in Chapter 3. Chapter 4 presents an improvement of the framework introduced in Chapter 2. In particular, the improvement allows for providing flow-sensitivity to the analysis of heap elements. A proof of soundness for the resulting framework is detailed in Chapter 5;
- Chapter 6 describes a semantic framework for Ethereum smart contracts. Chapter 7 discusses a static analysis for the control flow graph reconstruction of Ethereum smart contracts. A soundness proof is presented in Chapter 8.

CHAPTER 2

HornDroid: A Practical and Sound Static Analysis of Android Applications by SMT Solving

2.1 Introduction

The Android platform is by far the most popular choice for mobile devices nowadays, with billions of applications routinely installed on a massive number of different phones and tablets. Given this increasing popularity, personal information and other sensitive data stored on Android devices constitute an attractive target for breaching users' privacy at scale by malicious application developers. Information flow control frameworks for Android have thus emerged as a prominent research direction, with several different proposals spanning from dynamic analysis [EGH$^+$14, JAF$^+$13, TR14, HHJ$^+$11] to static analysis [ZO12, YY12, MS12, GCEC12, KYY$^+$12, LLW$^+$12, ARF$^+$14, GKP$^+$15, LBB$^+$15]. Static analysis is particularly appealing for information flow control, given its ability to provide full coverage of all the possible execution paths and the possibility to be employed in the vetting phase, i.e., before the application is uploaded onto the Google Play store.

The most recent works in this area [ARF$^+$14, GKP$^+$15, LBB$^+$15, WROR14] are impressive in their efforts to support a significant fragment of the Android platform. Most of them leverage existing static analysers by encoding Android applications in a suitable format, e.g., FlowDroid [ARF$^+$14], DroidSafe [GKP$^+$15], and IccTA [LBB$^+$15] use Soot [VRGH$^+$00], while CHEX [LLW$^+$12] uses Wala [FD12]. Observing that existing static analysers come with intrinsic limitations that limit the precision of the analysis (e.g., Soot and Wala do not calculate all objects' points-to information in a both flow- and context-sensitive way), Amandroid [WROR14] relies on a dedicated data-flow analysis algorithm.

Despite all this progress and sophisticated machinery, none of these tools achieves a satisfactory degree of soundness: even on benchmarks written by the community and consisting of simple programs (i.e., Droidbench [ARF$^+$14]), for which the ground truth is known, all existing tools miss several malicious leaks (false negatives). This, along with the fact that none of these tools comes with a formal model or soundness proof, makes one wonder how accurately these analyses capture all the subtleties of the Android execution model, which is far from being trivial [PS14], and to which extent their results are reliable on real-life applications, for which the ground truth is not known.

Furthermore, the lack of precise and fully documented analysis definitions complicates the comparison between different approaches: for instance, there is no universal agreement on a single notion of object-sensitivity [SBL11a], though object-sensitivity has been recognized as crucial to support a precise analysis of real-world Android applications [ARF$^+$14]. Hence, at the time of writing, the only way to grasp the relative strengths and weaknesses of different static analysis tools for Android applications relies on an hands-on testing on some common benchmark and a source code inspection of their implementation.

Our Contributions We present a fresh approach to the static analysis of Android applications, i.e., a data-flow analysis based on *Horn clause resolution* [BMR12]. The core idea is to soundly abstract the semantics of Android applications into a set of Horn clauses and to formulate security properties as a set of proof obligations, which can be automatically discharged by off-the-shelf SMT solvers. In particular:

- We prove the soundness of our analysis against a rigorous formal model of a large fragment of the Android ecosystem, covering Dalvik bytecode, the event-driven nature of the activity lifecycle, and inter-component communication. While elaborating the proof, we identified a few critical corner-cases that affect the soundness guarantees provided by some of the previous static analysis tools for Android. We believe that this formal model may constitute a foundational framework, serving as a starting and comparison point for future work in the field;

- We fine-tune the Horn clause generation in order to optimize precision and efficiency, while retaining soundness. Being a data-flow analysis rather than a pure taint analysis, our solution statically approximates runtime values, in contrast to most of the previous works in the field [ARF+14, GKP+15, LBB+15]. This boosts the precision of the analysis: for instance, it makes it possible to statically determine whether a conditional branch will never be taken at runtime and ignore it. A salient feature of our approach is the usage of SMT solving to discharge proof obligations. From an engineering point of view, this allows one to fine-tune the analysis while still building on off-the-shelf verification tools, thereby leveraging the continuous advances in this field.

- We develop a tool, called HornDroid, which implements the analysis described in the formal model and complements it in order to support additional Android features, such as reflection, exceptions, and threading. HornDroid automatically generates Horn clauses from the application bytecode and relies on the state-of-the-art SMT solver *z3* [dMB08b] for discharging proof obligations.

- We conduct a performance evaluation on Droidbench, a collection of 120 programs written by the community, comparing HornDroid with IccTA [LBB+15] (an extension of FlowDroid [ARF+14] to inter-component communication), Amandroid [WROR14] and DroidSafe [GKP+15]. HornDroid outperforms the competitors in terms of sensitivity (i.e., soundness) and performance, while retaining a high specificity (i.e., precision): HornDroid is the only tool that identifies all the explicit information flows, it exhibits just one more false positive than Amandroid (the most accurate tool), and it is one order of magnitude faster than IccTA and AmanDroid, and two orders of magnitude faster than DroidSafe. Furthermore, we show that HornDroid scales well to real-life applications from Google Play by a comparative evaluation on the two largest applications from the Google Play Top 30, i.e., Candy Crash Soda Saga and Facebook, which pose significant problems to existing tools.

2.2 Design and Motivations

Static information flow control for Android applications is a mature research area nowadays [ZO12, YY12, MS12, GCEC12, KYY+12, LLW+12], with IccTA [LBB+15] (an extension of FlowDroid [ARF+14] to inter-component communication), AmanDroid [WROR14] and DroidSafe [GKP+15] representing the state-of-the-art in this field. Although all these proposals are impressive projects, which significantly advanced the area of information flow control for Android applications, they all have significant limitations, motivating the need for novel research proposals.

We make this need apparent by focussing on two important design choices where these tools differ: *value-sensitivity* and *flow-sensitivity*. It is instructive to highlight the import of these choices in terms of both the soundness and the precision of the resulting static analysis. Table 2.1 summarizes the design choices of the tools we consider, including ours.

	IccTA	*AD*	*DS*	*HD*
Value-sensitivity	no	yes	no	yes
Flow-sensitivity	yes	yes	no	partial

Table 2.1: Design Choices for Static Analysis Tools

2.2.1 Value-sensitivity

Value-sensitivity is the ability of a static analysis to approximate runtime values and use this information to improve precision, e.g., by skipping unreachable program branches [NNH99]. Concretely, consider the following code:

```
int x = 0;
for (int y = 0; y <= 10; y++) { x++; }
TelephonyManager tm = ...
String imei = tm.getDeviceId();
if (x == 0) { leak(imei); }
```

Though this code is perfectly safe, all the existing tools (IccTA, AmanDroid, and DroidSafe) will identify it as leaky. IccTA and DroidSafe conservatively assume all the program points to be potentially reachable. Even AmanDroid raises a false alarm for this code, though it internally implements a dedicated data-flow analysis [WROR14].

Besides this simple example, there are many reasons why real-world static analysis tools for Android applications should be value-sensitive to be practically useful. First, several features of Java and the Android APIs, most notably *reflection* and *dictionary-like* containers, e.g., intents and bundles, need value-sensitivity to be analysed precisely. Second, the loss of precision entailed by value-insensitivity may creep and interact poorly with other desirable features of the static analysis, e.g., *context-sensitivity*, which has been deemed as crucial by previous studies [ARF+14, GKP+15].

Context-sensitivity is the ability of the analysis to compute different static approximations upon different method calls. To understand why the benefits of context-sensitivity can be voided by value-insensitivity, consider the following method, where we assume to know a valid upper bound for the GPS location values:

```
void m (double x, double y) {
  if (x <= MAX_X && y <= MAX_Y)
    ...
  else
    leak("Invalid location:" + x + y);
}
```

Context-insensitive static analyses would detect a dangerous information flow whenever the method `m` is invoked at two different program points and one of these invocations provides the location of the device in the actual parameters, while the other one provides an invalid location. The reason is that the method `m` would be analysed only once, hence the static analysis would detect that both public and confidential values may reach a sink. Conversely, a context-sensitive analysis potentially has the ability to discriminate between the two methods invocations and be precise, but the lack of value-sensitivity would necessarily lead to the detection of a non-existent information flow.

Finally, it is worth noticing that value-sensitivity is crucial to support security-relevant, value-dependent security queries (e.g., “Is the credit card number sent on HTTP rather than on HTTPS?” or “Is the picture actually uploaded on Facebook, as opposed to some other untrusted website?”).

2.2.2 Flow-sensitivity

Flow-sensitivity is the ability of a static analysis to take the order of statements into account and compute different approximations at different program points [NNH99]. To understand its importance, consider the following code:

```
TelephonyManager tm = ...
String imei = tm.getDeviceId();
imei = new String("empty");
leak(imei);
```

Though the code above is safe, the flow-insensitive analysis implemented in DroidSafe will identify it as leaky, since the variable `imei` does contain secret information at some program point. Conversely, both FlowDroid and AmanDroid will correctly deem the program as safe.

Clearly, it is tempting to target a flow-sensitive information flow analysis tool to achieve a higher level of precision, but, as pointed out by the authors of DroidSafe [GKP^{+}15], flow-sensitivity is very hard to get right for Android applications, due to their extensive use of asynchronous callbacks. Both FlowDroid and AmanDroid suggest to tackle this problem by introducing a *dummy main method* emulating each possible interleaving of the callbacks defining the application lifecycle. Unfortunately, it is difficult to ensure that the dummy main method construction is accurate and comprehensive, which leads to missing malicious information flows [GKP$^+$15].

2.2.3 HornDroid

Our tool, HornDroid, targets a *sound* and *practical* information flow analysis for Android applications. We report on the design choices we made to hit the sweet spot between these two potentially conflicting requirements.

HornDroid implements a *value-sensitive* information flow analysis. As anticipated, value-sensitivity is crucial to support a practically useful analysis of real-world applications. The analysis implemented in HornDroid is reminiscent of *abstract interpretation*, whereby a computable abstract semantics over-approximates the operational semantics of a program. As it is customary for abstract interpretation, the design of the analysis is parametric with respect to the choice of a set of *abstract domains*, defining how runtime values are statically approximated: one can then fine-tune the precision of the analysis by testing different abstract domains. To ensure the scalability of our value-sensitive analysis, the abstract semantics implemented in HornDroid is based on *Horn clauses*, whose efficient resolution is supported by state-of-the-art SMT solvers [BMR12].

HornDroid performs a *flow-sensitive* information flow analysis on the registers employed by the Dalvik Virtual Machine, while implementing a *flow-insensitive* analysis for callback methods and heap locations. This is crucial to preserve the precision of the analysis, without sacrificing soundness. We already mentioned that previous studies highlighted that flow-sensitive analyses might quickly produce unsound results, due to the challenges of predicting all the possible orderings of the Android callbacks [GKP$^+$15]. Moreover, while carrying out the soundness proof for HornDroid, we realized that *static fields* are particularly delicate to treat in a flow-sensitive fashion. The reason is that static fields provide a way to implement a shared memory between otherwise memory-isolated components running in the same application. Given that the execution order of different Android components is extremely hard to predict, due to their callback-driven nature, it turns out that flow-insensitivity for static fields is in practice needed for soundness. Indeed, since static fields can be used to exchange pointers to heap locations, a sound flow-sensitive analysis for heap locations is in general hard to achieve. Our soundness proof, instead, confirms that flow-sensitivity can be implemented for the registers employed by the Dalvik Virtual Machine without missing any malicious information flow.

2.3 Operational Semantics

We base our technical development on μ-Dalvik$_A$, a formal model of the Android semantics obtained by extending the μ-Dalvik calculus [JMF12] with a complete characterisation of the activity-specific aspects of the Android platform [PS14].

2.3.1 Background and Scope

Android applications are developed in Java and then compiled to a custom bytecode format called *Dalvik*, which is run by the Dalvik Virtual Machine (DVM). Unlike Java VMs, which are stack machines, the DVM adopts a register-based architecture. Android applications are different from standard Java programs, since they are structured in *components* of four different types: activities, services, content providers and broadcast receivers [The16a]. These components represent distinct entry points of the Android framework into the application. Hence, the operational behaviour of an Android application does not simply amount to the sequential execution of its bytecode implementation, but it heavily relies on callbacks from the Android framework, as a reaction to user inputs, system events, or inter-component communication. Different Android components, either in the same application or from different applications, can communicate by exchanging *intents*, i.e., dictionary-like messaging objects. Intents may be sent either to a specific component (*explicit* intents) or to any component which declares the will of providing a given functionality (*implicit* intents).

In our formal model we consider Android applications consisting of activities only. We focus on activities, since a tested semantics is available for them and because they exhibit the most complicated lifecycle among all the component types [PS14]. Also, we only model intra-application communication based on explicit intents: implicit intents are mostly, if not only, used for inter-application messages. As we discuss in Section 2.5, μ-Dalvik$_A$ does not cover all the Android features supported by HornDroid: the purpose of μ-Dalvik$_A$ is to ensure that the design principles at the core of HornDroid are sound and that most of the Android-specific subtleties have been taken into due account.

2.3.2 Syntax

We write $(r_i)^{i \leq n}$ for the sequence $r_1, \ldots, r_n$. If the length of the sequence is immaterial, we just write r^* and we still let r_j stand for its j-th element. We represent the empty sequence with a dot $(\cdot)$. We let $r^*[j \mapsto r']$ be the sequence obtained from r^* by replacing its j-th element with r'. A *partial map* is a sequence of key-value bindings $(k_i \mapsto v_i)^*$, where all the keys k_i are pairwise distinct. Given a partial map M, let $dom(M)$ stand for the set of its keys and let $M(k) = v$ whenever the binding $k \mapsto v$ occurs in M. We identify partial maps which are identical up to the order of their key-value bindings.

Table 2.2 provides the syntax of μ-Dalvik$_A$ programs. It is an extension of the original μ-Dalvik syntax [JMF12] with a few additional statements modelling method calls to Android APIs used for inter-component communication.

A μ-Dalvik$_A$ program P is a sequence of classes cls^*, which in turn are defined by a class name c, a direct super-class c', some implemented interfaces c^*, and a number of fields fld^* and methods mtd^*. Field declarations $f : \tau$ include the field name f and its type τ, while method declarations $m : \tau^* \xrightarrow{n} \tau \ \{st^*\}$ include the method name m, the argument types τ^*, the return type τ, and the method body st^*. The annotation n on top of the

P	$::=$	cls^*
cls	$::=$	$\texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{\mathit{fld}^*; \mathit{mtd}^*\}$
τ_{prim}	$::=$	$\texttt{bool} \mid \texttt{int} \mid \dots$
τ	$::=$	$c \mid \tau_{prim} \mid \texttt{array}[\tau]$
fld	$::=$	$f : \tau$
mtd	$::=$	$m : \tau^* \xrightarrow{n} \tau\ \{st^*\}$
st	$::=$	$\texttt{goto}\ pc$
	$\mid$	$\texttt{move}\ \mathit{lhs}\ \mathit{rhs}$
	$\mid$	$\texttt{if}_{\otimes}\ r_1\ r_2\ \texttt{then}\ pc$
	$\mid$	$\texttt{unop}_{\odot}\ r_d\ r_s$
	$\mid$	$\texttt{binop}_{\oplus}\ r_d\ r_1\ r_2$
	$\mid$	$\texttt{new}\ r_d\ c$
	$\mid$	$\texttt{newarray}\ r_d\ r_l\ \tau$
	$\mid$	$\texttt{checkcast}\ r_s\ \tau$
	$\mid$	$\texttt{instof}\ r_d\ r_s\ \tau$
	$\mid$	$\texttt{invoke}\ r_o\ m\ r^*$
	$\mid$	$\texttt{sinvoke}\ c\ m\ r^*$
	$\mid$	$\texttt{return}$
	$\mid$	$\texttt{newintent}\ r_i\ c$
	$\mid$	$\texttt{put-extra}\ r_i\ r_k\ r_v$
	$\mid$	$\texttt{get-extra}\ r_i\ r_k\ \tau$
	$\mid$	$\texttt{start-activity}\ r_i$

r	$\in$	$\mathit{Registers}$
pc	$\in$	$\mathbb{N}$
$\oplus$	$::=$	$+ \mid - \mid \dots$
$\odot$	$::=$	$- \mid \neg \mid \dots$
$\otimes$	$::=$	$< \mid > \mid \dots$
prim	$::=$	$\texttt{true} \mid \texttt{false} \mid \dots$
lhs	$::=$	r
	$\mid$	$r[r]$
	$\mid$	$r.f$
	$\mid$	$c.f$
rhs	$::=$	lhs
	$\mid$	prim

Table 2.2: μ-Dalvik$_A$ Syntax

arrow tracks the number of local registers used by the method, which is statically known in Dalvik.

We briefly discuss below the statements of the language. An unconditional branch `goto` pc sets the program counter to pc. The statement `move` $\mathit{lhs}\ \mathit{rhs}$ moves the right-hand side rhs into the left-hand side lhs: here, lhs may be a register r, an array cell $r_1[r_2]$, an object field $r.f$, or a static field $c.f$; rhs may be any of these elements or a constant. A conditional branch $\texttt{if}_{\otimes}\ r_1\ r_2\ \texttt{then}\ pc$ compares the content of two registers r_1 and r_2 using the comparison operator $\otimes$ and sets the program counter to pc if the check is successful, otherwise it moves to the next instruction. We then have unary and binary operations, represented by $\texttt{unop}_{\odot}\ r_d\ r_s$ and $\texttt{binop}_{\oplus}\ r_d\ r_1\ r_2$ respectively, where r_d is the destination register where the result of the operation must be stored and the other registers contain the operands. Object creation is modelled by `new` $r_d\ c$, which creates an object of class c and stores a pointer to it in r_d; array creation is similarly handled by `newarray` $r_d\ r_l\ \tau$, where r_d is the destination register where the pointer to the new array must be stored, r_l contains the array length and τ specifies the type of the array cells. The type cast statement `checkcast` $r_s\ \tau$ checks whether the register r_s

contains a pointer to an object of type τ and it moves to the next instruction if this is the case, otherwise it stops the execution. The statement `instof` $r_d\ r_s\ \tau$ stores `true` in r_d if r_s points to an object of type τ, otherwise it stores `false`. A method invocation `invoke` $r_o\ m\ r^*$ calls the method m on the receiver object pointed by r_o, passing the values in the registers r^* as actual arguments. The invocation of static methods is modelled by `sinvoke` $c\ m\ r^*$. The `return` statement has no argument, rather there is a special register r_{ret} for holding return values: the return value must be moved to r_{ret} by the callee before calling `return`.

The last four statements are used to model inter-component communication. Intent creation is modelled by `newintent` $r_i\ c$, which creates an intent for the activity c and stores a pointer to it in r_i. The statement `put-extra` $r_i\ r_k\ r_v$ adds to the intent pointed by r_i a new key-value binding $k \mapsto v$, where k and v are the contents of r_k and r_v respectively. The statement `get-extra` $r_i\ r_k\ \tau$ retrieves from the intent pointed by r_i the value bound to key k, where k is the content of r_k, provided that this value has type τ. Finally, `start-activity` r_i sends the intent pointed by r_i, thus starting a new activity. Throughout the chapter, we only consider *well-formed* programs.

Definition 1. A program P is *well-formed* iff: (1) all its class names are pairwise distinct, (2) for each of its classes, all the field names are pairwise distinct, and (3) for each of its classes, all the method names are pairwise distinct.

Notice that the last condition of the definition above is not restrictive, since overloading resolution is performed at compile time in Java [Thec] and Dalvik bytecode thus identifies methods through their signature, rather than their name. In our formalism, we then suppose that method names are tagged with some distinctive information drawn from their signature, so that we can identify each method of a given class just by its name. Notice that two different classes can still define two methods with the same name, which is important to model dynamic dispatching.

From now on, we focus our attention on some well-formed program $P = \mathit{cls}^*$. Most of the definitions we present in the chapter depend on P, but we do not make this dependence explicit in the notation to keep it lighter.

2.3.3 Dalvik Semantics

Table 2.3 defines the semantic domains employed by the operational semantics of μ-Dalvik$_A$. Values include primitive values and *locations*, i.e., pointers to heap elements extended with an annotation λ. Annotations have no semantic import and are only needed for our static analysis: we will discuss their role in Section 2.4.

A *local configuration* $\Sigma = \alpha \cdot \pi \cdot H \cdot S$ represents the state of a specific activity. It includes a call stack α, a pending activity stack π, a heap H, and a static heap S. A call stack α is a list of *local states*, which is populated upon method invocation. Each local state includes: (1) a program point $pp = c, m, pc$, where c and m identify the invoked method,

Pointers	p	$\in$	$\mathit{Pointers}$
Program points	pp	$::=$	c, m, pc
Annotations	λ	$::=$	$pp \mid c \mid in(c)$
Locations	ℓ	$::=$	p_λ
Values	u, v	$::=$	$prim \mid \ell$
Registers	R	$::=$	$(r \mapsto v)^*$
Local states	L	$::=$	$\langle pp \cdot st^* \cdot R \rangle$
Call stacks	α	$::=$	$\varepsilon \mid L :: \alpha$
Pending activity stacks	π	$::=$	$\varepsilon \mid i :: \pi$
Objects	o	$::=$	$\{\!\mid c; (f_\tau \mapsto v)^* \mid\!\}$
Arrays	a	$::=$	$\tau[v^*]$
Intents	i	$::=$	$\{\!\mid @c; (k \mapsto v)^* \mid\!\}$
Memory blocks	b	$::=$	$o \mid a \mid i$
Heaps	H	$::=$	$(\ell \mapsto b)^*$
Static heaps	S	$::=$	$(c.f \mapsto v)^*$
Local configurations	Σ	$::=$	$\alpha \cdot \pi \cdot H \cdot S$

Table 2.3: μ-Dalvik$_A$ Semantic Domains

while pc points to the next instruction to execute; (2) a list of statements st^*, modelling the method body; and (3) a map R binding local registers to their current value.

A pending activity stack π is a list of intents, which are treated as (untyped) dictionaries in our formalism. As anticipated, for the sake of simplicity, we only consider explicit intents in the formalisation, i.e., intents which are meant to be delivered to an activity of a given class c: this class is specified after the 'at' symbol (@) in the intent syntax[1]. We use π to keep track of which activities have been started by the activity modelled by the local configuration.

Finally, a heap H is a mapping between locations and memory blocks, where each block is either an object, an array or an intent. Object fields are annotated with their static type, though we typically omit this annotation when it is unimportant. The static heap S simply binds static fields to their corresponding value.

The small-step operational semantics of μ-Dalvik$_A$ is defined by a reduction relation $\Sigma \rightsquigarrow \Sigma'$. Reduction takes place by fetching the next statement to execute, based on the program counter of the top-most local state of the call stack in Σ, and by running it to produce Σ'. The definition of the reduction relation is lengthy, but unsurprising, and it is given in § 3.1. The only point worth noticing here is that, when a new memory block is created, e.g., by `new`, the corresponding pointer to the heap is annotated with the program point c, m, pc where creation takes place.

[1]Extending the formalism to include implicit intents would not be difficult, but this would introduce non-determinism on the choice of the receiving activity, thus making the presentation harder to follow.

2.3.4 Activity Semantics

The operational behaviour of an activity does not depend only on its bytecode implementation, but also on external events, like user inputs and system callbacks. The event-driven nature of Android applications gives rise to highly non-deterministic executions, which are not trivial to approximate correctly by static analysis.

Formalizing Activities

We start by introducing a formal notion of activity.

Definition 2. A class *cls* is an *activity class* iff $cls = \texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{fld^*; mtd^*\}$ for some $c' \leq \texttt{Activity}$. An *activity* is an instance of an activity class. We stipulate that each activity has the following fields: (1) *finished*: a boolean flag stating whether the activity has finished or not; (2) *intent*: a pointer to the intent which started the activity; (3) *result*: a pointer to an intent storing the result of the activity computation; and (4) *parent*: a pointer to the parent activity, i.e., the activity which started the present one.

We require that each activity has a (possibly empty) set of *event handlers* for user inputs: given an activity class c, we let $handlers(c) = \{m_1, \dots, m_n\}$ be the set of the names of the methods of c which may be dispatched when some user input event occurs. We assume a set of activity states *ActStates* and a relation $Lifecycle \subseteq ActStates \times ActStates$ defining the state transitions admitted by the activity lifecycle [PS14]. We assume that each activity class c has a set of callbacks for each activity state s, whose names are returned by a function $cb(c, s)$; for the *running* state we let $cb(c, running) = handlers(c)$, i.e., when an activity is running, any callback set for user inputs may be dispatched.

We then extend the syntax of μ-Dalvik$_A$ with the elements in Table 2.4. A *frame* φ includes a location ℓ pointing to an activity, a corresponding activity state s, a pending activity stack π and a call stack α. Frames are organized in an *activity stack* Ω, modelling different activities executing in the same application: a single frame in Ω has the priority of execution and is underlined. A *configuration* Ψ includes an activity stack Ω, a heap H and a static heap S.

Activity states	s	$\in$	$ActStates$
Frames	φ	$::=$	$\langle \ell, s, \pi, \alpha \rangle \mid \underline{\langle \ell, s, \pi, \alpha \rangle}$
Activity stacks	Ω	$::=$	$\varphi \mid \varphi :: \Omega$
Configurations	Ψ	$::=$	$\Omega \cdot H \cdot S$

Convention: each activity stack Ω contains at most one active (underlined) frame.

Table 2.4: Extensions to the Syntax of μ-Dalvik$_A$

Reduction Rules

Before presenting the formal semantics, we need to introduce some additional definitions. We start with the notion of *callback stack*, identifying the admissible format of a call stack for new frames pushed on the activity stack upon the invocation of a callback from the Android system. Let $sign(c, m) = \tau^* \xrightarrow{n} \tau$ iff there exists a class cls_i such that $cls_i = \texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{fld^*; mtd^*, m : \tau^* \xrightarrow{n} \tau\ \{st^*\}\}$. Let then *lookup* stand for a *method lookup* function such that $lookup(c, m) = (c', st^*)$ iff: (1) c' is the class defining the method which is dispatched when m is invoked on an object of type c, and (2) st^* is the method body.

Definition 3. Given a location ℓ pointing to an activity of class c, we let $\alpha_{\ell.s}$ stand for an arbitrary *callback stack* for state s, i.e., any call stack $\langle c', m, 0 \cdot st^* \cdot R\rangle :: \varepsilon$, where $(c', st^*) = lookup(c, m)$ for some $m \in cb(c, s)$, $sign(c', m) = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ and:

$$R = ((r_i \mapsto \mathbf{0})^{i \leq loc}, r_{loc+1} \mapsto \ell, (r_{loc+1+j} \mapsto v_j)^{j \leq n}),$$

for some values $v_1, \ldots, v_n$ of the correct type $\tau_1, \ldots, \tau_n$.

In the definition, we let $\mathbf{0}$ be the default value for local registers. There is just one default value for registers in the model, since registers are untyped in Dalvik. In the following, it is also convenient to presuppose for each type τ the existence of a default value $\mathbf{0}_\tau$, used to initialize fields of type τ upon object creation.

A tricky aspect of the operational semantics of activities, which has never been formalized before, is the *serialization* of objects upon inter-component communication. Different activities may exchange objects using intents, but these objects are never passed by reference: rather, they are serialized at the sender side and a copy of them is created at the receiver side. The intent itself is serialized upon communication. We formalize this serialization routine by two mutually recursive functions $ser^H_{Val}(v) = (v', H')$ and $ser^H_{Blk}(b) = (b', H')$, returning a serialized copy of their argument and a new heap where all the pointers created in the serialization process have been instantiated correctly. We refer to Table 2.6 below for the definition of the two functions. Their definition uses a set of pointers Γ to keep track of which pointers have already been followed in the serialization process, so as to allow the serialization of memory blocks including self-references.

Finally, the operational semantics requires the following definition of *successful* call stack. A successful call stack is the call stack of an activity which has completed its computation.

Definition 4. A call stack α is *successful* if and only if $\alpha = \langle pp \cdot \texttt{return} \cdot R\rangle :: \varepsilon$ for some pp and R. We let $\overline{\alpha}$ range over successful call stacks.

Now we have all the ingredients to define the formal semantics of activities, which is given by the reduction rules in Table 2.5 and Table 2.6. As anticipated, the rules closely follow previous work by Payet and Spoto [PS14], which we extend to provide a more accurate account of inter-component communication by modelling value-passing based on a serialization routine. We give a short explanation of all the rules, we refer to [PS14] for a longer description.

Rule (A-Active) allows the execution of the statements in the active frame, using the reduction relation for local configurations described in Section 2.3.3. Rule (A-Deactivate) models the situation where the active frame has run up to completion: the frame loses priority and one of the other rules can be applied. Rule (A-Step) models the transition of the top-level activity from state s to one of its successors s' in the activity lifecycle: correspondingly, a new callback method is executed. Two side-conditions constrain the possible state transitions, based on the presence of pending activities to start and on whether the activity has finished or not.

Rule (A-Destroy) models the removal of a finished activity from the activity stack. Rule (A-Back) models the scenario where the user hits the back button on the Android device and the top-most activity gets finished by the system. Rule (A-Replace) corresponds to screen orientation changes: the foreground activity is destroyed and gets replaced by a fresh activity instance; notice that the new pointer to the heap is annotated with the class of the activity. Rule (A-Hidden) models the scenario where a new activity (the frame φ) has come to the foreground and hides a previously running activity, which gets stopped or destroyed by the system.

The starting of a new activity is modelled by rule (A-Start). The top-most activity is paused or stopped and there is some intent i to be sent to c: the intent is serialized and a new instance of c is pushed on the activity stack, setting its *intent* field to a pointer to the serialized copy of i and setting its *parent* field to a pointer to the activity which sent the intent. The pointer to the new activity is annotated with the class c, while the pointer to the serialized copy of the intent gets the annotation $in(c)$: again, this is needed just for the static analysis and will be discussed later. Notice that, if multiple activities need to be started, rule (A-Swap) allows a parent activity to substitute itself to a child activity on the top of the activity stack, so that rule (A-Start) can be applied again to fire the remaining intents. Finally, rule (A-Result) allows a finished activity in the foreground to return the result of its computation to the parent activity: the parent activity gets a serialized copy of the result and becomes active by executing a corresponding callback, bound to the *onActivityResult* state.

2.3.5 Examples

One reason why it is useful to have a formal semantics before devising a static analysis technique is to pinpoint corner cases which may potentially lead to unsound analysis results. We discuss two examples below.

Static Fields

Even though inter-component communication does not allow for the exchange of references, activities in the same application can still share memory by using static fields. This is apparent in the formal semantics, since the syntax of configurations Ψ contains a global

$$\text{(A-Active)}\quad \frac{\alpha \cdot \pi \cdot H \cdot S \rightsquigarrow \alpha' \cdot \pi' \cdot H' \cdot S'}{\Omega :: \underline{\langle \ell, s, \pi, \alpha \rangle} :: \Omega' \cdot H \cdot S \Rightarrow \Omega :: \underline{\langle \ell, s, \pi', \alpha' \rangle} :: \Omega' \cdot H' \cdot S'}$$

$$\text{(A-Deactivate)}\quad \Omega :: \underline{\langle \ell, s, \pi, \overline{\alpha} \rangle} :: \Omega' \cdot H \cdot S \Rightarrow \Omega :: \langle \ell, s, \pi, \overline{\alpha} \rangle :: \Omega' \cdot H \cdot S$$

$$\text{(A-Step)}\quad \frac{\begin{array}{c}(s, s') \in \mathit{Lifecycle} \qquad \pi \neq \varepsilon \Rightarrow (s, s') = (\mathit{running}, \mathit{onPause}) \\ H(\ell).\mathit{finished} = \mathtt{true} \Rightarrow (s, s') \in \{(\mathit{running}, \mathit{onPause}), (\mathit{onPause}, \mathit{onStop}), (\mathit{onStop}, \mathit{onDestroy})\}\end{array}}{\langle \ell, s, \pi, \overline{\alpha} \rangle :: \Omega \cdot H \cdot S \Rightarrow \underline{\langle \ell, s', \pi, \alpha_{\ell.s'} \rangle} :: \Omega \cdot H \cdot S}$$

$$\text{(A-Destroy)}\quad \frac{H(\ell).\mathit{finished} = \mathtt{true}}{\Omega :: \langle \ell, \mathit{onDestroy}, \pi, \overline{\alpha} \rangle :: \Omega' \cdot H \cdot S \Rightarrow \Omega :: \Omega' \cdot H \cdot S}$$

$$\text{(A-Back)}\quad \frac{H' = H[\ell \mapsto H(\ell)[\mathit{finished} \mapsto \mathtt{true}]]}{\langle \ell, \mathit{running}, \varepsilon, \overline{\alpha} \rangle :: \Omega \cdot H \cdot S \Rightarrow \langle \ell, \mathit{running}, \varepsilon, \overline{\alpha} \rangle :: \Omega \cdot H' \cdot S}$$

$$\text{(A-Replace)}\quad \frac{H(\ell) = \{\!|c; (f_\tau \mapsto v)^*, \mathit{finished} \mapsto u|\!\} \qquad o = \{\!|c; (f_\tau \mapsto \mathbf{0}_\tau)^*, \mathit{finished} \mapsto \mathtt{false}|\!\} \qquad H' = H, p_c \mapsto o}{\langle \ell, \mathit{onDestroy}, \pi, \overline{\alpha} \rangle :: \Omega \cdot H \cdot S \Rightarrow \underline{\langle p_c, \mathit{constructor}, \pi, \alpha_{p_c.\mathit{constructor}} \rangle} :: \Omega \cdot H' \cdot S}$$

$$\text{(A-Hidden)}\quad \frac{\varphi = \langle \ell, s, \pi, \overline{\alpha} \rangle \qquad s \in \{\mathit{onResume}, \mathit{onPause}\} \qquad (s', s'') \in \{(\mathit{onPause}, \mathit{onStop}), (\mathit{onStop}, \mathit{onDestroy})\}}{\varphi :: \Omega :: \langle \ell', s', \pi', \overline{\alpha}' \rangle :: \Omega' \cdot H \cdot S \Rightarrow \varphi :: \Omega :: \underline{\langle \ell', s'', \pi', \alpha_{\ell'.s''} \rangle} :: \Omega' \cdot H \cdot S}$$

$$\text{(A-Start)}\quad \frac{\begin{array}{c}s \in \{\mathit{onPause}, \mathit{onStop}\} \qquad i = \{\!|@c; (k \mapsto v)^*|\!\} \qquad \emptyset \vdash \mathit{ser}^H_{\mathit{Blk}}(i) = (i', H') \qquad p_c, p'_{in(c)} \notin \mathit{dom}(H, H') \\ o = \{\!|c; (f_\tau \mapsto \mathbf{0}_\tau)^*, \mathit{finished} \mapsto \mathtt{false}, \mathit{intent} \mapsto p'_{in(c)}, \mathit{parent} \mapsto \ell|\!\} \qquad H'' = H, H', p_c \mapsto o, p'_{in(c)} \mapsto i'\end{array}}{\langle \ell, s, i :: \pi, \overline{\alpha} \rangle :: \Omega \cdot H \cdot S \Rightarrow \underline{\langle p_c, \mathit{constructor}, \varepsilon, \alpha_{p_c.\mathit{constructor}} \rangle} :: \langle \ell, s, \pi, \overline{\alpha} \rangle :: \Omega \cdot H'' \cdot S}$$

$$\text{(A-Swap)}\quad \frac{\begin{array}{c}\varphi' = \langle \ell', \mathit{onPause}, \varepsilon, \overline{\alpha}' \rangle \qquad H(\ell').\mathit{finished} = \mathtt{true} \\ \varphi = \langle \ell, s, i :: \pi, \overline{\alpha} \rangle \qquad s \in \{\mathit{onPause}, \mathit{onStop}\} \qquad H(\ell').\mathit{parent} = \ell\end{array}}{\varphi' :: \varphi :: \Omega \cdot H \cdot S \Rightarrow \varphi :: \varphi' :: \Omega \cdot H \cdot S}$$

$$\text{(A-Result)}\quad \frac{\begin{array}{c}\varphi' = \langle \ell', \mathit{onPause}, \varepsilon, \overline{\alpha}' \rangle \qquad H(\ell').\mathit{finished} = \mathtt{true} \qquad \varphi = \langle \ell, s, \varepsilon, \overline{\alpha} \rangle \qquad s \in \{\mathit{onPause}, \mathit{onStop}\} \\ H(\ell').\mathit{parent} = \ell \qquad \emptyset \vdash \mathit{ser}^H_{\mathit{Val}}(H(\ell').\mathit{result}) = (\ell'', H') \qquad H'' = (H, H')[\ell \mapsto H(\ell)[\mathit{result} \mapsto \ell'']]\end{array}}{\varphi' :: \varphi :: \Omega \cdot H \cdot S \Rightarrow \underline{\langle \ell, s, \varepsilon, \alpha_{\ell.\mathit{onActivityResult}} \rangle} :: \varphi' :: \Omega \cdot H'' \cdot S}$$

Conventions: the activity stack on the left-hand side does not contain underlined frames, but for the first two rules.

Table 2.5: Reduction Relation for Configurations ($\Omega \cdot H \cdot S \Rightarrow \Omega' \cdot H' \cdot S'$), additional definitions are in Table 2.6.

$$\Gamma \vdash ser^{H}_{Val}(prim) = (prim, \cdot) \qquad \frac{p_\lambda \in \Gamma}{\Gamma \vdash ser^{H}_{Val}(p_\lambda) = (\nu(p_\lambda), \cdot)}$$

$$\frac{p_\lambda \notin \Gamma \qquad \Gamma \cup \{p_\lambda\} \vdash ser^{H}_{Blk}(H(p_\lambda)) = (b, H'') \qquad H' = H'', \nu(p_\lambda) \mapsto b}{\Gamma \vdash ser^{H}_{Val}(p_\lambda) = (\nu(p_\lambda), H')}$$

$$\frac{\forall i \in [1,n] : \Gamma \vdash ser^{H}_{Val}(v_i) = (u_i, H_i) \qquad H' = H_1, \dots, H_n}{\Gamma \vdash ser^{H}_{Blk}(\tau[(v_i)^{i \leq n}]) = (\tau[(u_i)^{i \leq n}], H')}$$

$$\frac{\forall i \in [1,n] : \Gamma \vdash ser^{H}_{Val}(v_i) = (u_i, H_i) \qquad H' = H_1, \dots, H_n}{\Gamma \vdash ser^{H}_{Blk}(\{\!| c'; (f_i \mapsto v_i)^{i \leq n} |\!\}) = (\{\!| c'; (f_i \mapsto u_i)^{i \leq n} |\!\}, H')}$$

$$\frac{\forall i \in [1,n] : \Gamma \vdash ser^{H}_{Val}(v_i) = (u_i, H_i) \qquad H' = H_1, \dots, H_n}{\Gamma \vdash ser^{H}_{Blk}(\{\!| @c'; (k_i \mapsto v_i)^{i \leq n} |\!\}) = (\{\!| @c'; (k_i \mapsto u_i)^{i \leq n} |\!\}, H')}$$

Conventions: in the serialization rules we assume the existence of a function $\nu(\_)$ assigning to each pointer a fresh pointer with the same annotation, used to store the result of the serialization.

Table 2.6: Additional definitions and rules for Table 2.5

static heap S, which can be accessed by using publicly known names of static fields. We then observe that the order of execution of different activities, or even different callbacks inside the same activity, is tough to predict: for instance, the rules in Table 2.5 highlight that even activities which are not on the top of the activity stack may become active and execute callbacks by rule (A-Hidden). Also, the same callback may be executed multiple times, since the Android system may routinely recreate an activity due to user activities (e.g., screen orientation changes), which cannot be known statically, as modelled by rule (A-Replace).

The implication on the static analysis is that it is extremely challenging to implement flow-sensitivity on accesses to static fields without producing unsound results. Furthermore, given that static fields may be used to share pointers to heap locations, flow-sensitivity for heap accesses is also hard to achieve. Since we target soundness in this work, the static analysis we devise in the next section is flow-insensitive on both static fields and heap locations.

Serialization

Rule (A-Start) of the operational semantics highlights that intents are serialized upon inter-component communication. This means that, when a parent activity starts a child activity, the latter operates on a copy of the intent sent by the former and not on the same intent.

The implication on static analysis is that, although the callback bound to the *onActivityResult* state of the parent activity is always executed after the construction of the child activity, no change to the intent done by the child activity should overwrite the original over-approximation of the intent computed for the parent activity when a result is returned to it. This applies to any object which is serialized with the intent. The static analysis in the next section provides a conservative over-approximation of this behaviour.

2.4 Static Analysis

The static analysis we propose works by translating an input program P into a corresponding *abstract* program Δ, i.e., a set of Horn clauses modelling an over-approximation of its semantics. By feeding these clauses to an automated theorem prover and by showing the unsatisfiability of an appropriate logical formula, we can prove that some set of undesired configurations is never reached by P.

2.4.1 Overview

The analysis is based on the syntactic categories in Table 2.7. We start by discussing how values are approximated. We presuppose the existence of an arbitrary set of abstract domains used to approximate primitive values: for each primitive value *prim*, we assume that there exists a corresponding abstraction $\widehat{prim}$, e.g., integer numbers could be approximated by their sign. Locations of the form $\ell = p_\lambda$, instead, are abstracted into their annotation λ. An *abstract value* $\hat{v}$ is a set of elements drawn from either the abstract domains or the set of annotations.

The different forms of annotations λ provide insight on different aspects of the static analysis. Program point annotations $pp = c, m, pc$ are used to represent pointers to memory blocks instantiated using the statements `new`, `newarray` and `newintent`: by abstracting these elements with the program point where they are created, we implement a *plain-object-sensitive* static analysis [SBL11b]. We chose it because it is well-understood and convenient to both formalize and present. Class name annotations c, instead, are used to represent activities in an object-insensitive way: different activities of the same class c are all abstracted by the annotation c, since it is generally hard to discriminate between different activity instances statically. Finally, we use the annotation $in(c)$ to abstract all the intents which are used to start an activity of class c.

Coming to memory blocks, our analysis is field-sensitive on objects, but field-insensitive on both arrays and intents. It is easier to implement field-sensitivity for objects, since field names are statically known in Java. Implementing field-sensitivity for arrays would require precise information on array bounds and indexes; intents, instead, would need an accurate string analysis, to deal with their dictionary-like programming patterns. It would be possible to leverage existing proposals [DDA11] to implement a more precise analysis in terms of field-sensitivity, but we propose a more straightforward framework here to focus on the Android-specific aspects of the analysis. Notice that, just like the objects they approximate, abstract objects $\hat{o}$ feature type annotations on their fields, which are omitted when unimportant.

Facts	f	$::=$	
Abs. registers		$\mid$	$\mathsf{R}_{\mathsf{pp}}(t^* \,;\, t^*)$
Abs. heap entries		$\mid$	$\mathsf{H}(t, t')$
Abs. static fields		$\mid$	$\mathsf{S}_{\mathsf{c,f}}(t)$
Abs. right-hand sides		$\mid$	$\mathsf{RHS}_{\mathsf{pp}}(t)$
Abs. results		$\mid$	$\mathsf{Res}_{\mathsf{c,m}}(t^* \,;\, t)$
Abs. pending activities		$\mid$	$\mathsf{I}(t, t')$
Set membership		$\mid$	$t \in t'$
Subtyping		$\mid$	$t \leq t'$
Horn clauses		$\mid$	$\forall x^*. \bigwedge_i \mathsf{f}_i \implies \mathsf{f}$
Abs. programs	Δ	$::=$	$\{\mathsf{f}_1, \dots, \mathsf{f}_n\}$
Abs. values	$\hat{u}, \hat{v}$	$::=$	$\emptyset \mid \{\widehat{prim}\} \mid \{\lambda\} \mid \hat{v} \cup \hat{v}$
Abs. objects	$\hat{o}$	$::=$	$\{\!\mid c; (f_\tau \mapsto \hat{v})^* \mid\!\}$
Abs. arrays	$\hat{a}$	$::=$	$\tau[\hat{v}]$
Abs. intents	$\hat{\imath}$	$::=$	$\{\!\mid @c; \hat{v} \mid\!\}$
Abs. mem. blocks	$\hat{b}$	$::=$	$\hat{o} \mid \hat{a} \mid \hat{\imath}$
Variables	x, y	$\in$	Vars
Constants	k	$::=$	$\hat{v} \mid \hat{b} \mid \tau \mid \lambda$
Terms	t	$::=$	$\mathsf{k} \mid x \mid \mathit{in}(t)$

Table 2.7: Abstract Domains and Analysis Facts

Abstract values and abstract memory blocks, plus all the types available in the analysed program and the annotations, determine a universe of *constants*, ranged over by k. A *term* t is either a constant k, a variable x drawn from a denumerable set Vars disjoint from the set of constants, or an expression of the form $\mathit{in}(t')$ for some term t'. The set of terms is used to define the syntax of *facts* f, logical formulas built on selected predicate symbols used by the analysis.

The fact $\mathsf{R}_{\mathsf{c,m,pc}}(\hat{u}^* \,;\, \hat{v}^*)$ states that, whenever the method m of class c is invoked with some arguments over-approximated by $\hat{u}^*$, the state of the local registers at the pc-th statement is over-approximated by $\hat{v}^*$. The syntax of the fact highlights that: (1) the analysis is flow-sensitive for register values, since it computes different static approximations at different program points, and (2) method invocations are handled in a context-sensitive way, where the notion of context coincides with the (abstraction of) the actual arguments supplied to the method upon invocation. The fact $\mathsf{H}(\lambda, \hat{b})$ states that some location p_λ refers to a heap element storing a memory block over-approximated by $\hat{b}$ at some point of the program execution. Notice that the fact does not contain any program point information, i.e., the analysis is flow-insensitive for heap locations, which is important for soundness (see Section 2.3.5). Similarly, the fact $\mathsf{S}_{\mathsf{c,f}}(\hat{v})$ states that the static field f

of class c contains a value which is over-approximated by $\hat{v}$ at some point of the program execution. The fact $\mathsf{RHS}_{\mathsf{pp}}(\hat{v})$ states that the right-hand side of the `move` statement at program point pp evaluates to a value over-approximated by $\hat{v}$. The fact $\mathsf{Res}_{\mathsf{c,m}}(\hat{u}^* \,;\, \hat{v})$ states that, whenever the method m of class c is invoked with some arguments over-approximated by $\hat{u}^*$, its return value is over-approximated by $\hat{v}$. The fact $\mathsf{I}(c, \hat{i})$ tracks that an activity of class c has sent an intent which is over-approximated by $\hat{i}$. We then have set membership facts $t \in t'$ and subtyping facts $\tau \leq \tau'$ with the apparent meaning.

Finally, Horn clauses define the abstract semantics of programs. A Horn clause has the form:

$$\forall x_1, \ldots, \forall x_m . \mathsf{f}_1 \wedge \ldots \wedge \mathsf{f}_n \implies \mathsf{f},$$

where all the variables of $\mathsf{f}_1, \ldots, \mathsf{f}_n, \mathsf{f}$ belong to $\{x_1, \ldots, x_m\}$ and each variable of f occurs among the variables of $\mathsf{f}_1, \ldots, \mathsf{f}_n$. Since most of the Horn clauses we present do not make use of constants, to improve readability we omit the universal quantifiers in front of Horn clauses and we just represent each variable occurring therein with a constant of the expected type. The few exceptions where constants are actually used are disambiguated using a sans serif font, e.g., we use c to denote the constant corresponding to the activity class c specifically, rather than some universally quantified variable standing for an arbitrary activity class. We let an underscore (_) stand for any syntactic element occurring in a Horn clause which is not significant to understanding.

2.4.2 Analysis Specification

Abstract Semantics of Dalvik

We start by presenting the abstract evaluation rules for right-hand sides, which are simple and provide a good intuition on how the static analysis works. These rules are given in Table 2.8.

$$\langle\!\langle prim \rangle\!\rangle_{pp} = \{\mathsf{RHS}_{\mathsf{pp}}(\{\widehat{prim}\})\} \qquad \langle\!\langle r_i \rangle\!\rangle_{pp} = \{\mathsf{R}_{\mathsf{pp}}(\_\,;\, \hat{v}^*) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{v}_i)\}$$

$$\langle\!\langle c.f \rangle\!\rangle_{pp} = \{\mathsf{S}_{\mathsf{c,f}}(\hat{v}) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{v})\}$$

$$\langle\!\langle r_i.f \rangle\!\rangle_{pp} = \{\mathsf{R}_{\mathsf{pp}}(\_\,;\, \hat{v}^*) \wedge \lambda \in \hat{v}_i \wedge \mathsf{H}(\lambda, \{\!|c; (f' \mapsto \hat{v}')^*, f \mapsto \hat{u}|\!\}) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{u})\}$$

$$\langle\!\langle r_i[r_j] \rangle\!\rangle_{pp} = \{\mathsf{R}_{\mathsf{pp}}(\_\,;\, \hat{v}^*) \wedge \lambda \in \hat{v}_i \wedge \mathsf{H}(\lambda, \tau[\hat{u}]) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{u})\}$$

Table 2.8: Abstract Evaluation of Right-hand Sides

To abstract a primitive value $prim$ at any program point pp, we pick the corresponding element $\widehat{prim}$ from the underlying abstract domain. To abstract the content of the register r_i at program point pp, we take the fact $\mathsf{R}_{\mathsf{pp}}(\_\,;\, \hat{v}^*)$ and we return the i-th abstract value $\hat{v}_i$. To abstract the content of a static field $c.f$ at any program point, we take any fact $\mathsf{S}_{\mathsf{c,f}}(\hat{v})$ and we return the abstract value $\hat{v}$. Abstracting the content of the field f of an object at program point pp is slightly more complicated: if the pointer to the object is stored in the register r_i, we pick the i-th abstract value $\hat{v}_i$ from the fact $\mathsf{R}_{\mathsf{pp}}(\_\,;\hat{v}^*)$ modelling the state of the registers at pp; then, if $\hat{v}_i$ contains any pointer abstraction λ, we use it to match a corresponding abstract heap entry $\mathsf{H}(\lambda, \hat{o})$ and we return the value of the field f of the abstract object $\hat{o}$ contained therein. We similarly abstract the content of array cells: just notice that, since the representation of arrays is field-insensitive, the index of the cell does not play any role in the static analysis.

The rules for abstracting a right-hand side are useful to define the abstract semantics of the `move` statement. Other statements require some additional definitions. First, for each comparison operator $\oslash$ and each primitive operation $\odot, \oplus$ of the concrete semantics, we presuppose the existence of a corresponding abstract operation $\hat{\oslash}$, $\hat{\odot}$ and $\hat{\oplus}$ defined over the elements of the appropriate abstract domain. Then, given an abstract memory block $\hat{b}$, we define a function $\widehat{\textit{get-type}}(\hat{b})$ as follows:

$$\widehat{\textit{get-type}}(\hat{b}) = \begin{cases} c & \text{if } \hat{b} = \{\!|c; (f \mapsto \hat{v})^*|\!\} \\ \texttt{array}[\tau] & \text{if } \hat{b} = \tau[\hat{v}] \\ \texttt{Intent} & \text{if } \hat{b} = \{\!|@c; \hat{v}|\!\} \end{cases}$$

Finally, we assume a function $\widehat{\textit{lookup}}(m)$, which returns the set of classes which define (or inherit) a method called m.

With these definitions, we are ready to introduce the abstract semantics of statements. The idea is to define, for each possible form of statement st, a translation $(\!|st|\!)_{pp}$ into a set of Horn clauses, which over-approximate the semantics of st at program point pp. The full formal semantics of the translation is given in Table 2.9 and explained below.

The rule for `goto` pc' propagates the state of the registers at the current program counter pc to pc'. The rule for $\texttt{if}_{\oslash}$ r_i r_j `then` pc' propagates the state of the registers at the current program counter pc either to pc' or to $pc + 1$, based on the outcome of a comparison $\hat{\oslash}$ between the abstract values $\hat{v}_i$ and $\hat{v}_j$ approximating the content of registers r_i and r_j respectively: both branches may be enabled, as the result of an over-approximation of the contents of the registers. The two rules for unary and binary operations just employ the appropriate abstract operation to update the approximation of the content of the destination register r_d. The four rules for the `move` statement rely on the auxiliary rules for abstracting a right-hand side we introduced before: these rules store their result in a RHS fact, which occurs in the premises of the Horn clause used to update the abstraction of the left-hand side. The most interesting point to notice here is that field-sensitivity or its absence has an import on how fields are updated: for objects, we replace the old value of the field with the new one; for arrays and intents, instead, we add the new value to the old approximation, since their abstraction over-approximates the content of the entire data structure, rather than just the single element which is updated. The rules for `instof` and `checkcast` use the $\widehat{\textit{get-type}}$ function previously defined.

$$
\begin{aligned}
&\llparenthesis \texttt{goto}\ pc' \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc'}}(_\,;\,\hat{v}^*)\}\\
&\llparenthesis \texttt{if}_{\otimes}\ r_i\ r_j\ \texttt{then}\ pc' \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \hat{v}_i \mathbin{\hat{\otimes}} \hat{v}_j \implies \mathsf{R}_{\mathsf{c,m,pc'}}(_\,;\,\hat{v}^*)\} \cup\\
& &&\quad \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \neg(\hat{v}_i \mathbin{\hat{\otimes}} \hat{v}_j) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*)\}\\
&\llparenthesis \texttt{binop}_{\oplus}\ r_d\ r_i\ r_j \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*[d \mapsto \hat{v}_i \mathbin{\hat{\oplus}} \hat{v}_j])\}\\
&\llparenthesis \texttt{unop}_{\odot}\ r_d\ r_i \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*[d \mapsto \hat{\odot}\, \hat{v}_i])\}\\
&\llparenthesis \texttt{move}\ r_d\ rhs \rrparenthesis_{pp} &&= \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}') \wedge \mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*[d \mapsto \hat{v}'])\} \cup \langle\!\langle rhs \rangle\!\rangle_{pp}\\
&\llparenthesis \texttt{move}\ r_a[r_{idx}]\ rhs \rrparenthesis_{pp} &&= \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \lambda \in \hat{v}_a \wedge \mathsf{H}(\lambda, \tau[\hat{v}']) \implies \mathsf{H}(\lambda, \tau[\hat{v}' \cup \hat{v}''])\} \cup\\
& &&\quad \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*)\} \cup \langle\!\langle rhs \rangle\!\rangle_{pp}\\
&\llparenthesis \texttt{move}\ r_o.f\ rhs \rrparenthesis_{pp} &&= \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \lambda \in \hat{v}_o \wedge \mathsf{H}(\lambda, \{\!| c'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}' |\!\}) \implies\\
& &&\quad \mathsf{H}(\lambda, \{\!| c'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'') |\!\})\} \cup\\
& &&\quad \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*)\} \cup \langle\!\langle rhs \rangle\!\rangle_{pp}\\
&\llparenthesis \texttt{move}\ c'.f\ rhs \rrparenthesis_{pp} &&= \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}') \implies \mathsf{S}_{\mathsf{c',f}}(\hat{v}')\} \cup \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies\\
& &&\quad \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*)\} \cup \langle\!\langle rhs \rangle\!\rangle_{pp}\\
&\llparenthesis \texttt{instof}\ r_d\ r_s\ \tau \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \lambda \in \hat{v}_s \wedge \mathsf{H}(\lambda, \hat{b}) \wedge \widehat{\textit{get-type}}(\hat{b}) \leq \tau \implies\\
& &&\quad \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*[d \mapsto \widehat{\texttt{true}}])\} \cup\\
& &&\quad \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \lambda \in \hat{v}_s \wedge \mathsf{H}(\lambda, \hat{b}) \wedge \widehat{\textit{get-type}}(\hat{b}) \not\leq \tau \implies\\
& &&\quad \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*[d \mapsto \widehat{\texttt{false}}])\}\\
&\llparenthesis \texttt{checkcast}\ r_s\ \tau \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \lambda \in \hat{v}_s \wedge \mathsf{H}(\lambda, \hat{b}) \wedge \widehat{\textit{get-type}}(\hat{b}) \leq \tau \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*)\}\\
&\llparenthesis \texttt{invoke}\ r_o\ m'\ (r_{i_j})^{j \leq n} \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \lambda \in \hat{v}_o \wedge \mathsf{H}(\lambda, \{\!| c'; (f \mapsto \hat{u})^* |\!\}) \wedge c' \leq \mathsf{c}'' \implies\\
& &&\quad \mathsf{R}_{\mathsf{c'',m',0}}((\hat{v}_{i_j})^{j \leq n}\,;\,(\hat{\mathbf{0}}_k)^{k \leq loc}, (\hat{v}_{i_j})^{j \leq n}) \mid c'' \in \widehat{\textit{lookup}}(m') \wedge\\
& &&\quad \textit{sign}(c'', m') = (\tau_j)^{j \leq n} \xrightarrow{loc} \tau\} \cup\\
& &&\quad \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \lambda \in \hat{v}_o \wedge \mathsf{H}(\lambda, \{\!| c'; (f \mapsto \hat{u})^* |\!\}) \wedge c' \leq \mathsf{c}'' \wedge\\
& &&\quad \mathsf{Res}_{\mathsf{c'',m'}}((\hat{v}_{i_j})^{j \leq n}\,;\,\hat{v}'_{ret}) \implies\\
& &&\quad \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*[ret \mapsto \hat{v}'_{ret}]) \mid c'' \in \widehat{\textit{lookup}}(m')\}\\
&\llparenthesis \texttt{sinvoke}\ c'\ m'\ (r_{i_j})^{j \leq n} \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c',m',0}}((\hat{v}_{i_j})^{j \leq n}\,;\,(\hat{\mathbf{0}}_k)^{k \leq loc}, (\hat{v}_{i_j})^{j \leq n}) \mid\\
& &&\quad \textit{sign}(c', m') = (\tau_j)^{j \leq n} \xrightarrow{loc} \tau\} \cup\\
& &&\quad \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \mathsf{Res}_{\mathsf{c',m'}}((\hat{v}_{i_j})^{j \leq n}\,;\,\hat{v}'_{ret}) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*[ret \mapsto \hat{v}'_{ret}])\}\\
&\llparenthesis \texttt{new}\ r_d\ c' \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{H}(\mathsf{pp}, \{\!| c'; (f \mapsto \hat{\mathbf{0}}_{\tau})^* |\!\}\} \cup\\
& &&\quad \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*[d \mapsto \mathsf{pp}])\}\\
&\llparenthesis \texttt{newarray}\ r_d\ r_l\ \tau \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{H}(\mathsf{pp}, \tau[\hat{\mathbf{0}}_{\tau}])\} \cup\\
& &&\quad \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*[d \mapsto \mathsf{pp}])\}\\
&\llparenthesis \texttt{return} \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(\hat{v}^*_{call}\,;\,\hat{v}^*) \implies \mathsf{Res}_{\mathsf{c,m}}(\hat{v}^*_{call}\,;\,\hat{v}_{ret})\}\\
&\llparenthesis \texttt{start-activity}\ r_i \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \lambda \in \hat{v}_i \wedge \mathsf{H}(\lambda, \{\!| @c'; \hat{u} |\!\}) \implies \mathsf{I}(\mathsf{c}, \{\!| @c'; \hat{u} |\!\})\} \cup\\
& &&\quad \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*)\}\\
&\llparenthesis \texttt{newintent}\ r_d\ c' \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{H}(\mathsf{pp}, \{\!| @c'; \emptyset |\!\})\} \cup\\
& &&\quad \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*[d \mapsto \mathsf{pp}])\}\\
&\llparenthesis \texttt{put-extra}\ r_i\ r_k\ r_j \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \lambda \in \hat{v}_i \wedge \mathsf{H}(\lambda, \{\!| @c'; \hat{v}' |\!\}) \implies \mathsf{H}(\lambda, \{\!| @c'; \hat{v}' \cup \hat{v}_j |\!\})\} \cup\\
& &&\quad \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*)\}\\
&\llparenthesis \texttt{get-extra}\ r_i\ r_k\ \tau \rrparenthesis_{pp} &&= \{\mathsf{R}_{\mathsf{pp}}(_\,;\,\hat{v}^*) \wedge \lambda \in \hat{v}_i \wedge \mathsf{H}(\lambda, \{\!| @c'; \hat{v}' |\!\}) \implies \mathsf{R}_{\mathsf{c,m,pc+1}}(_\,;\,\hat{v}^*[ret \mapsto \hat{v}'])\}
\end{aligned}
$$

Table 2.9: Abstract Semantics of μ-Dalvik$_A$ - Statements (let $pp = c, m, pc$)

The rule for `invoke` is the most complicated one, since it has to deal with dynamic dispatching. The challenge here is that the name of the invoked method is statically known from the syntax of the statement, but the method implementation is not, since it depends on the runtime type of the receiver object, an information which is only over-approximated when solving the Horn clauses, rather than when generating them. We then use the method name and the number of arguments passed upon invocation to narrow the set of possible classes of the receiver object, using the functions $\widehat{lookup}$ and *sign*, and we generate one Horn clause for each of them. We then rely on subtyping to make the analysis precise, by imposing that a Horn clause generated for class c'' can only be fired if the class c' of (the abstraction of) the receiver object is a subtype of c''. Besides implementing a sound approximation of the dynamic dispatching mechanism, the rule for `invoke` generates additional Horn clauses used to propagate the abstraction of the method return value from the callee to the caller: this is done by using a Res fact, which is introduced by a `return` statement in the implementation of the callee, as we discuss below. The rule for static method invocation follows a similar logic, but it is significantly simpler, due to the lack of dynamic dispatching on static calls.

The rules for object and array creation create a new abstract heap entry $\mathsf{H}(\lambda, \hat{b})$, where λ is the current program point and $\hat{b}$ is the abstraction of a freshly initialized object/array. The rule for `return` introduces a Res fact, storing an over-approximation of the method return value; notice that the arguments $\hat{v}^*_{call}$ supplied upon method invocation are propagated in the Res fact, which is important to implement context-sensitivity, i.e., to propagate the result to the right caller. The rule for `start-activity` tracks that the present activity c has sent an intent: an over-approximation of the intent is propagated from the corresponding abstract heap entry into the I fact modelling the presence of a pending activity which is about to start. The last rules for managing intents should be easy to understand, based on the intuitions given for the other rules.

Abstract Semantics of Activities

We can finally introduce the abstract semantics of activities. Intuitively, it is defined by: (1) the Horn clauses produced by translating each statement in the bytecode, and (2) a small set of bytecode-independent Horn clauses, abstracting the event-driven behaviour of activities. This is formalized next.

Definition 5. Let $P = (cls_i)^{i \leq n}$ be a program where $cls_i = \texttt{cls}\ c_i \leq c'\ \texttt{imp}\ c^*\ \{fld^*; (mtd_j)^{j \leq h_i}\}$ and $mtd_j = m_j : \tau^* \xrightarrow{loc} \tau\ \{(st_k)^{k \leq s_{ij}}\}$, we let $(\!|P|\!)$ be defined as follows:

$$(\!|P|\!) = \bigcup_{i \leq n, j \leq h_i, k \leq s_{ij}} (\!|st_k|\!)_{c_i, m_j, k} \cup \mathcal{R},$$

where $\mathcal{R}$ stands for the union of all the rules in Table 2.10.

We explain the rules from Table 2.10. Rule *Cbk* simulates the invocation of a callback: since we do not approximate the activity state in the abstract semantics, any callback

$$
\begin{array}{lcl}
\mathit{Cbk} & = & \{\mathsf{H}(c, \{\!|c; (f \mapsto \_)^{*}|\!\}) \land c \leq \mathsf{c}' \implies \mathsf{R}_{\mathsf{c}',\mathsf{m},0}((\top_{\tau_j})^{j \leq n};\, (\hat{\mathbf{0}}_k)^{k \leq loc}, c, (\top_{\tau_j})^{j \leq n}) \mid \\
 & & c' \text{ is an activity class} \land \exists s : m \in cb(c', s) \land \mathit{sign}(c', m) = \tau_1, \dots, \tau_n \xrightarrow{loc} \tau\} \\
\mathit{Fin} & = & \{\mathsf{H}(c, \{\!|c; (f \mapsto \_)^{*}, \mathit{finished} \mapsto \_|\!\}) \implies \mathsf{H}(c, \{\!|c; (f \mapsto \_)^{*}, \mathit{finished} \mapsto \top_{\texttt{bool}}|\!\})\} \\
\mathit{Rep} & = & \{\mathsf{H}(c, \{\!|c; (f_\tau \mapsto \_)^{*}|\!\}) \implies \mathsf{H}(c, \{\!|c; (f_\tau \mapsto \hat{\mathbf{0}}_\tau)^{*}|\!\})\} \\
\mathit{Act} & = & \{\mathsf{I}(c', \{\!|@c; \hat{v}|\!\})) \implies \mathsf{H}(\mathit{in}(c), \{\!|@c; \hat{v}|\!\})\} \cup \\
 & & \{\mathsf{I}(c', \{\!|@c; \hat{v}|\!\})) \implies \mathsf{H}(c, \{\!|c; (f_\tau \mapsto \hat{\mathbf{0}}_\tau)^{*}, \\
 & & \mathit{finished} \mapsto \widehat{\texttt{false}}, \mathit{parent} \mapsto c', \mathit{intent} \mapsto \mathit{in}(c)|\!\})\} \\
\mathit{Res} & = & \{\mathsf{H}(c', \{\!|c'; (f' \mapsto \_)^{*}, \mathit{parent} \mapsto c, \mathit{result} \mapsto \lambda|\!\} \land \mathsf{H}(c, \{\!|c; (f \mapsto \_)^{*}, \mathit{result} \mapsto \_|\!\} \implies \\
 & & \mathsf{H}(c, \{\!|c; (f \mapsto \_)^{*}, \mathit{result} \mapsto \lambda|\!\}\} \\
\mathit{Sub} & = & \{\tau \leq \tau' \mid \tau \leq \tau' \text{ is a valid subtyping judgement}\}
\end{array}
$$

Table 2.10: Abstract Semantics of μ-Dalvik$_A$ - Activity Rules

method bound to a state s of the activity lifecycle may be non-deterministically dispatched; the statically unknown arguments supplied to the callback are abstracted by the top element ($\top$) of the abstract domain associated to their type, which is a sound over-approximation of any value of that type. Rule *Fin* tracks updates to the *finished* field of an activity in the abstract semantics: since it is hard to track whether an activity has finished or not statically, the rule sets the field to the top element of the abstract domain used to represent boolean value ($\top_{\texttt{bool}}$). Rule *Rep* approximates the behaviour of rule (A-Replace) of the concrete semantics: the activity fields may be reset to their default abstract value as the result of a screen orientation change.

Rule *Act* represents the starting of a new activity. If an intent has been sent by an activity of class c' to start an activity of class c, we introduce: (1) a new abstract heap entry to bind an abstraction of the intent to $in(c)$, and (2) a new abstract heap entry to bind an abstraction of the started activity to c. No serialization happens in the abstract semantics: if an intent is used to send an object in the concrete semantics, a reference to the corresponding abstract object is sent in our abstraction. This is sound, since our analysis is flow-insensitive on heap values, hence no over-approximation of the original object is ever lost as the result of an update to the heap at the receiver side. We then have rule *Res*, which is used to communicate a result from a child activity to its parent, thus simulating the behaviour of rule (A-Result) in the concrete semantics; again, no serialization happens in the process, rather a pointer to the result is passed. Finally, rule *Sub* corresponds to an axiomatization of the subtyping relationships for the analysed program.

2.4.3 Formal Results

The soundness of the analysis is proved using *representation functions*, a standard approach in program analysis [NNH99]. The representation function β_{Cnf} maps an arbitrary configuration Ψ into a corresponding set of facts Δ, modelling an over-approximation of Ψ. Its definition is lengthy, but unsurprising, e.g., each element $\ell \mapsto b$ of the heap is converted into an abstract heap entry $\mathsf{H}(\lambda, \hat{b})$, where λ is the annotation on ℓ and $\hat{b}$ is an abstraction of b. After defining β_{Cnf}, we introduce a partial order $\sqsubseteq$ on analysis facts, with the intuitive understanding that $\mathsf{f} \sqsubseteq \mathsf{f}'$ whenever f is a more precise abstraction than f'. The partial order is then lifted to abstract programs by having $\Delta <: \Delta'$ if and only if $\forall \mathsf{f} \in \Delta : \exists \mathsf{f}' \in \Delta' : \mathsf{f} \sqsubseteq \mathsf{f}'$.

Our main theorem states that any reachable configuration in the concrete semantics is over-approximated by some set of facts which is provable using the abstract semantics of the program and an abstraction of the initial configuration. The proof is parametric with respect to the choice of the abstract domains/operations used for primitive values, provided they offer some minimal soundness guarantees. This allows for choosing different trade-offs between efficiency and precision of the analysis.

Theorem 1 (Preservation). *If $\Psi \Rightarrow^* \Psi'$ under a program P, there exists $\Delta :> \beta_{Cnf}(\Psi')$ such that:*

$$(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta.$$

By providing an over-approximation of any reachable configuration of the concrete semantics in terms of a corresponding set of facts, the theorem can be used to prove the absence of undesired information flows of sensitive data into local registers of selected sink methods. In particular, we leverage the theorem to develop a provably sound taint analysis, based on standard ideas. We refer to § 3.2 for full details.

2.5 Experiments

We developed HornDroid, a static analysis tool for Android applications based on our theory. HornDroid implements a sound, fully automatic taint analysis aimed at detecting malicious information flows in Android applications. The analysis is based on a publicly available database of sources and sinks specific to the Android platform [RAB14].

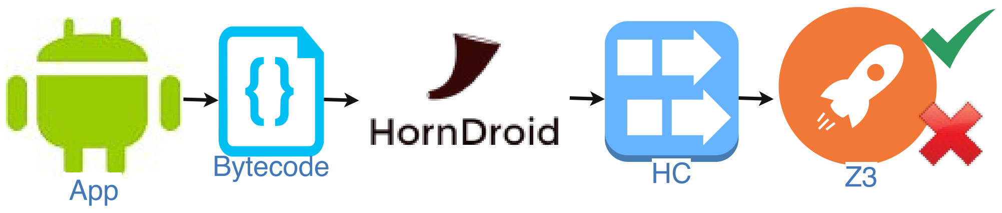


Figure 2.1: HornDroid Architecture

The architecture of HornDroid is shown in Figure 2.1. Given an Android application as an input, HornDroid generates Horn clauses defining an over-approximation of the application semantics, following the formal specification in Section 2.4; the choice of the underlying abstract domains and operations implements a simple taint propagation logic. The Horn clauses are encoded in the SMT-LIB format supported by many popular SMT solvers, including our choice *z3* [dMB08b]. HornDroid automatically generates analysis queries based on its database of sources and sinks[2] and the unsatisfiability of the queries is verified using the Property-Directed Reachability (PDR) engine implemented in *z3* [HB12]. If no query is satisfiable, no information leak from a source to a sink may occur in the analysed application.

2.5.1 Evaluation on DroidBench

DroidBench [ARF+14] is a set of small applications which has been proposed by the research community as a testing ground for static information flow analysis tools for Android. The current version of the benchmark (2.0) includes 120 test cases, featuring both leaky (positive) and benign (negative) examples. We tested IccTA, AmanDroid, DroidSafe and HornDroid on this benchmark, the results are summarized in the confusion matrix in Table 2.11, reporting the number of true positives (tp), true negatives (tn), false positives (fp) and false negatives (fn) produced by the tools.

	Output	
	leaky	*benign*
	IccTA/AD/DS/HD	*IccTA/AD/DS/HD*
leaky	tp : 64 / 70 / 89 / 96	fn : 36 / 30 / 11 / 4
benign	fp : 8 / 5 / 10 / 6	tn : 11 / 14 / 9 / 13

Table 2.11: Confusion Matrix on DroidBench

IccTA does not detect 36 out of 100 leaky applications, AmanDroid misses 30 and DroidSafe still misses 11. Most of the leaks missed by IccTA and AmanDroid are due to flow-sensitivity and some callbacks which are not correctly detected by the analysis; as to DroidSafe, we do not have definite answers on the unsound results, given the sheer size of the project and the lack of complete documentation. HornDroid performs much better than all its competitors on DroidBench, since it only misses 4 leaky applications: all these cases are related to *implicit flows*, which are not covered by standard taint analyses (and our formal proof).

But even better, despite the strong security guarantees it provides, the analysis performed by HornDroid is not overly conservative, since it detects as potentially leaky only 6 out of 19 benign applications. We notice that 3 of these false alarms are due to flow insensitivity of the heap abstraction, one to an over-approximation of exceptions, and 2 to an over-approximated treatment of inter-app communication. Only AmanDroid is more precise, since it produces one less false positive; on the other hand, it misses many more malicious information flows than HornDroid (30 vs 4). For the sake of completeness, we report in Table 2.13 a full breakdown of the experiments on DroidBench, omitting the cases where all the tools agree with the ground truth.

[2]We use the latest and largest database available in the literature at the time of writing, i.e. the one used in DroidSafe [GKP+15].

The experimental results on DroidBench are summarized by a few standard statistical measures in Table 2.12, which highlight that soundness in HornDroid does not come at the cost of precision.

	IccTA	*AD*	*DS*	*HD*
Sensitivity	0.64	0.70	0.89	0.96
Specificity	0.58	0.74	0.47	0.68
F-Measure	0.61	0.72	0.62	0.80

$\mathit{Sensitivity} = tp/(tp + fn) \sim$ Soundness
$\mathit{Specificity} = tn/(tn + fp) \sim$ Precision
$\mathit{F\text{-}Measure} = 2 * (sens * spec)/(sens + spec) \sim$ Aggregate

Table 2.12: Performance Measures on DroidBench

Besides the quality of the results, also performances are important. Table 2.15 reports the mean and the median of the analysis times for the applications in DroidBench. As it turns out, HornDroid is one order of magnitude faster than both IccTA and AmanDroid, which in turn are one order of magnitude faster than DroidSafe. The extremely good performances of HornDroid are due to both design choices, like flow insensitivity on the activity lifecycle, and excellent support by *z3* in Horn clauses resolution.

2.5.2 Evaluation on Real Applications

In order to evaluate the practicality of our analysis, we performed a test on the two largest applications available in the Google Play Top 30: the game Candy Crash Soda Saga (51.7 Mb) and the Facebook application (46.5 Mb). We ran the experiments on a server with 64 multi-thread cores and 758 Gb of memory, however the highest memory consumption by HornDroid was around 10 Gb, so it is possible to reproduce our results even on a modern commercial machine.

HornDroid found an information leak in Facebook, while Candy Crash Soda Saga appears to be secure. The analysis took around 30 minutes and 60 minutes, respectively. We tested all the existing competitors on both applications, to check whether they could confirm the analysis results. Unfortunately, AmanDroid crashed just after the beginning of the analysis of Facebook, while both DroidSafe and IccTA failed to terminate within the timeout we set (2 hours). We were able instead to analyse Candy Crash Soda Saga using AmanDroid in around 50 minutes, getting an information flow. After a manual inspection, we realized this is a false positive due to the incorrect inclusion of the `onHandleIntent` method of the class `IntentService` among the possible sources of sensitive information: this is not included in more recent proposals [GKP+15, LBB+15]. Both IccTA and DroidSafe were not able to analyse the application within 2 hours.

Category	*Case*	*Leaky?*	*IccTA*	*DS*	*AD*	*HD*
Aliasing	Merge1	*no*	yes	yes	*no*	yes
Android-Specific	Parcel1	*yes*	no	*yes*	*yes*	*yes*
	PublicAPIField1	*yes*	no	*yes*	no	*yes*
	PublicAPIField2	*yes*	no	*yes*	no	*yes*
Arrays and Lists	ArrayAccess1	*no*	yes	yes	yes	*no*
	ArrayAccess2	*no*	yes	yes	yes	*no*
	ArrayCopy1	*yes*	*yes*	*yes*	no	*yes*
	ArrayToString1	*yes*	no	*yes*	*yes*	*yes*
	HashMapAccess1	*no*	yes	yes	*no*	*no*
	ListAccess1	*no*	yes	yes	yes	*no*
	MultidimensionalArray1	*yes*	*yes*	no	*yes*	*yes*
Callbacks	MultiHandlers1	*yes*	no	no	no	*yes*
	Ordering1	*yes*	no	*yes*	*yes*	*yes*
	RegisterGlobal1	*yes*	*yes*	*yes*	no	*yes*
	RegisterGlobal2	*yes*	*yes*	*yes*	no	*yes*
	Unregister1	*no*	yes	yes	yes	yes
Emulator Detection	ContentProvider1	*yes*	*yes*	*yes*	no	*yes*
	IMEI1	*yes*	no	no	no	no
	PlayStore1	*yes*	*yes*	*yes*	no	*yes*
Fields and Object Sensitivity	FieldSensitivity4	*no*	*no*	yes	*no*	yes
	ObjectSensitivity2	*no*	*no*	yes	*no*	yes
General Java	Exceptions3	*no*	yes	yes	yes	yes
	Serialization1	*yes*	no	*yes*	no	*yes*
	StartProcessWithSecret1	*yes*	no	*yes*	no	*yes*
	StaticInitialization1	*yes*	no	*yes*	*yes*	*yes*
	StaticInitialization3	*yes*	no	*yes*	*yes*	*yes*
	StringFormatter1	*yes*	no	*yes*	no	*yes*
	StringPatternMatching1	*yes*	no	*yes*	*yes*	*yes*
	StringToCharArray1	*yes*	*yes*	*yes*	no	*yes*
	StringToOutputStream1	*yes*	no	*yes*	*yes*	*yes*
	VirtualDispatch3	*no*	yes	*no*	*no*	*no*
Implicit Flows	ImplicitFlow1	*yes*	no	*yes*	no	*yes*
	ImplicitFlow2	*yes*	no	no	no	no
	ImplicitFlow3	*yes*	no	no	no	no
	ImplicitFlow4	*yes*	no	no	no	no
Inter-App Communication	Echoer	*yes*	no	*yes*	no	*yes*
	SendSMS	*yes*	*yes*	*yes*	no	*yes*
Inter-Component Communication	ActivityCommunication1	*yes*	*yes*	*yes*	no	*yes*
	ActivityCommunication3	*yes*	no	*yes*	*yes*	*yes*
	ActivityCommunication6	*yes*	no	*yes*	*yes*	*yes*
	ComponentNotInManifest1	*no*	*no*	yes	*no*	yes
	IntentSink1	*yes*	no	*yes*	*yes*	*yes*
	IntentSink2	*yes*	no	*yes*	no	*yes*
	IntentSource1	*yes*	no	*yes*	*yes*	*yes*
	ServiceCommunication1	*yes*	no	*yes*	*yes*	*yes*
	Singletons1	*yes*	no	no	no	*yes*

Table 2.13: DroidBench Results (continues in Table 2.14)

Category	*Case*	*Leaky?*	*IccTA*	*DS*	*AD*	*HD*
Lifecycle	ActivityLifecycle1	*yes*	no	*yes*	*yes*	*yes*
	ActivitySavedState1	*yes*	no	*yes*	*yes*	*yes*
	ApplicationLifecycle1	*yes*	*yes*	*yes*	no	*yes*
	ApplicationLifecycle2	*yes*	*yes*	*yes*	no	*yes*
	ApplicationLifecycle3	*yes*	*yes*	*yes*	no	*yes*
	BroadcastReceiverLifecycle2	*yes*	no	*yes*	no	*yes*
	FragmentLifecycle1	*yes*	no	*yes*	*yes*	*yes*
	FragmentLifecycle2	*yes*	no	*yes*	no	*yes*
	SharedPreferenceChanged1	*yes*	*yes*	no	*yes*	*yes*
Reflection	Reflection1	*yes*	*yes*	no	*yes*	*yes*
	Reflection2	*yes*	no	no	no	*yes*
	Reflection3	*yes*	no	*yes*	*yes*	*yes*
	Reflection4	*yes*	no	no	*yes*	*yes*
Threading	Executor1	*yes*	*yes*	*yes*	no	*yes*
	JavaThread1	*yes*	*yes*	*yes*	no	*yes*
	JavaThread2	*yes*	no	*yes*	no	*yes*
	Looper1	*yes*	no	*yes*	no	*yes*

Table 2.14: DroidBench Results (continuation of Table 2.13)

	IccTA	*AD*	*DS*	*HD*
Average Analysis Time	19	11	176	1
Median Analysis Time	15	10	186	1

Table 2.15: Analysis Time for DroidBench (Seconds)

2.5.3 Features and Limitations

As anticipated, the formalisation in the previous sections only captures the *core* of the analysis implemented in HornDroid and establishes the soundness of its principles. The tool, however, supports more features which are needed to make the analysis scale to real applications. We detail here some important aspects of HornDroid which are not covered by our formal model and we comment on current limitations.

Android Components

Although the μ-Dalvik$_A$ model only represents activities and their lifecycle, HornDroid supports all the component types available on the Android platform, including services, broadcast receivers and content providers [The16a]. The implementation of the analysis for these components does not significantly differ from the one for activities we presented in the chapter, though it requires a correct modelling of their specific lifecycle.

Fragments

Fragments are used to separate the functionality of an activity among different independent sub-components [The16b]. In order to support a sound analysis of fragments, HornDroid over-approximates their lifecycle by executing all the fragments along with the containing activity in a flow-insensitive way. This might lead to precision problems on real applications, but this is the simplest of the sound options, which follows the philosophy we adopted for activity analysis.

Arrays

Though the static analysis we formalized is field-insensitive on arrays, HornDroid supports a more precise treatment of array indexes. Being value-sensitive, HornDroid statically approximates which indexes of an array may be accessed at runtime: if a secret value is stored in the first position of the array, but only the second element of the array is leaked, the tool does not raise the alarm, contrarily to all the other existing tools (cf. the breakdown on the experiments in Table 2.13).

Exceptions

HornDroid implements a conservative solution to handle exceptions, i.e., exceptions are always assumed to be thrown. A similar coarse over-approximation is implemented in FlowDroid [ARF+14].

Inter-app Communication

HornDroid has limited support for inter-application communication, i.e., it conservatively detects an information leak whenever an intent storing secret data is sent to another application. More precise results could be achieved by analysing all the communicating applications simultaneously, but the current implementation of HornDroid only supports the analysis of a single application.

Threading

HornDroid handles multithreading by assuming that threads are executed in a sequential, but arbitrary order, much in the same spirit of the callbacks defining the activity lifecycle. This is the same strategy used in FlowDroid. We conjecture that this strategy is sound in our case, since the analysis is flow insensitive on everything except for registers, which are not shared. For flow-sensitive analysis techniques (e.g., FlowDroid), instead, this strategy is in general unsound, since it may miss potential interleavings arising due to synchronization on shared memory (e.g., static heaps). The only aspect that should be added to our static analysis is a thread pool simulation. In Java, every time the method `execute` is called on a thread, this is placed in a pool and then executed by the system by calling the runnable method `run`. Our static analysis similarly binds each invocation of `execute` to a corresponding `run` method.

Reflection

Though supporting reflection soundly is an open research problem [SKB14], HornDroid still covers a significant fraction of common reflection cases by implementing a simple string analysis. The solution we propose is in the same spirit of DroidSafe, i.e., reflective calls which can be statically resolved are replaced by direct calls to the appropriate method. Pragmatically, however, we observed that we are able to achieve much better results than DroidSafe for the reflection cases in DroidBench.

Limitations

A comprehensive implementation of analysis stubs for method calls to the Android APIs is still lacking: we only implemented some selected stubs for our experiments, to show that our approach is feasible and practical. When a stub to an external library is missing, the tool tries to be conservative: the return value of the call is over-approximated to the top element of the corresponding abstract domain, and it is tainted whenever at least one of the arguments is tainted. Other important limitations of HornDroid are shared with existing solutions [ARF$^+$14, GKP$^+$15]. First, the analysis does not capture *implicit* information flows at present. Second, the analysis does not consider *native code*: this is a point we leave as a future work, observing that SMT solving has been successfully applied in the past to C code (see, e.g., the SLAM project [BLR11]). Third, the analysis is oblivious to the *semantics* of the information flows, i.e., it lacks any built-in declassification mechanism to qualify legitimate data flows. Since our analysis approximates data information rather than just tracking taints, however, it is in principle possible to encode expressive data-dependent declassification policies, e.g., one could define the result of an encryption as untainted only if the encryption is performed with the right key.

2.6 Additional Related Work

Several papers have proposed an operational semantics for Android applications by now. The first attempt is due to Chaudhuri [Cha09], who presented a core calculus to model Android applications. Later research proposed much more concrete models: Jeon *et al.* developed μ-Dalvik, a relatively simple formal language which thoroughly models a significant fraction of the Dalvik opcodes [JMF12]. Wognsen *et al.* presented an even richer language, which also formalizes exceptions and some common uses of reflection [WKOH14]. Recently, Payet and Spoto complemented existing research by defining the first operational semantics for Android activities [PS14]. The semantics takes into account the event-driven behaviour of the activity lifecycle and, to some extent, the inter-component communication mechanism. Unfortunately, though, it represents only a small subset of the opcodes available in Dalvik and just models the control flow of activities, rather than the data flows enabled by inter-component communication. Our proposal integrates [JMF12] and [PS14], while providing the first accurate description of how data flows between different components of an Android application.

Cassandra [LMS+14] is, to the best of our knowledge, the only tool implementing a provably sound information flow analysis for Android applications. The analysis is based on security types: well-typed programs ensure a termination-insensitive notion of non-interference, which proves the absence of both explicit and implicit information flows. By capturing implicit flows, Cassandra provides stronger security assurances than other static analysis tools, including ours. On the other hand, the analysis implemented in Cassandra is exclusively focused on the bytecode, and it does not track information leaks enabled by the application lifecycle. Moreover, the design of Cassandra is not very practical, since it requires application developers to write security certificates, giving a typing of all fields and methods in the application. Being type-based, Cassandra does not track any static approximation of runtime values, thus making it easy for malicious developers to force an overwhelming number of false alarms. We are not aware of any experimental evaluation of Cassandra so far.

Static analyses for improving the security of Android applications are not limited to information flow control: important applications include the detection of over-privileged apps [FCH+11] and of attack surfaces for privilege escalation [BCS13]. Finally, it is worth mentioning that also dynamic analysis of Android applications is a popular research line [EGH+14, JAF+13, TR14, HHJ+11]. Dynamic analysis is largely complementary to static analysis, since it is typically more precise, but it hardly provides full coverage of all the possible execution paths and thus is not suitable to be employed in the vetting phase of an application.

CHAPTER 3

Proofs of Chapter 2

Chapter Outline: In Section 3.1 we describe the instrumented semantics for the Dalvik bytecode; in Section 3.2 we present the soundness proofs.

3.1 Formal Semantics of Statements

We present an *instrumented* semantics, which is useful for our soundness proof. With respect to the informal presentation in Section 2.3, we need to extend the syntax of semantic domains as follows:

$$\begin{array}{lll} L & ::= & \langle pp \cdot v^* \cdot st^* \cdot R \rangle \\ \Sigma & ::= & \ell \cdot \alpha \cdot \pi \cdot H \cdot S \end{array}$$

In the instrumented semantics, local states L additionally contain a sequence of values v^*, representing the actual arguments provided upon method invocation when the local state was pushed on the call stack. Local configurations Σ, instead, are extended with a pointer ℓ to the activity modelled by the configuration.

Definition 6. Given a heap H, we let the partial function $type_H(v)$ be defined as follows:

$$type_H(v) = \begin{cases} c & \text{if } v = \ell \wedge H(\ell) = \{\!|c; (f \mapsto v)^*|\!\} \\ \texttt{array}[\tau] & \text{if } v = \ell \wedge H(\ell) = \tau[v^*] \\ \texttt{Intent} & \text{if } v = \ell \wedge H(\ell) = \{\!|@c; (k \mapsto v)^*|\!\} \\ \tau_{prim} & \text{if } v = prim \end{cases}$$

where τ_{prim} is the type of the primitive value *prim*.

Let now $super(c) = c'$ iff there exists cls_i s.t. $cls_i = \texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{fld^*; mtd^*\}$. Similarly, let $inter(c) = \{c^*\}$ iff there exists cls_i s.t. $cls_i = \texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{fld^*;$

$mtd^*\}$. Table 3.1 gives the subtyping rules for μ-Dalvik$_A$, which are used, e.g., when defining the outcome of a type cast statement. Notice that array subtyping is covariant, which is unsound in presence of side-effects: like Java and the original presentation of μ-Dalvik, we detect possible type errors at runtime.

$$\text{(Sub-Refl)}\quad \tau \leq \tau \qquad \text{(Sub-Trans)}\quad \frac{\tau \leq \tau' \qquad \tau' \leq \tau''}{\tau \leq \tau''} \qquad \text{(Sub-Ext)}\quad c \leq \mathit{super}(c) \qquad \text{(Sub-Impl)}\quad \frac{c' \in \mathit{inter}(c)}{c \leq c'}$$

$$\text{(Sub-Array)}\quad \frac{\tau \leq \tau'}{\mathtt{array}[\tau] \leq \mathtt{array}[\tau']}$$

Table 3.1: Subtyping ($\tau \leq \tau'$)

Let $a[i] = v_i$ whenever $a = \tau[v^*]$ and $o.f = v$ whenever $o = \{\!|c; (f_i \mapsto v_i)^*, f \mapsto v|\!\}$. Table 3.2 defines a convenience relation used to evaluate the right-hand side of a move instruction under a local configuration Σ. Notice that the evaluation of registers depends only on the top-most local state of the call stack of Σ.

$$\text{(Rhs-Register)}\quad \Sigma[\![r]\!] = R(r) \qquad \text{(Rhs-Array)}\quad \frac{\ell = \Sigma[\![r_a]\!] \quad a = H(\ell) \quad j = \Sigma[\![r_{idx}]\!]}{\Sigma[\![r_a[r_{idx}]]\!] = a[j]} \qquad \text{(Rhs-Object)}\quad \frac{\ell = \Sigma[\![r_o]\!] \quad o = H(\ell)}{\Sigma[\![r_o.f]\!] = o.f} \qquad \text{(Rhs-Static)}\quad \Sigma[\![c.f]\!] = S(c.f)$$

$$\text{(Rhs-Prim)}\quad \Sigma[\![\mathit{prim}]\!] = \mathit{prim}$$

Convention: in all the rules, let $\Sigma = \_ \cdot \alpha \cdot \pi \cdot H \cdot S$ with $\alpha = \langle pp \cdot \_ \cdot st^* \cdot R\rangle :: \alpha'$.

Table 3.2: Evaluation of Right-hand Sides ($\Sigma[\![rhs]\!] = v$)

It is also useful to define substitutions for different syntactic categories, e.g., we let $o[f \mapsto v] = \{\!|c; (f_i \mapsto v_i)^*[f \mapsto v]|\!\}$ when $o = \{\!|c; (f_i \mapsto v_i)^*|\!\}$, and $\Sigma[H \mapsto H'] = \ell \cdot \alpha \cdot \pi \cdot H' \cdot S$ when $\Sigma = \ell \cdot \alpha \cdot \pi \cdot H \cdot S$. We do not provide full formal definitions for these substitutions, since their meaning will be clear from the context: it is only worth noticing that substitutions operating on elements of a local state only affect the *top-most* local state of a local configuration Σ when applied to it. For instance, given $\Sigma = \ell \cdot \alpha \cdot \pi \cdot H \cdot S$ with $\alpha = \langle pp \cdot v^* \cdot st^* \cdot R\rangle :: \alpha'$, we let $\Sigma[R \mapsto R'] = \ell \cdot \alpha'' \cdot \pi \cdot H \cdot S$ where $\alpha'' = \langle pp \cdot v^* \cdot st^* \cdot R'\rangle :: \alpha'$, i.e., α' is unchanged.

We are finally ready to define the formal semantics of statements. Let $\Sigma = \ell \cdot \alpha \cdot \pi \cdot H \cdot S$, we let $\mathit{get\text{-}stm}(\Sigma) = st_{pc}$ when $\alpha = \langle c, m, pc \cdot \_ \cdot st^* \cdot R \rangle :: \alpha'$; we then let $\Sigma \rightsquigarrow \Sigma'$ if $\mathit{get\text{-}stm}(\Sigma) = st$ and $\Sigma, st \Downarrow \Sigma'$ can be proved using the rules in Table 3.3 and Table 3.4. There are only three perhaps surprising points: (1) when storing a value in an array cell, a dynamic check on the type of the value is performed, so as to ensure type soundness even in presence of the unsound subtyping rule for arrays; (2) when a new object is created, the pointer to it is annotated with the program point where creation takes place; and (3) upon method invocation, the value of the actual arguments is tracked in the syntax of the new local state. While (1) is an important aspect of the operational semantics, both (2) and (3) only serve static analysis purposes. Notice that we also use *lookup* to retrieve method bodies upon static calls: in this case, we assume $c' = c$.

3.2 Proofs

3.2.1 Representation Functions

We presuppose the existence of a representation function β_{Prim} which associates to each primitive value *prim* a corresponding abstract value $\{\widehat{prim}\}$. For a location $\ell = p_\lambda$, we let $\beta_{\mathit{Loc}}(\ell) = \{\lambda\}$. Based on this, we define $\beta_{\mathit{Val}}(v)$ as follows:

$$\beta_{\mathit{Val}}(v) = \begin{cases} \beta_{\mathit{Prim}}(v) & \text{if } v = \mathit{prim} \\ \beta_{\mathit{Loc}}(v) & \text{if } v = \ell \end{cases}$$

We typically omit brackets around singleton abstract values. We then define $\beta_{\mathit{Blk}}(b)$ as follows:

$$\beta_{\mathit{Blk}}(b) = \begin{cases} \{\!| c; (f \mapsto \hat{v})^* |\!\} & \text{if } b = \{\!| c; (f \mapsto v)^* |\!\} \text{ and } \forall i : \beta_{\mathit{Val}}(v_i) = \hat{v}_i \\ \{\!| @c; \hat{v} |\!\} & \text{if } b = \{\!| @c; (f \mapsto v)^* |\!\} \text{ and } \hat{v} = \sqcup_i \beta_{\mathit{Val}}(v_i) \\ \tau[\hat{v}] & \text{if } b = \tau[v^*] \text{ and } \hat{v} = \sqcup_i \beta_{\mathit{Val}}(v_i) \end{cases}$$

Using these definitions, we can define how configurations are translated into facts by a corresponding representation function. This requires one to define a number of clauses, summarized below:

$$\begin{array}{lcl}
\beta_{\mathit{Lst}}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle) & = & \{\mathsf{R}_{c,m,pc}(\hat{u}^* \,;\, \hat{v}^*) \mid \forall j : \hat{u}_j = \beta_{\mathit{Val}}(u_j) \land \forall k : \hat{v}_k = \beta_{\mathit{Val}}(R(r_k))\} \cup \bigcup_i (\!| st_i |\!)_{c,m,i} \\
\beta_{\mathit{Call}}(\alpha) & = & \bigcup_{i \in [1,n]} \beta_{\mathit{Lst}}(L_i) \text{ whenever } \alpha = L_1 :: \ldots :: L_n \\
\beta_{\mathit{Heap}}(H) & = & \{\mathsf{H}(\lambda, \hat{b}) \mid H = H', \ell \mapsto b \land \lambda = \beta_{\mathit{Loc}}(\ell) \land \hat{b} = \beta_{\mathit{Blk}}(b)\} \\
\beta_{\mathit{Stat}}(S) & = & \{\mathsf{S}(c, f, \hat{v}) \mid S = S', c.f \mapsto v \land \hat{v} = \beta_{\mathit{Val}}(v)\} \\
\beta^{\ell}_{\mathit{Pact}}(\pi) & = & \{\mathsf{I}(c, \hat{b}) \mid c = \beta_{\mathit{Loc}}(\ell) \land \pi = \pi_0 :: i :: \pi_1 \land \hat{b} = \beta_{\mathit{Blk}}(i)\} \\
\beta_{\mathit{Lcnf}}(\ell \cdot \alpha \cdot \pi \cdot H \cdot S) & = & \beta_{\mathit{Call}}(\alpha) \cup \beta^{\ell}_{\mathit{Pact}}(\pi) \cup \beta_{\mathit{Heap}}(H) \cup \beta_{\mathit{Stat}}(S) \\
\beta_{\mathit{Frm}}(\langle \ell, s, \pi, \alpha \rangle) & = & \beta_{\mathit{Frm}}(\underline{\langle \ell, s, \pi, \alpha \rangle}) = \beta^{\ell}_{\mathit{Pact}}(\pi) \cup \beta_{\mathit{Call}}(\alpha) \\
\beta_{\mathit{Stk}}(\Omega) & = & \bigcup_{i \in [1,n]} \beta_{\mathit{Frm}}(\varphi_i) \text{ whenever } \Omega = \varphi_1 :: \ldots :: \varphi_n \\
\beta_{\mathit{Cnf}}(\Omega \cdot H \cdot S) & = & \beta_{\mathit{Stk}}(\Omega) \cup \beta_{\mathit{Heap}}(H) \cup \beta_{\mathit{Stat}}(S)
\end{array}$$

$$\text{(R-Goto)}\quad \Sigma, \texttt{goto}\ pc' \Downarrow \Sigma[pc \mapsto pc']$$

$$\text{(R-True)}\quad \frac{\Sigma[\![r_1]\!] \olessthan \Sigma[\![r_2]\!]}{\Sigma, \texttt{if}_{\olessthan}\ r_1\ r_2\ \texttt{then}\ pc' \Downarrow \Sigma[pc \mapsto pc']}$$

$$\text{(R-False)}\quad \frac{\neg(\Sigma[\![r_1]\!] \olessthan \Sigma[\![r_2]\!])}{\Sigma, \texttt{if}_{\olessthan}\ r_1\ r_2\ \texttt{then}\ pc' \Downarrow \Sigma^+}$$

$$\text{(R-MoveReg)}\quad \frac{\begin{array}{c} v = \Sigma[\![rhs]\!] \\ R' = R[r \mapsto v] \end{array}}{\Sigma, \texttt{move}\ r\ rhs \Downarrow \Sigma^+[R \mapsto R']}$$

$$\text{(R-MoveFld)}\quad \frac{\begin{array}{cc} v = \Sigma[\![rhs]\!] & \ell = \Sigma[\![r_o]\!] \\ o = H(\ell) & H' = H[\ell \mapsto o[f \mapsto v]] \end{array}}{\Sigma, \texttt{move}\ r_o.f\ rhs \Downarrow \Sigma^+[H \mapsto H']}$$

$$\text{(R-MoveArr)}\quad \frac{\begin{array}{ccc} & v = \Sigma[\![rhs]\!] & \\ \ell = \Sigma[\![r_a]\!] & type_H(\ell) = \texttt{array}[\tau] & type_H(v) \leq \tau \\ a = H(\ell) & j = \Sigma[\![r_{idx}]\!] & H' = H[\ell \mapsto a[j \mapsto v]] \end{array}}{\Sigma, \texttt{move}\ r_a[r_{idx}]\ rhs \Downarrow \Sigma^+[H \mapsto H']}$$

$$\text{(R-MoveSFld)}\quad \frac{\begin{array}{c} v = \Sigma[\![rhs]\!] \\ S' = S[c'.f \mapsto v] \end{array}}{\Sigma, \texttt{move}\ c'.f\ rhs \Downarrow \Sigma^+[S \mapsto S']}$$

$$\text{(R-UnOp)}\quad \frac{\begin{array}{c} v = \odot\Sigma[\![r_s]\!] \\ R' = [r_d \mapsto v] \end{array}}{\Sigma, \texttt{unop}_{\odot}\ r_d\ r_s \Downarrow \Sigma^+[R \mapsto R']}$$

$$\text{(R-BinOp)}\quad \frac{\begin{array}{c} v = \Sigma[\![r_1]\!] \oplus \Sigma[\![r_2]\!] \\ R' = R[r_d \mapsto v] \end{array}}{\Sigma, \texttt{binop}_{\oplus}\ r_d\ r_1\ r_2 \Downarrow \Sigma^+[R \mapsto R']}$$

$$\text{(R-NewObj)}\quad \frac{\begin{array}{cc} \multicolumn{2}{c}{o = \{\!|c'; (f_\tau \mapsto \mathbf{0}_\tau)^*|\!\}} \\ \multicolumn{2}{c}{\ell = p_{c,m,pc} \notin dom(H)} \\ H' = H[\ell \mapsto o] & R' = R[r_d \mapsto \ell] \end{array}}{\Sigma, \texttt{new}\ r_d\ c' \Downarrow \Sigma^+[H \mapsto H', R \mapsto R']}$$

$$\text{(R-NewArr)}\quad \frac{\begin{array}{cc} \multicolumn{2}{c}{len = \Sigma[\![r_l]\!]} \\ a = \tau[(\mathbf{0}_\tau)^{j \leq len}] & \ell = p_{c,m,pc} \notin dom(H) \\ H' = H[\ell \mapsto a] & R' = R[r_d \mapsto \ell] \end{array}}{\Sigma, \texttt{newarray}\ r_d\ r_l\ \tau \Downarrow \Sigma^+[H \mapsto H', R \mapsto R']}$$

$$\text{(R-Cast)}\quad \frac{\begin{array}{c} \ell = \Sigma[\![r_s]\!] \\ type_H(\ell) \leq \tau \end{array}}{\Sigma, \texttt{checkcast}\ r_s\ \tau \Downarrow \Sigma^+}$$

$$\text{(R-InstOfTrue)}\quad \frac{\begin{array}{c} \ell = \Sigma[\![r_s]\!] \\ type_H(\ell) \leq \tau \\ R' = R[r_d \mapsto \texttt{true}] \end{array}}{\Sigma, \texttt{instof}\ r_d\ r_s\ \tau \Downarrow \Sigma^+[R \mapsto R']}$$

$$\text{(R-InstOfFalse)}\quad \frac{\begin{array}{c} \ell = \Sigma[\![r_s]\!] \\ type_H(\ell) \not\leq \tau \\ R' = R[r_d \mapsto \texttt{false}] \end{array}}{\Sigma, \texttt{instof}\ r_d\ r_s\ \tau \Downarrow \Sigma^+[R \mapsto R']}$$

Convention: in all the rules, let $\Sigma = \_ \cdot \alpha \cdot \pi \cdot H \cdot S$ with $\alpha = \langle c, m, pc \cdot \_ \cdot \_ \cdot R \rangle :: \alpha_0$. We let Σ^+ (resp. α^+) stand for Σ (resp. α) where pc is replaced by $pc + 1$.

Table 3.3: Concrete small step semantics of μ-Dalvik$_A$ $(\Sigma, st \Downarrow \Sigma')$ - Statements

(R-Return)

$$\frac{\begin{array}{c}\alpha = \langle c, m, pc \cdot \_ \cdot \_ \cdot R\rangle :: \langle pp' \cdot v^* \cdot st^* \cdot R'\rangle :: \alpha' \\ \alpha'' = \langle pp' \cdot v^* \cdot st^* \cdot R'[r_{ret} \mapsto \Sigma\llbracket r_{ret}\rrbracket]\rangle :: \alpha'\end{array}}{\Sigma, \texttt{return} \Downarrow \Sigma[\alpha \mapsto \alpha'']}$$

(R-SCall)

$$\frac{\begin{array}{c}lookup(c', m') = (c', st^*) \qquad sign(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau \\ R' = ((r_j \mapsto \mathbf{0})^{j \le loc}, (r_{loc+k} \mapsto \Sigma\llbracket r'_k\rrbracket)^{k \le n}) \\ \alpha'' = \langle c', m', 0 \cdot (\Sigma\llbracket r'_k\rrbracket)^{k\le n} \cdot st^* \cdot R'\rangle :: \alpha^+\end{array}}{\Sigma, \texttt{sinvoke}\ c'\ m'\ r'_1, \ldots, r'_n \Downarrow \Sigma[\alpha \mapsto \alpha'']}$$

(R-Call)

$$\frac{\begin{array}{c}\ell = \Sigma\llbracket r_o\rrbracket \\ lookup(type_H(\ell), m') = (c', st^*) \qquad sign(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau \\ R' = ((r_j \mapsto \mathbf{0})^{j \le loc}, r_{loc+1} \mapsto \ell, (r_{loc+1+k} \mapsto \Sigma\llbracket r'_k\rrbracket)^{k \le n}) \\ \alpha'' = \langle c', m', 0 \cdot (\Sigma\llbracket r'_k\rrbracket)^{k\le n} \cdot st^* \cdot R'\rangle :: \alpha^+\end{array}}{\Sigma, \texttt{invoke}\ r_o\ m'\ r'_1, \ldots, r'_n \Downarrow \Sigma[\alpha \mapsto \alpha'']}$$

(R-NewIntent)

$$\frac{\begin{array}{c}i = \{\!|@c'; \cdot|\!\} \qquad \ell = p_{c,m,pc} \notin dom(H) \\ H' = H[\ell \mapsto i] \qquad R' = R[r_d \mapsto \ell]\end{array}}{\Sigma, \texttt{newintent}\ r_d\ c' \Downarrow \Sigma^+[H \mapsto H', R \mapsto R']}$$

(R-PutExtra)

$$\frac{\begin{array}{c}\ell = \Sigma\llbracket r_i\rrbracket \qquad i = H(\ell) \qquad k = \Sigma\llbracket r_k\rrbracket \\ v = \Sigma\llbracket r_v\rrbracket \qquad H' = H[\ell \mapsto i[k \mapsto v]]\end{array}}{\Sigma, \texttt{put-extra}\ r_i\ r_k\ r_v \Downarrow \Sigma^+[H \mapsto H']}$$

(R-GetExtra)

$$\frac{\begin{array}{c}\ell = \Sigma\llbracket r_i\rrbracket \\ k = \Sigma\llbracket r_k\rrbracket \qquad H(\ell) = i \qquad type_H(i.k) \le \tau \\ v = i.k \qquad R' = R[r_{ret} \mapsto v]\end{array}}{\Sigma, \texttt{get-extra}\ r_i\ r_k\ \tau \Downarrow \Sigma^+[R \mapsto R']}$$

(R-StartAct)

$$\frac{\ell = \Sigma\llbracket r_i\rrbracket \qquad H(\ell) = i \qquad \pi' = i :: \pi}{\Sigma, \texttt{start-activity}\ r_i \Downarrow \Sigma^+[\pi \mapsto \pi']}$$

Convention: in all the rules, let $\Sigma = \_ \cdot \alpha \cdot \pi \cdot H \cdot S$ with $\alpha = \langle c, m, pc \cdot \_ \cdot \_ \cdot R\rangle :: \alpha_0$. We let Σ^+ (resp. α^+) stand for Σ (resp. α) where pc is replaced by $pc + 1$.

Table 3.4: Concrete small step semantics of μ-Dalvik$_A$ (Σ, $st \Downarrow \Sigma'$) - Statements (Table 3.3 cont.)

3.2.2 Ordering Abstract Values and Facts

We presuppose the existence of a pre-order $\sqsubseteq_{Prim}$ on primitive singleton abstract values. Based on this, we define a pre-order $\sqsubseteq_{Val}$ on abstract values by having $\hat{u} \sqsubseteq_{Val} \hat{v}$ iff:

- $\forall \widehat{prim} \in \hat{u} : \exists \widehat{prim}' \in \hat{v} : \widehat{prim} \sqsubseteq_{Prim} \widehat{prim}'$;
- $\forall \lambda \in \hat{u} : \lambda \in \hat{v}$.

We then build a pre-order $\sqsubseteq_{Seq}$ on sequences of abstract values by having $\hat{u}^* \sqsubseteq_{Seq} \hat{v}^*$ iff $\hat{u}^*$ and $\hat{v}^*$ have the same length and:

$$\forall i : \hat{u}_i \sqsubseteq_{Val} \hat{v}_i.$$

We can then define a pre-order $\sqsubseteq_{Blk}$ on abstract blocks as follows:

- if $\hat{b} = \{\!|c; (f \mapsto \hat{u})^*|\!\}$ and $\hat{b}' = \{\!|c; (f \mapsto \hat{v})^*|\!\}$ and $\hat{u}^* \sqsubseteq_{Seq} \hat{v}^*$, then $\hat{b} \sqsubseteq_{Blk} \hat{b}'$;
- if $\hat{b} = \{\!|@c; \hat{u}|\!\}$ and $\hat{b}' = \{\!|@c; \hat{v}|\!\}$ and $\hat{u} \sqsubseteq_{Val} \hat{v}$, then $\hat{b} \sqsubseteq_{Blk} \hat{b}'$;
- if $\hat{b} = \tau[\hat{u}]$ and $\hat{b}' = \tau[\hat{v}]$ and $\hat{u} \sqsubseteq_{Val} \hat{v}$, then $\hat{b} \sqsubseteq_{Blk} \hat{b}'$.

Finally, we let $\mathsf{f} \sqsubseteq \mathsf{f}'$ be the least pre-order on facts such that:

- $\mathsf{R}_{c,m,pc}(\hat{u}^*_{call}\,;\,\hat{u}^*) \sqsubseteq \mathsf{R}_{c,m,pc}(\hat{v}^*_{call}\,;\,\hat{v}^*)$ whenever $\hat{u}^*_{call} \sqsubseteq_{Seq} \hat{v}^*_{call}$ and $\hat{u}^* \sqsubseteq_{Seq} \hat{v}^*$;
- $\mathsf{H}(\lambda, \hat{b}) \sqsubseteq \mathsf{H}(\lambda, \hat{b}')$ whenever $\hat{b} \sqsubseteq_{Blk} \hat{b}'$;
- $\mathsf{S}(c, f, \hat{u}) \sqsubseteq \mathsf{S}(c, f, \hat{v})$ whenever $\hat{u} \sqsubseteq_{Val} \hat{v}$;
- $\mathsf{RHS}_{\mathsf{pp}}(\hat{u}) \sqsubseteq \mathsf{RHS}_{\mathsf{pp}}(\hat{v})$ whenever $\hat{u} \sqsubseteq_{Val} \hat{v}$;
- $\mathsf{Res}_{c,m}(\hat{u}^*_{call}\,;\,\hat{u}^*) \sqsubseteq \mathsf{Res}_{c,m}(\hat{v}^*_{call}\,;\,\hat{v}^*)$ whenever $\hat{u}^*_{call} \sqsubseteq_{Seq} \hat{v}^*_{call}$ and $\hat{u}^* \sqsubseteq_{Seq} \hat{v}^*$;
- $\mathsf{I}(c, \hat{b}) \sqsubseteq \mathsf{I}(c, \hat{b}')$ whenever $\hat{b} \sqsubseteq_{Blk} \hat{b}'$.

3.2.3 Formal Results

Preliminaries

Definition 7. A local configuration $\Sigma = \ell \cdot \alpha \cdot \pi \cdot H \cdot S$ is *well-formed* if and only if, whenever $\alpha = L_1 :: \ldots :: L_n$, we have:

- either $n \in \{0, 1\}$, i.e., α is either empty or it contains just a single local state;
- or $n \geq 2$ and for each $i \in [2, n]$, either of the following conditions hold true:

- $L_i = \langle c', m', pc' \cdot v^* \cdot st'^* \cdot R' \rangle$ and $L_{i-1} = \langle c, m, pc \cdot \_ \cdot st^* \cdot R \rangle$ with $st_{pc} =$ `invoke` $r_o\ m'\ r'_1, \dots, r'_n$,
  $lookup(type_H(\Sigma[\![r_o]\!]), m') = (c', st'^*)$, $sign(c', m') = \tau_1, \dots, \tau_n \xrightarrow{loc} \tau$ and $v^* = (\Sigma[\![r'_k]\!])^{k \leq n}$
- $L_i = \langle c', m', pc' \cdot v^* \cdot st'^* \cdot R' \rangle$ and $L_{i-1} = \langle c, m, pc \cdot \_ \cdot st^* \cdot R \rangle$ with $st_{pc} =$ `sinvoke` $c'\ m'\ r'_1, \dots, r'_n$,
  $lookup(c', m') = (c', st'^*)$, $sign(c', m') = \tau_1, \dots, \tau_n \xrightarrow{loc} \tau$ and $v^* = (\Sigma[\![r'_k]\!])^{k \leq n}$.

Lemma 1 (Preserving Local Well-formation)**.** *If Σ is well-formed and $\Sigma \rightsquigarrow^* \Sigma'$, then Σ' is well-formed.*

Proof. By induction on the length of the reduction sequence and a case analysis on the last rule applied. □

Definition 8. A heap H is *well-typed* if and only if, whenever $H(\ell) = \{\!|c; (f_i \mapsto v_i)^{i \leq n}|\!\}$, for all $i \in [1, n]$ we have $type_H(v_i) \leq \tau_i$, where τ_i is the declared type of field f_i for an object of type c according to the underlying program.

Assumption 1 (Java Type Soundness)**.** If $\ell \cdot \alpha \cdot \pi \cdot H \cdot S \rightsquigarrow \ell \cdot \alpha' \cdot \pi' \cdot H' \cdot S'$, then for any value v we have $type_{H'}(v) \leq type_H(v)$. Moreover, if H is well-typed, then also H' is well-typed.

Definition 9. A configuration $\Psi = \Omega \cdot H \cdot S$ is *well-formed* if and only if:

- whenever $\Omega = \Omega_0 :: \varphi :: \Omega_1$ with $\varphi \in \{\langle \ell, s, \pi, \alpha \rangle, \underline{\langle \ell, s, \pi, \alpha \rangle}\}$, we have $H(\ell) = \{\!|c; (f \mapsto v)^*|\!\}$ for some activity class c and $\ell = p_c$ for some pointer p;
- whenever $\Omega = \Omega_0 :: \varphi :: \Omega_1$ with $\varphi \in \{\langle \ell, s, \pi, \alpha \rangle, \underline{\langle \ell, s, \pi, \alpha \rangle}\}$, we have that $\Sigma = \ell \cdot \alpha \cdot \pi \cdot H \cdot S$ is a well-formed local configuration;
- H is a well-typed heap.

Lemma 2 (Preserving Well-formation)**.** *If Ψ is well-formed and $\Psi \Rightarrow^* \Psi'$, then Ψ' is well-formed.*

Proof. By induction on the length of the reduction sequence and a case analysis on the last rule applied, using Lemma 1 and Assumption 1 to deal with case (A-Active). □

From now on, we tacitly focus only on well-formed configurations. All the formal results only apply to them: notice that well-formed configurations always reduce to well-formed configurations by Lemma 2.

Main Results

Lemma 3. *If $\Delta \subseteq \Delta'$, then $\Delta <: \Delta'$.*

Lemma 4. *If $\Delta <: \Delta'$ and $\Delta' <: \Delta''$, then $\Delta <: \Delta''$.*

Lemma 5. *If $\Delta_1 <: \Delta_2$ and $\Delta_3 <: \Delta_4$, then $\Delta_1 \cup \Delta_3 <: \Delta_2 \cup \Delta_4$.*

Assumption 2 (Soundness of the Abstract Operations)**.** We assume all the following properties:

- if $u \otimes v$, then $\hat{u} \mathrel{\hat{\otimes}} \hat{v}$ for any $\hat{u}, \hat{v}$ such that $\hat{u} :> \beta_{Val}(u)$ and $\hat{v} :> \beta_{Val}(v)$
- for any $\hat{v} :> \beta_{Val}(v)$, we have $\hat{\odot}\hat{v} :> \beta_{Val}(\odot v)$
- for any $\hat{u}, \hat{v}$ such that $\hat{u} :> \beta_{Val}(u)$ and $\hat{v} :> \beta_{Val}(v)$, we have $\hat{u} \mathbin{\hat{\oplus}} \hat{v} :> \beta_{Val}(u \oplus v)$

Assumption 3 (Overriding)**.** If $lookup(c, m) = (c', st^*)$, then $c \leq c'$.

In the next results, let $\Delta \vdash \Delta'$ whenever $\Delta \vdash \mathsf{f}$ for each $\mathsf{f} \in \Delta'$.

Lemma 6 (Right-hand Sides)**.** *Let $\Sigma = \ell \cdot \alpha \cdot \pi \cdot H \cdot S$ with $\alpha = \langle pp \cdot u^* \cdot st^* \cdot R \rangle$ and let $\Sigma[\![rhs]\!] = v$, then for any $\Delta :> \beta_{Lcnf}(\Sigma)$ there exists $\hat{v}$ such that $\beta_{Val}(v) \sqsubseteq_{Val} \hat{v}$ and $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{\mathsf{pp}}(\hat{v})$.*

Proof. By a case analysis on the structure of *rhs*. □

Lemma 7 (Local Preservation)**.** *If $\Sigma \rightsquigarrow \Sigma'$ under a given program P, then for any $\Delta :> \beta_{Lcnf}(\Sigma)$ there exists $\Delta' :> \beta_{Lcnf}(\Sigma')$ such that $(\!|P|\!) \cup \Delta \vdash \Delta'$.*

Proof. (Sketch) By a case analysis on the rule applied in the reduction step. The cases for the `move` instruction use Lemma 6. The case for the `return` instruction exploits the (implicit) well-formation assumption of the local configuration Σ. The case for the `invoke` instruction uses Assumption 3. The cases for comparison operators and primitive operations exploit Assumption 2. □

Lemma 8 (Serialization)**.** *Both the following statements hold true:*

- *if $ser^H_{Val}(v) = (v', H')$, then $\beta_{Val}(v) = \beta_{Val}(v')$*
- *if $ser^H_{Blk}(b) = (b', H')$, then $\beta_{Blk}(b) = \beta_{Blk}(b')$*

Proof. If $v = prim$, then $v' = prim$ and $\beta_{Val}(v) = \beta_{Val}(v') = \beta_{Prim}(prim)$. If $v = p_\lambda$, then $v' = p'_\lambda$ for some pointer p' and $\beta_{Val}(v) = \beta_{Loc}(p_\lambda) = \lambda = \beta_{Loc}(p'_\lambda) = \beta_{Val}(v')$. The second point is a direct consequence of the first one. □

Definition 10. We define a function $size^H$ which assigns to values and blocks a natural number as follows:

- $\Gamma \vdash size^H(prim) = 1$
- $\ell \notin \Gamma; \Gamma, \ell \vdash size^H(\ell) = 1 + size^H(H(\ell))$
- $\ell \in \Gamma; \Gamma, \ell \vdash size^H(\ell) = 0$
- $\Gamma \vdash size^H(\{\!|c; (f_i \mapsto v_i)^*|\!\}) = 1 + \sum_i size^H(v_i)$
- $\Gamma \vdash size^H(\{\!|@c; (k_i \mapsto v_i)^*|\!\}) = 1 + \sum_i size^H(v_i)$
- $\Gamma \vdash size^H(\tau[v^*]) = 1 + \sum_i size^H(v_i)$

Lemma 9 (Heap Serialization)**.** *If $\Delta :> \beta_{Heap}(H)$, then:*

- $ser^H_{Val}(v) = (v', H')$ *implies* $\Delta :> \beta_{Heap}(H')$
- $ser^H_{Blk}(b) = (b', H')$ *implies* $\Delta :> \beta_{Heap}(H')$

Proof. By simultaneous induction on the size of the syntactic element in the antecedent. If $v = prim$, then H' is empty, hence $\beta_{Heap}(H') = \emptyset$ and we are done. If $v = p_\lambda$, then $H' = H'', p'_\lambda \mapsto b$ with $ser^H_{Blk}(H(p_\lambda)) = (b, H'')$ and $v' = p'_\lambda$. By induction hypothesis $\Delta :> \beta_{Heap}(H'')$, so to conclude we just need to show that:

$$\begin{array}{rcll} \Delta & :> & \beta_{Heap}(p'_\lambda \mapsto b) & \\ & = & \{\mathsf{H}(\lambda, \beta_{Blk}(b))\} & \text{by definition} \\ & = & \{\mathsf{H}(\lambda, \beta_{Blk}(H(p_\lambda)))\} & \text{by Lemma 8} \\ & = & \beta_{Heap}(p_\lambda \mapsto H(p_\lambda)) & \text{by definition} \end{array}$$

but this follows from the hypothesis $\Delta :> \beta_{Heap}(H)$. The remaining cases for blocks follow by inductive hypothesis. □

Theorem 2 (Preservation)**.** *If $\Psi \Rightarrow^* \Psi'$ under a given program P, then there exists $\Delta :> \beta_{Cnf}(\Psi')$ such that $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta$.*

Proof. By induction on the length of the reduction sequence. If the reduction sequence is empty, we have $\Psi' = \Psi$ and the result follows by picking $\Delta = \beta_{Cnf}(\Psi)$. Otherwise, assume that $\Psi \Rightarrow^* \Omega \cdot H \cdot S$ in $n \geq 0$ reduction steps and let $\Omega \cdot H \cdot S \Rightarrow \Omega' \cdot H' \cdot S'$. By induction hypothesis there exists $\Delta' :> \beta_{Cnf}(\Omega \cdot H \cdot S)$ such that $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta'$, we show that there exists Δ such that $\Delta :> \beta_{Cnf}(\Omega' \cdot H' \cdot S')$ and $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta$. The proof is by a case analysis on the rule applied in the last reduction step:

(A-Active) : let $\Omega = \Omega_0 :: \underline{\langle \ell, s, \pi, \alpha \rangle} :: \Omega_1$ and $\Omega' = \Omega_0 :: \underline{\langle \ell, s, \pi', \alpha' \rangle} :: \Omega_1$ with $\ell \cdot \alpha \cdot \pi \cdot H \cdot S \rightsquigarrow \ell \cdot \alpha' \cdot \pi' \cdot H' \cdot S'$. Since $\beta_{Lcnf}(\ell \cdot \alpha \cdot \pi \cdot H \cdot S) \subseteq \beta_{Cnf}(\Omega \cdot H \cdot S)$, we have $\beta_{Lcnf}(\ell \cdot \alpha \cdot \pi \cdot H \cdot S) <: \beta_{Cnf}(\Omega \cdot H \cdot S)$ by Lemma 3. Since $\beta_{Lcnf}(\ell \cdot \alpha \cdot \pi \cdot H \cdot S) <: \beta_{Cnf}(\Omega \cdot H \cdot S)$ and $\beta_{Cnf}(\Omega \cdot H \cdot S) <: \Delta'$, we get $\beta_{Lcnf}(\ell \cdot \alpha \cdot \pi \cdot H \cdot S) <: \Delta'$ by Lemma 4. Hence, by Lemma 7 there exists $\Delta'' :> \beta_{Lcnf}(\ell \cdot \alpha' \cdot \pi' \cdot H' \cdot S')$ such that $(\!|P|\!) \cup \Delta' \vdash \Delta''$. By the weakening property of the logic, the latter implies $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \cup \Delta' \vdash \Delta''$. Since we have $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta'$ and $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \cup \Delta' \vdash \Delta''$, we get $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta''$ by the admissibility of the cut rule. Recall now that $\Delta'' :> \beta_{Lcnf}(\ell \cdot \alpha' \cdot \pi' \cdot H' \cdot S') = \beta_{Call}(\alpha') \cup \beta^{\ell}_{Pact}(\pi') \cup \beta_{Heap}(H') \cup \beta_{Stat}(S')$, so we have:

(1) $\Delta'' :> \beta_{Call}(\alpha')$

(2) $\Delta'' :> \beta^{\ell}_{Pact}(\pi')$

(3) $\Delta'' :> \beta_{Heap}(H')$

(4) $\Delta'' :> \beta_{Stat}(S')$

We then observe that $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta' :> \beta_{Cnf}(\Omega \cdot H \cdot S)$, which similarly implies:

(5) $\Delta' :> \beta_{Stk}(\Omega_0)$

(6) $\Delta' :> \beta_{Stk}(\Omega_1)$

Combining all these facts, we get $\Delta' \cup \Delta'' :> \beta_{Cnf}(\Omega' \cdot H' \cdot S')$ by Lemma 5. Given that $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta' \cup \Delta''$, we conclude the case;

(A-Deactivate) : in this case $\beta_{Cnf}(\Omega \cdot H \cdot S) = \beta_{Cnf}(\Omega' \cdot H' \cdot S')$, hence the conclusion immediately follows by the induction hypothesis;

(A-Step) : let $\Omega = \langle \ell, s, \pi, \overline{\alpha} \rangle :: \Omega_0$ and $\Omega' = \underline{\langle \ell, s', \pi, \alpha_{\ell.s'} \rangle} :: \Omega_0$ for some $(s, s') \in \mathit{Lifecycle}$, $H' = H$ and $S' = S$. Since $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta' :> \beta_{Cnf}(\Omega \cdot H \cdot S)$, we have:

(1) $\Delta' :> \beta_{Stk}(\Omega_0)$

(2) $\Delta' :> \beta^{\ell}_{Pact}(\pi)$

Since we only focus on well-formed configurations, we have $H(\ell) = \{\!|c; (f \mapsto u)^*|\!\}$ for some activity class c and $\ell = p_c$ for some pointer p. We then observe that $\alpha_{\ell.s'} = \langle c', m, 0 \cdot v^* \cdot st^* \cdot R \rangle :: \varepsilon$, where $(c', st^*) = \mathit{lookup}(c, m)$ for some $m \in cb(c, s)$, $\mathit{sign}(c', m) = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ and:

$$R = ((r_i \mapsto \mathbf{0})^{i \leq loc}, r_{loc+1} \mapsto \ell, (r_{loc+1+j} \mapsto v_j)^{j \leq n}),$$

for some values $v_1, \ldots, v_n$ of the correct type $\tau_1, \ldots, \tau_n$. By Assumption 3, we also have $c \leq c'$.

Given that $\Delta' :> \beta_{Cnf}(\Omega \cdot H \cdot S)$, we have $\Delta' :> \beta_{Heap}(H)$, which implies that there exists $\mathsf{H}(\lambda, \hat{b}) \in \Delta'$ such that $\lambda = \beta_{Loc}(\ell) = c$ and $\hat{b} \sqsupseteq \beta_{Blk}(\{\!|c; (f \mapsto u)^*|\!\})$. This implies that $\hat{b} = \{\!|c; (f \mapsto \hat{v})^*|\!\}$ for some v^* such that $\forall i : \hat{v}_i \sqsupseteq \beta_{Val}(u_i)$. Since

$(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta'$ and $\mathsf{H}(\lambda, \hat{b}) = \mathsf{H}(c, \{\!|c; (f \mapsto \hat{v})^*|\!\}) \in \Delta'$, we have in particular $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \mathsf{H}(c, \{\!|c; (f \mapsto \hat{v})^*|\!\})$, hence:

$$(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \mathsf{R}_{c',m,0}((\top_{\tau_j})^{j\leq n}\,;\,(\hat{\mathbf{0}})^{k\leq loc}, c, (\top_{\tau_j})^{j\leq n}),$$

by using the implications *Cbk* included in $(\!|P|\!)$. We then observe that:

$$\{\mathsf{R}_{c',m,0}((\top_{\tau_j})^{j\leq n}\,;\,(\hat{\mathbf{0}})^{k\leq loc}, c, (\top_{\tau_j})^{j\leq n})\} :> \beta_{Call}(\alpha_{\ell.s'})$$

By combining (1), (2) and the last observation through Lemma 5 we then get:

$$\{\mathsf{R}_{c',m,0}((\top_{\tau_j})^{j\leq n}\,;\,(\hat{\mathbf{0}})^{k\leq loc}, c, (\top_{\tau_j})^{j\leq n})\} \cup \Delta' :> \\ \beta_{Call}(\alpha_{\ell.s'}) \cup \beta_{Stk}(\Omega_0) \cup \beta^{\ell}_{Pact}(\pi) = \beta_{Stk}(\Omega')$$

Since $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \{\mathsf{R}_{c',m,0}((\top_{\tau_j})^{j\leq n}\,;\,(\hat{\mathbf{0}})^{k\leq loc}, c, (\top_{\tau_j})^{j\leq n})\} \cup \Delta'$, we conclude the case;

(A-Destroy) : in this case $\beta_{Cnf}(\Omega' \cdot H' \cdot S') \subseteq \beta_{Cnf}(\Omega \cdot H \cdot S)$, hence $\beta_{Cnf}(\Omega' \cdot H' \cdot S') <: \beta_{Cnf}(\Omega \cdot H \cdot S)$ by Lemma 3. Since $\beta_{Cnf}(\Omega' \cdot H' \cdot S') <: \beta_{Cnf}(\Omega \cdot H \cdot S)$ and $\beta_{Cnf}(\Omega \cdot H \cdot S) <: \Delta'$, we have $\beta_{Cnf}(\Omega' \cdot H' \cdot S') <: \Delta'$ by Lemma 4. Given that $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta'$, we conclude the case;

(A-Back) : let $\Omega' = \Omega = \langle \ell, running, \varepsilon, \overline{\alpha} \rangle :: \Omega_0$, $H' = H[\ell \mapsto H(\ell)[finished \mapsto \texttt{true}]]$ and $S' = S$. Let $b = H(\ell)$. Since we only focus on well-formed configurations, we have $b = \{\!|c; (f \mapsto u)^*, finished \mapsto v|\!\}$ for some activity class c and some boolean value v. Let then $b' = H'(\ell) = \{\!|c; (f \mapsto u)^*, finished \mapsto \texttt{true}|\!\}$ according to the reduction rule.

Given that $\Delta' :> \beta_{Cnf}(\Omega \cdot H \cdot S)$, we have $\Delta' :> \beta_{Heap}(H)$, which implies that there exists $\mathsf{H}(\lambda, \hat{b}) \in \Delta'$ such that $\lambda = \beta_{Loc}(\ell)$ and $\hat{b} \sqsupseteq \beta_{Blk}(b)$. This means that $\hat{b} = \{\!|c; (f \mapsto \hat{u})^*, finished \mapsto \hat{v}|\!\}$ for some u^*, v such that $\forall i : \hat{u}_i \sqsupseteq \beta_{Val}(u)$ and $\hat{v} \sqsupseteq \beta_{Val}(v)$. We then observe that:

$$\beta_{Blk}(b') = \{\!|c; (f \mapsto \beta_{Val}(u))^*, finished \mapsto \widehat{\texttt{true}}|\!\}$$

Since $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta'$ and $\mathsf{H}(\lambda, \hat{b}) \in \Delta'$, we have in particular $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \mathsf{H}(\lambda, \hat{b})$, hence:

$$(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \mathsf{H}(\lambda, \{\!|c; (f \mapsto \hat{u})^*, finished \mapsto \top_{\texttt{bool}}|\!\}),$$

by using the implication *Fin* included in $(\!|P|\!)$. We then observe that:

$$\begin{aligned}\mathsf{H}(\lambda, \{\!|c; (f \mapsto \hat{u})^*, finished \mapsto \top_{\texttt{bool}}|\!\}) &\sqsupseteq \mathsf{H}(\lambda, \{\!|c; (f \mapsto \hat{u})^*, finished \mapsto \widehat{\texttt{true}}|\!\}) \\ &= \mathsf{H}(\beta_{Loc}(\ell), \{\!|c; (f \mapsto \hat{u})^*, finished \mapsto \widehat{\texttt{true}}|\!\}) \\ &\sqsupseteq \mathsf{H}(\beta_{Loc}(\ell), \beta_{Blk}(b'))\end{aligned}$$

Hence, $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta' \cup \{\mathsf{H}(\lambda, \{\!|c; (f \mapsto \hat{u})^*, finished \mapsto \top_{\texttt{bool}}|\!\})\} :> \beta_{Heap}(H')$, which is enough to conclude the case;

(A-Replace) : let $\Omega = \langle \ell, \mathit{onDestroy}, \pi, \overline{\alpha} \rangle :: \Omega_0$ and $\Omega' = \underline{\langle p_c, \mathit{constructor}, \pi, \alpha_{p_c.\mathit{constructor}} \rangle} :: \Omega_0$ with $H(\ell) = \{\!|c; (f \mapsto v)^*, \mathit{finished} \mapsto u|\!\}$, $H' = H, p_c \mapsto o$ with $o = \{\!|c; (f \mapsto \mathbf{0}_\tau)^*, \mathit{finished} \mapsto \mathtt{false}|\!\}$, and $S' = S$. Since we only focus on well-formed configurations, we know that c is an activity class and $\ell = p'_c$ for some pointer p'.

Given that $(\!|P|\!) \cup \beta_{\mathit{Cnf}}(\Psi) \vdash \Delta' :> \beta_{\mathit{Cnf}}(\Omega \cdot H \cdot S)$, we have:

(1) $\Delta' :> \beta^{\ell}_{\mathit{Pact}}(\pi)$

(2) $\Delta' :> \beta_{\mathit{Stk}}(\Omega_0)$

Since $\beta_{\mathit{Loc}}(\ell) = \beta_{\mathit{Loc}}(p'_c) = \beta_{\mathit{Loc}}(p_c)$, from (1) we get:

(3) $\Delta' :> \beta^{p_c}_{\mathit{Pact}}(\pi)$

We then observe that $\alpha_{p_c.\mathit{constructor}} = \langle c', m, 0 \cdot v^* \cdot st^* \cdot R \rangle :: \varepsilon$, where $(c', st^*) = \mathit{lookup}(c, \mathit{constructor})$, $\mathit{sign}(c', \mathit{constructor}) = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ and:

$$R = ((r_i \mapsto \mathbf{0})^{i \le loc}, r_{loc+1} \mapsto p_c, (r_{loc+1+j} \mapsto v'_j)^{j \le n}),$$

for some values $v'_1, \ldots, v'_n$ of the correct type $\tau_1, \ldots, \tau_n$. By Assumption 3, we also have $c \le c'$.

Given that $\Delta' :> \beta_{\mathit{Cnf}}(\Omega \cdot H \cdot S)$, we have $\Delta' :> \beta_{\mathit{Heap}}(H)$, which implies that there exists $\mathsf{H}(\lambda, \hat{b}) \in \Delta'$ such that $\lambda = \beta_{\mathit{Loc}}(\ell) = c$ and $\hat{b} \sqsupseteq \beta_{\mathit{Blk}}(H(\ell))$. This implies that $\hat{b} = \{\!|c; (f \mapsto \hat{v})^*, \mathit{finished} \mapsto \hat{u}|\!\}$ for some $\hat{v}^*, \hat{u}$ such that $\forall i : \hat{v}_i \sqsupseteq \beta_{\mathit{Val}}(v_i)$ and $\hat{u} \sqsupseteq \beta_{\mathit{Val}}(u)$. Since $(\!|P|\!) \cup \beta_{\mathit{Cnf}}(\Psi) \vdash \Delta'$ and $\mathsf{H}(\lambda, \hat{b}) \in \Delta'$, we have in particular $(\!|P|\!) \cup \beta_{\mathit{Cnf}}(\Psi) \vdash \mathsf{H}(\lambda, \hat{b}) = \mathsf{H}(c, \{\!|c; (f \mapsto \hat{v})^*, \mathit{finished} \mapsto \hat{u}|\!\})$, hence:

$$(\!|P|\!) \cup \beta_{\mathit{Cnf}}(\Psi) \vdash \mathsf{R}_{c',m,0}((\top_{\tau_j})^{j \le n} ; (\hat{\mathbf{0}})^{k \le loc}, c, (\top_{\tau_j})^{j \le n}),$$

by using the implications Cbk included in $(\!|P|\!)$. We then observe that:

$$\{\mathsf{R}_{c',m,0}((\top_{\tau_j})^{j \le n} ; (\hat{\mathbf{0}})^{k \le loc}, c, (\top_{\tau_j})^{j \le n})\} :> \beta_{\mathit{Call}}(\alpha_{p_c.\mathit{constructor}})$$

By combining (2), (3) and the last observation through Lemma 5 we then get:

$$\begin{aligned}\{\mathsf{R}_{c',m,0}((\top_{\tau_j})^{j \le n} ; (\hat{\mathbf{0}})^{k \le loc}, c, (\top_{\tau_j})^{j \le n})\} \cup \Delta' :> \beta_{\mathit{Call}}(\alpha_{p_c.\mathit{constructor}}) \cup \beta_{\mathit{Stk}}(\Omega_0)\\ \cup\, \beta^{p_c}_{\mathit{Pact}}(\pi) = \beta_{\mathit{Stk}}(\Omega')\end{aligned}$$

Since $(\!|P|\!) \cup \beta_{\mathit{Cnf}}(\Psi) \vdash \{\mathsf{R}_{c',m,0}((\top_{\tau_j})^{j \le n} ; (\hat{\mathbf{0}})^{k \le loc}, c, (\top_{\tau_j})^{j \le n})\} \cup \Delta'$, we proved that the change to the activity stack is correctly over-approximated.

To conclude, we need to deal with the change to the heap. We first observe that $(\!|P|\!) \cup \beta_{\mathit{Cnf}}(\Psi) \vdash \Delta' :> \beta_{\mathit{Cnf}}(\Omega \cdot H \cdot S)$ and $\beta_{\mathit{Cnf}}(\Omega \cdot H \cdot S) :> \beta_{\mathit{Heap}}(H)$, hence:

(4) $\Delta' :> \beta_{\mathit{Heap}}(H)$

Since $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \mathsf{H}(\lambda, \hat{b}) = \mathsf{H}(c, \{\!|c; (f \mapsto \hat{v})^*, \mathit{finished} \mapsto \hat{u}|\!\})$, we have[1]:

$$(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathit{finished} \mapsto \widehat{\mathtt{false}}|\!\})),$$

by using the implication *Rep*. We then observe that:

$$\{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathit{finished} \mapsto \widehat{\mathtt{false}}|\!\}))\} :> \beta_{Heap}(p_c \mapsto o).$$

By combining (4) with the latter observation by Lemma 5, we get:

$$\Delta' \cup \{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathit{finished} \mapsto \widehat{\mathtt{false}}|\!\}))\} :> \beta_{Heap}(H')$$

Since $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta' \cup \{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathit{finished} \mapsto \widehat{\mathtt{false}}|\!\}))\}$, we proved that also the change to the heap is over-approximated correctly;

(A-Hidden) : analogous to case (A-Step);

(A-Start) : let $\Omega = \langle \ell, s, i :: \pi, \overline{\alpha} \rangle :: \Omega_0$ and $\Omega' = \underline{\langle p_c, \mathit{constructor}, \varepsilon, \alpha_{p_c.\mathit{constructor}} \rangle} :: \langle \ell, s, \pi, \overline{\alpha} \rangle :: \Omega_0$ with $i = \{\!|@c; (k \mapsto v)^*|\!\}$. Also, let $S' = S$ and $H' = H, H'', p_c \mapsto o, p'_{in(c)} \mapsto i'$ with $\mathit{ser}^H_{Blk}(i) = (i', H'')$ and $o = \{\!|c; (f \mapsto \mathbf{0}_\tau)^*, \mathit{finished} \mapsto \mathtt{false}, \mathit{intent} \mapsto p'_{in(c)}, \mathit{parent} \mapsto \ell|\!\}$. Since we only focus on well-formed configurations, we know that $\ell = p'_{c'}$ for some pointer p' and some activity class c'.

Given that $(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \Delta' :> \beta_{Cnf}(\Omega \cdot H \cdot S)$, we have $\Delta' :> \beta^{\ell}_{Pact}(i :: \pi)$, which implies that there exists $\mathsf{I}(\lambda, \hat{b}) \in \Delta'$ such that $\lambda = \beta_{Loc}(\ell) = c'$ and $\hat{b} \sqsupseteq \beta_{Blk}(i)$. This implies that $\hat{b} = \{\!|@c; \hat{v}|\!\}$ for some $\hat{v}$ such that $\hat{v} \sqsupseteq \sqcup_i \beta_{Val}(v_i)$. We then have:

$$(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \mathsf{H}(in(c), \{\!|@c; \hat{v}|\!\}),$$

and:

$$(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathit{finished} \mapsto \widehat{\mathtt{false}}, \mathit{parent} \mapsto c', \mathit{intent} \mapsto in(c)|\!\}),$$

by using the implications *Act* included in $(\!|P|\!)$. Using the latter fact and the implications *Cbk*, we can prove that the change to the activity stack is over-approximated correctly, similarly to what we did in case (A-Replace): we omit details.

We focus instead on the changes to the heap. Since $\Delta' :> \beta_{Heap}(H)$ and $\mathit{ser}^H_{Blk}(i) = (i', H'')$, we know that $\Delta' :> \beta_{Heap}(H'')$ by Lemma 9. We then observe that:

$$\{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathit{finished} \mapsto \widehat{\mathtt{false}}, \mathit{parent} \mapsto c', \mathit{intent} \mapsto in(c)|\!\})\} = \beta_{Heap}(p_c \mapsto o)$$

[1] We assume here that boolean fields are initialized to `false`. The proof can be adapted to the case where they are initialized to `true` by using the implication in rule *Fin*.

Finally, we notice that:

$$\begin{array}{lcll} \{\mathsf{H}(in(c), \{\!|@c; \hat{v}|\!\})\} & :> & \{\mathsf{H}(in(c), \beta_{Blk}(i)\} & \text{since } \hat{b} = \{\!|@c; \hat{v}|\!\}) \sqsupseteq \beta_{Blk}(i) \\ & = & \beta_{Heap}(p'_{in(c)} \mapsto i) & \text{by definition} \\ & = & \beta_{Heap}(p'_{in(c)} \mapsto i') & \text{by Lemma 8} \end{array}$$

By combining all these observations, we prove that the new heap is over-approximated correctly;

(A-Swap) : in this case $\beta_{Cnf}(\Omega \cdot H \cdot S) = \beta_{Cnf}(\Omega' \cdot H' \cdot S')$, hence the conclusion immediately follows by the induction hypothesis;

(A-Result) : let:

$$\Omega = \langle \ell', onPause, \varepsilon, \overline{\alpha}' \rangle :: \langle \ell, s, \varepsilon, \overline{\alpha} \rangle :: \Omega_0,$$

and:

$$\Omega' = \underline{\langle \ell, s, \varepsilon, \alpha_{\ell.onActivityResult} \rangle} :: \langle \ell', onPause, \varepsilon, \overline{\alpha}' \rangle :: \Omega_0,$$

with $H(\ell').parent = \ell$. Also, let $S' = S$ and $H' = (H, H'')[\ell \mapsto H(\ell)[result \mapsto \ell'']]$ with:

$$ser^{H}_{Val}(H(\ell').result) = (\ell'', H'').$$

Since we focus only on well-formed configurations, we have $\ell = p_c$ and $\ell' = p'_{c'}$ for some pointers p, p' and some activity classes c, c'. Also, let $H(\ell) = \{\!|c; (f \mapsto \hat{v})^*|\!\}$ and $H(\ell') = \{\!|c'; (f' \mapsto \hat{v}')^*, parent \mapsto \ell|\!\}$. Since $H(\ell) = \{\!|c; (f \mapsto \hat{v})^*|\!\}$, to prove that the changes to the activity stack are correctly over-approximated we can proceed like in case (A-Step), using the implications in *Cbk*: we omit details.

We focus instead on the changes to the heap. Since $\Delta' :> \beta_{Cnf}(\Omega \cdot H \cdot S)$, we have in particular:

(1) $\Delta' :> \beta_{Heap}(H)$

By (1) and $ser^{H}_{Val}(H(\ell').result) = (\ell'', H'')$, using Lemma 9, we prove:

(2) $\Delta' :> \beta_{Heap}(H'')$

Again by (1), there exists $\mathsf{H}(\lambda, \hat{b}) \in \Delta'$ such that $\lambda = \beta_{Loc}(\ell) = c$ and $\hat{b} \sqsupseteq \beta_{Blk}(H(\ell))$. This implies that $\hat{b} = \{\!|c; (f \mapsto \hat{v})^*|\!\}$ for some $\hat{v}^*$ s.t. $\forall i : \hat{v}_i \sqsupseteq \beta_{Val}(v_i)$. Similarly, we show that there exists $\mathsf{H}(\lambda', \hat{b}') \in \Delta'$ s.t. $\lambda' = \beta_{Loc}(\ell') = c'$ and $\hat{b}' \sqsupseteq \beta_{Blk}(H(\ell'))$, and $\hat{b}' = \{\!|c'; (f' \mapsto \hat{v}')^*, parent \mapsto c, result \mapsto \lambda''|\!\}$ for some $\hat{v}'^*, \lambda''$ such that $\forall i : \hat{v}'_i \sqsupseteq \beta_{Val}(v'_i)$ and $\lambda'' = \beta_{Loc}(H(\ell').result)$. Hence, we have:

$$(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \mathsf{H}(c, \{\!|c; (f \mapsto \hat{v})^*|\!\}) \wedge \mathsf{H}(c', \{\!|c'; (f' \mapsto \hat{v}')^*, parent \mapsto c|\!\}),$$

which allows us to prove:

$$(\!|P|\!) \cup \beta_{Cnf}(\Psi) \vdash \mathsf{H}(c, \{\!|c; (f \mapsto \hat{v})^*[result \mapsto \lambda'']|\!\}),$$

by using the implication *Res*. We then observe that:

$$\begin{array}{rcl} \{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{v})^*[\mathit{result} \mapsto \lambda'']|\!\})\} & :> & \beta_{\mathit{Heap}}(\ell \mapsto H(\ell)[\mathit{result} \mapsto H(\ell').\mathit{result}]) \\ & & \text{by definition} \\ & = & \beta_{\mathit{Heap}}(\ell \mapsto H(\ell)[\mathit{result} \mapsto \ell'']) \\ & & \text{by Lemma 8} \end{array}$$

Since $H' = (H, H'')[\ell \mapsto H(\ell)[\mathit{result} \mapsto \ell'']] = H[\ell \mapsto H(\ell)[\mathit{result} \mapsto \ell'']], H''$, by combining (1), (2) and the last observation using Lemma 5, we conclude as follows:

$$(\!|P|\!) \cup \beta_{\mathit{Cnf}}(\Psi) \vdash \Delta' \cup \{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{v})^*[\mathit{result} \mapsto \lambda'']|\!\})\} :> \beta_{\mathit{Heap}}(H')$$

□

CHAPTER 4

fsHornDroid: A Sound Flow-Sensitive Heap Abstraction for the Static Analysis of Android Applications

4.1 Introduction

There are many relevant security concerns for Android applications, e.g., privilege escalation [FWM+11, BCS13] and component hijacking [LLW+12], but the most important challenge in the area is arguably *information flow control*, since Android applications are routinely granted access to personal information and other sensitive data stored on the device where they are installed. To counter the threats posed by malicious applications, the research community has proposed a plethora of increasingly sophisticated (static) information flow control frameworks for Android [YY12, ZO12, MS12, GCEC12, KYY+12, ARF+14, WROR14, GKP+15, CGM16]. Despite all this progress, however, none of these static analysis tools is able to properly reconcile soundness and precision in its treatment of heap-allocated data structures.

4.1.1 Soundness vs. Precision in Android Analyses

As it was previously mentioned in § 2.3.5, designing a static analysis for Android applications which is both sound and precise on the heap abstraction is very challenging, most notably because the Android ecosystem is highly concurrent, featuring multiple components running in the same application at the same time and sharing part of the heap. More complications come from the scheduling of these components, which is user-driven, e.g., via button clicks, and thus statically unknown. This means that it is hard to devise precise *flow-sensitive* heap abstractions for Android applications without breaking their soundness. Indeed, most existing static analysers for Android applications turn out to be unsound and miss malicious information leaks ingeniously hidden in the control flow: for instance, Table 4.1 shows a leaky code snippet that cannot be detected by FlowDroid [ARF+14], a state-of-the-art taint tracker for Android applications[1].

```
public class Leaky extends Activity {
  Storage st = new Storage();
  Storage st2 = new Storage();
  onRestart() { st2 = st; }
  onResume() { st2.s = getDeviceId(); }
  onPause() { send(st.s, "http://www.myapp.com/"); }
}
```

Table 4.1: A Subtle Information Leak

Assume that the `Storage` class has only one field `s` of type `String`, populated with the empty string by its default constructor. The activity class `Leaky` has two fields `st` and `st2` of type `Storage`. A leak of the device id may be performed in three steps. First, the activity is stopped and then restarted: after the execution of the `onRestart()` callback, `st2` becomes an alias of `st`. Then, the activity is paused and resumed. As a

[1]Android applications are written in Java and compiled to bytecode run by a register-based virtual machine (Dalvik). Most static analysis tools for Android analyse Dalvik bytecode, but we present our examples using a Java-like language to improve readability.

result, the execution of the `onPause()` callback communicates the empty string over the Internet, while the `onResume()` callback stores the device id in `st2` and thus in `st` due to aliasing. Finally, the activity is paused again and the device id is leaked by `onPause()`.

HornDroid presented in Chapter 2 is the sound static analyser for Android applications, as such, it correctly deals with the code snippet in Table 4.1. In order to retain soundness, however, HornDroid is quite conservative on the prediction of the control flow of Android applications and implements a *flow-insensitive* heap abstraction by computing just one static over-approximation of the heap, which is proved to be correct at all reachable program points. This is a significant limitation of the tool, since it prevents *strong updates* [LC11] on heap-allocated data structures and thus negatively affects the precision of the analysis. Concretely, to understand the practical import of this limitation, consider the Java code snippet in Table 4.2.

```
 1 public class Anon extends Activity {
 2   Contact[] m = new Contact[]();
 3   onStart() {
 4     for (int i = 0; i < contacts.length(); i++) {
 5       Contact c = contacts.getContact(i);
 6       c.phone = anonymise(c.phone);
 7       m[i] = c;
 8     }
 9     send(m, "http://www.cool-apps.com/");
10   }
11 }
```

Table 4.2: Anonymizing Contact Information

This code reads the contacts stored on the phone, but then calls the `anonymise` method at line 6 to erase any sensitive information (like phone numbers) before sending the collected data on the Internet. Though this code is benign, HornDroid raises a false alarm, since the field `c.phone` stores sensitive information after line 5 and strong updates of object fields are not allowed by the static analysis implemented in the tool.

Our Contributions In the present chapter we make the following contributions:

- We extend an operational semantics for a core fragment of the Android ecosystem described in Chapter 2 (Table 2.5 and Table 2.6) with multi-threading and exception handling, in order to provide a more accurate representation of the control flow of Android applications;

- We present the first static analysis for Android applications which is both flow-sensitive on the heap abstraction and provably sound with respect to the model above. Our proposal borrows ideas from *recency abstraction* [BR06] in order to hit a sweet spot between precision and efficiency, extending it for the first time to a concurrent setting;

- We implement our analysis as an extension of HornDroid presented in Chapter 2. This extension allows HornDroid to perform strong updates on heap-allocated data structures, thus significantly increasing the precision of the tool;

- We test our extension of HornDroid against DroidBench, a popular benchmark proposed by the research community [ARF+14]. We show that our changes to HornDroid lead to an improvement in the precision of the tool, while having only a moderate cost in terms of efficiency. We also discuss analysis results for 64 real applications to demonstrate the scalability of our approach. Our tool's sources and more details on the experiments are available online [fsh].

4.2 Design and Key Ideas

4.2.1 Our Proposal

Our proposal starts from the pragmatic observation that statically predicting the control flow of an Android application is daunting and error-prone [GKP+15]. For this reason, our analysis simply assumes that all the activities, threads and callbacks of the application to analyse are concurrently executed under an interleaving semantics[2]. In the following paragraphs, we just refer to threads for brevity.

The key observation to recover precision despite this conservative assumption is that the runtime behaviour of a given thread can only invalidate the static approximation of the heap of another thread whenever the two threads share memory. This means that the heap of each thread can be soundly analysed in a flow-sensitive fashion, as long as the thread runs isolated from all other threads. Our proposal refines this intuition and achieves a much higher level of precision by using two separate static approximations of the heap: a *flow-sensitive abstract heap* and a *flow-insensitive abstract heap*.

Abstract objects on the flow-sensitive abstract heap approximate concrete objects which are guaranteed to be local to a single thread (not shared). Moreover, these abstract objects always approximate exactly one concrete object, hence it is sound to perform *strong updates* on them. Abstract objects on the flow-insensitive abstract heap, instead, approximate either (1) one concrete object which may be shared between multiple threads, or (2) multiple concrete objects, e.g., produced by a loop. Thus, abstract objects on the flow-insensitive abstract heap only support *weak updates* to preserve soundness. In case (1), this is a consequence of the analysis conservatively assuming the concurrent

[2]We are aware of the fact that the Java Memory Model allows more behaviours than an interleaving semantics (see [Loc14] for a formalisation), but since its connections with Dalvik depend on the Android version and its definition is very complicated, in this work we just consider an interleaving semantics for simplicity.

execution of all the threads and the corresponding loss of precision on the control flow. In case (2), this follows from the observation that only one of the multiple concrete objects represented by the abstract object is updated at runtime, but the updated abstraction should remain sound for all the concrete objects, including those which are not updated. The analysis moves abstract objects from the flow-sensitive abstract heap to its flow-insensitive counterpart when one of the two invariants of the flow-sensitive abstract heap may be violated: this mechanism is called *lifting*.

Technically, the analysis identifies heap-allocated data structures using their allocation site, like most traditional abstractions [PB09, HL07, LC11, KS13]. Unlike these, however, each allocation site λ is bound to *two* distinct abstract locations: $\mathsf{FS}(\lambda)$ and $\mathsf{NFS}(\lambda)$. We use $\mathsf{FS}(\lambda)$ to access the flow-sensitive abstract heap and $\mathsf{NFS}(\lambda)$ to access the flow-insensitive abstract heap. The abstract location $\mathsf{FS}(\lambda)$ contains the abstraction of the *most-recently-allocated* object created at λ, provided that this object is *local* to the creating thread. Conversely, the abstract location $\mathsf{NFS}(\lambda)$ contains a sound abstraction of all the other objects created at λ.

Similar ideas have been proposed in *recency abstraction* [BR06], but standard recency abstraction only applies to sequential programs, where it is always sound to perform strong updates on the abstraction of the most-recently-allocated object. Our analysis, instead, operates in a concurrent setting and assumes that all the threads are concurrently executed under an interleaving semantics. As we anticipated, this means that, if a pointer may be shared between different threads, performing strong updates on the abstraction of the object indexed by the pointer would be unsound. Our analysis allows strong updates without sacrificing soundness by statically keeping track of a set of pointers which are known to be local to a single thread: only the abstractions of the most-recently-allocated objects indexed by these pointers are amenable for strong updates.

4.2.2 Examples

By being conservative on the execution order of callbacks, our analysis is able to analyse the leaky example of Table 4.1 soundly. We recall it in Table 4.3, where we annotate it with a simplified version of the facts generated by the analysis: the heap fact H provides a *flow-insensitive* heap abstraction, while the Sink fact denotes communication to a sink. We use line numbers to identify allocation sites and to index the heap abstractions.

In our analysis, activity objects are always abstracted in a flow-insensitive way, which is crucial for soundness, since we do not predict the execution order of their callbacks. When the activity is created, an abstract flow-insensitive heap fact $\mathsf{H}(1, \{\!|\texttt{Leaky}; \texttt{st} \mapsto \mathsf{NFS}(2), \texttt{st2} \mapsto \mathsf{NFS}(3)|\!\})$ is introduced, and two facts $\mathsf{H}(2, \{\!|\texttt{Storage}; \texttt{s} \mapsto \texttt{""}|\!\})$ and $\mathsf{H}(3, \{\!|\texttt{Storage}; \texttt{s} \mapsto \texttt{""}|\!\})$ abstract the objects pointed by the activity fields `st` and `st2`. Then the lifecycle events are abstracted: the `onRestart` method performs a weak update on the activity object, adding a fact $\mathsf{H}(1, \{\!|\texttt{Leaky}; \texttt{st} \mapsto \mathsf{NFS}(2), \texttt{st2} \mapsto \mathsf{NFS}(2)|\!\})$ which tracks aliasing; after the `onResume` method, `st` can thus point to two possible objects, as reflected by the abstract flow-insensitive heap facts generated at line 2 and at

```
1 public class Leaky extends Activity {
      H(1, ⦃Leaky; st ↦ NFS(2), st2 ↦ NFS(3)⦄)
      // flow-insensitivity on activity object
2   Storage st = new Storage();
        H(2, ⦃Storage; s ↦ ""⦄) // after the constructor
3   Storage st2 = new Storage();
        H(3, ⦃Storage; s ↦ ""⦄) // after the constructor
4   onRestart() { st2 = st; }
        H(1, ⦃Leaky; st ↦ NFS(2), st2 ↦ NFS(2)⦄) // aliasing
5   onResume() { st2.s = getDeviceId(); }
        H(2, ⦃Storage; s ↦ id⦄) ∧ H(3, ⦃Storage; s ↦ id⦄)
        // due to flow-insensitivity on activity object
6   onPause() { send(st.s, "http://www.myapp.com/");
        Sink("") ∧ Sink(id) // the leak is detected
7   }
8 }
```

Table 4.3: A Subtle Information Leak (Detected)

line 5. Since the latter fact tracks a sensitive value in the field `s`, the leak is caught in `onPause`.

Our analysis can also precisely deal with the benign example of Table 4.2 thanks to recency abstraction. We show a simplified version of the facts generated by the analysis in Table 4.4. If our static analysis only used a traditional allocation-site abstraction, the benefits of flow-sensitivity would be voided by the presence of the "for" loop in the code. Indeed, the allocation site of `c` would need to identify all the concrete objects allocated therein, hence a traditional static analysis could not perform strong updates on `c.phone` without breaking soundness and would raise a false alarm on the code.

The local state fact $\mathsf{LState}_{\mathsf{pp}}$ provides a flow-sensitive abstraction of the state of the registers and the heap at program point *pp*. Recall that activity objects are always abstracted in a flow-insensitive fashion, therefore the `Contact` array `m` is also abstracted by a flow-insensitive heap fact $\mathsf{H}(2, [])$. At each loop iteration, our static analysis abstracts the most-recently-allocated `Contact` object at line 5 in a flow-sensitive fashion. This is done by putting the abstract flow-sensitive location $\mathsf{FS}(5)$ in `c` and by storing the abstraction of the `Contact` object o_c in the flow-sensitive local state abstraction LState_5, using its allocation site 5 as a key. This allows us to perform a strong update on the `c.phone` field at line 6, overwriting the private information with a public one. At line 7 the program stores the public object in the array `m`, which is abstracted by a flow-insensitive heap fact: to preserve soundness, the flow-sensitive abstraction of o_c is *lifted* (downgraded) to a flow-insensitive abstraction by generating a flow-insensitive heap fact $\mathsf{H}(5, o_c[\texttt{phone} \mapsto \texttt{""}])$ and by changing the abstraction of `c` from $\mathsf{FS}(5)$ to $\mathsf{NFS}(5)$. We then perform a weak update on the array stored in `m` by generating a flow-insensitive heap fact $\mathsf{H}(2, [\mathsf{NFS}(5)])$. Thanks to the previous strong update, however, the end result is that `m` only stores public information at the end of the loop and no leak is detected.

```
1 public class Anon extends Activity {
      $\mathsf{H}(1, \{\!|\texttt{Anon;m} \mapsto \mathsf{NFS}(2)|\!\})$
      // flow-insensitivity on activity object
2   Contact[] m = new Contact[]();
        $\mathsf{H}(2, [])$ // new empty array is created
3   onStart() {
        $\mathsf{LState}_3(\texttt{c} \mapsto \texttt{null}; 5 \mapsto \bot)$
        // no allocated contact at location 5 yet
4     for (int i = 0; i < contacts.length(); i++) {
          $\mathsf{LState}_4(\texttt{c} \mapsto \texttt{null}; 5 \mapsto \bot) \wedge \mathsf{LState}_4(\texttt{c} \mapsto \mathsf{NFS}(5); 5 \mapsto \bot)$
          // loop invariant (see below)
5       Contact c = contacts.getContact(i);
            $\mathsf{LState}_5(\texttt{c} \mapsto \mathsf{FS}(5); 5 \mapsto o_c)$ // flow-sensitivity
6       c.phone = anonymise(c.phone);
            $\mathsf{LState}_6(\texttt{c} \mapsto \mathsf{FS}(5); 5 \mapsto o_c\{\texttt{phone} \mapsto \texttt{""}\})$ // strong update
7       m[i] = c;
            $\mathsf{LState}_7(\texttt{c} \mapsto \mathsf{NFS}(5); 5 \mapsto \bot) \wedge \mathsf{H}(5, o_c\{\texttt{phone} \mapsto \texttt{""}\}) \wedge \mathsf{H}(2, [\mathsf{NFS}(5)])$ // lifting is
            performed
8     }
9     send(m, "http://www.cool-apps.com/");
          $\mathsf{Sink}([o_c\{\texttt{phone} \mapsto \texttt{""}\}])$ // no leak is detected
10  }
11 }
```

Table 4.4: Anonymizing Contact Information (Allowed)

4.3 Concrete Semantics

Our static analysis is defined on top of an extension of μ-Dalvik$_A$, a formal model of a core fragment of the Android ecosystem presented in Table 3.3 and Table 3.4. It includes the main bytecode instructions of Dalvik, the register-based virtual machine running Android applications, and a few important API methods. Moreover, it captures the lifecycle of the most common and complex application components (*activities*), as well as inter-component communication based on asynchronous messages (*intents*, with a dictionary-like structure). Our extension of μ-Dalvik$_A$ adds two more ingredients to the model: *multi-threading* and *exceptions*, which are useful to get a full account of the control flow of Android applications. In this section, we focus on a relatively high-level overview of our extensions, later in § 5.1 we provide the formal details, including the full operational semantics.

4.3.1 Basic Syntax

We write $(r_i)^{i \leq n}$ to denote the sequence $r_1, \ldots, r_n$. When the length of the sequence is unimportant, we simply write r^*. Given a sequence r^*, r_j stands for its j-th element and $r^*[j \mapsto r']$ denotes the sequence obtained from r^* by substituting its j-th element with r'. We let $k_i \mapsto v_i$ denote a key-value binding and we represent partial maps using a sequence of key-value bindings $(k_i \mapsto v_i)^*$, where all the keys k_i are pairwise distinct; the order of the keys in a partial map is immaterial.

We introduce in Table 4.5 a few basic syntactic categories. A program P is a sequence of classes. A class $\texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{\mathit{fld}^*; \mathit{mtd}^*\}$ consists of a name c, a super-class c', a sequence of implemented interfaces c^*, a sequence of fields fld^*, and a sequence of methods mtd^*. A method $m : \tau^* \xrightarrow{n} \tau\ \{st^*\}$ consists of a name m, the type of its arguments τ^*, the return type τ, and a sequence of statements st^* defining the method body; the syntax of statements is explained below. The integer n on top of the arrow declares how many registers are used by the method. Observe that field declarations $f : \tau$ include the type of the field. A left-hand side lhs is either a register r, an array cell $r_1[r_2]$, an object field $r.f$, or a static field $c.f$, while a right-hand side rhs is either a left-hand side lhs or a primitive value prim.

P	$::=$	cls^*
cls	$::=$	$\texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{\mathit{fld}^*; \mathit{mtd}^*\}$
τ_{prim}	$::=$	$\texttt{bool} \mid \texttt{int} \mid \dots$
τ	$::=$	$c \mid \tau_{prim} \mid \texttt{array}[\tau]$
fld	$::=$	$f : \tau$
mtd	$::=$	$m : \tau^* \xrightarrow{n} \tau\ \{st^*\}$
lhs	$::=$	$r \mid r[r] \mid r.f \mid c.f$
prim	$::=$	$\texttt{true} \mid \texttt{false} \mid \dots$
rhs	$::=$	$\mathit{lhs} \mid \mathit{prim}$

Table 4.5: Basic Syntactic Categories

Table 4.6 reports the syntax of selected statements, along with a brief intuitive explanation of their semantics. Observe that statements do not operate directly on values, but rather on the content of the registers of the Dalvik virtual machine. The extensions with respect to Chapter 2 are in bold and are discussed in more detail in the following. Some of the next definitions are dependent on a program P, but we do not make this dependency explicit to keep the notation more concise.

4.3.2 Local Reduction

Notation Table 4.7 shows the main semantic domains used in the present section. We let p range over pointers from a countable set *Pointers*. A program point pp is a triple c, m, pc including a class name c, a method name m and a program counter pc (a natural number identifying a specific statement of the method). Annotations λ are auxiliary information with no semantic import, their use in the static analysis is discussed in Section 4.4. A location ℓ is an annotated pointer p_λ and a value v is either a primitive value or a location.

A *local state* $L = \langle pp \cdot u^* \cdot st^* \cdot R\rangle$ stores the state information of an invoked method, run by a given thread or activity. It is composed of a program point pp, identifying the currently executed statement; the method calling context u^*, which keeps track of the method arguments and is only used in the static analysis; the method body st^*, defining

$st ::=$	
`goto` pc	unconditionally jump to program counter pc
`invoke` $r_o\ m\ r^*$	invoke method m of the object in r_o with args r^*
`if`$_{\otimes}$ $r_1\ r_2$ `then` pc	jump to program counter pc if $r_1 \otimes r_2$
`return`	get the value of the special return register r_{res}
`move` $lhs\ rhs$	move rhs into lhs
`newintent` $r_i\ c$	put a pointer to a new intent for class c in r_i
`unop`$_{\odot}$ $r_d\ r_s$	compute $\odot r_s$ and put the result in r_d
`put-extra` $r_i\ r_k\ r_v$	bind the value of r_v to key r_k of the intent in r_i
`binop`$_{\oplus}$ $r_d\ r_1\ r_2$	compute $r_1 \oplus r_2$ and put the result in r_d
`get-extra` $r_i\ r_k\ \tau$	get the τ-value bound to key r_k of the intent in r_i
`new` $r_d\ c$	put a pointer to a new object of class c in r_d
`start-act` r_i	start a new activity by sending the intent in r_i
`newarray` $r_d\ r_l\ \tau$	put a pointer to a new τ-array of length r_l in r_d
`start-thread` r_t	start the thread in r_t
`throw` r_e	throw the exception stored in r_e
`interrupt` r_t	interrupt the thread in r_t
`move-except` r_e	store a pointer to the last thrown exception in r_e
`join` r_t	join the current thread with the thread in r_t

Table 4.6: Syntax and Informal Semantics of Selected Statements

the method implementation; and a register state R, mapping registers to their content. Registers are local to a given method invocation.

A *local state list* $L^{\#}$ is a list of local states. It is used to keep track of the state information of all the methods invoked by a given thread or activity. The *call stack* α is modeled as a local state list $L^{\#}$, possibly qualified by the `AbNormal`$(\cdot)$ modifier if the thread or activity is recovering from an exception.

Coming to memory, we define the *heap* H as a partial map from locations to *memory blocks*. There are three types of memory blocks in the formalism: objects, arrays and intents. An *object* $o = \{\!|c; (f_\tau \mapsto v)^*|\!\}$ stores its class c and a mapping between fields and values. Fields are annotated with their type, which is typically omitted when unneeded. An *array* $a = \tau[v^*]$ contains the type τ of its elements and the sequence of the values v^* stored into it. An *intent* $i = \{\!|@c; (k \mapsto v)^*|\!\}$ is composed by a class name c, identifying the intent recipient, and a sequence of key-value bindings $(k \mapsto v)^*$, defining the intent payload (a dictionary). The *static heap* S is a partial map from static fields to values.

Finally, we have *local configurations* $\Sigma = \ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$, representing the full state of a specific activity or thread. They include a location ℓ, pointing to the corresponding activity or thread object; a call stack α; a pending activity stack π, which is a list of intents keeping track of all the activities that have been started; a pending thread stack

Pointers	p	$\in$	$\mathit{Pointers}$
Program counters	pc	$\in$	$\mathbb{N}$
Program points	pp	$::=$	c, m, pc
Annotations	λ	$::=$	$pp \mid c \mid in(c)$
Locations	ℓ	$::=$	p_λ
Values	u, v	$::=$	$prim \mid \ell$
Register states	R	$::=$	$(r \mapsto v)^*$
Local states	L	$::=$	$\langle pp \cdot u^* \cdot st^* \cdot R \rangle$
Local state lists	$L^\#$	$::=$	$\varepsilon \mid L :: L^\#$
Call stacks	α	$::=$	$L^\# \mid \texttt{AbNormal}(L^\#)$
Objects	o	$::=$	$\{\!|c; (f_\tau \mapsto v)^*|\!\}$
Arrays	a	$::=$	$\tau[v^*]$
Intents	i	$::=$	$\{\!|@c; (k \mapsto v)^*|\!\}$
Memory blocks	b	$::=$	$o \mid a \mid i$
Heaps	H	$::=$	$(\ell \mapsto b)^*$
Static heaps	S	$::=$	$(c.f \mapsto v)^*$
Pending activity stacks	π	$::=$	$\varepsilon \mid i :: \pi$
Pending thread stacks	γ	$::=$	$\varepsilon \mid \ell :: \gamma$
Local configurations	Σ	$::=$	$\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$

Table 4.7: Semantic Domains for Local Reduction

γ, which is a list of pointers to the threads which have been started; a heap H, storing memory blocks; and a static heap S, storing the values of static fields.

We use several substitution notations in the reduction rules, with an obvious meaning. The only non-standard notations are Σ^+, which stands for Σ where the value of pc is replaced by $pc + 1$ in the top-most local state of the call stack, and the substitution of registers $\Sigma[r_d \mapsto u]$, which sets the value of the register r_d to u in the top-most local state of the call stack. This reflects the idea that the computation is performed on the local state of the last invoked method.

Local Reduction Relation The *local reduction* relation $\Sigma \rightsquigarrow \Sigma'$ models the evolution of a local configuration Σ into a new local configuration Σ' as the result of a computation step. The definition of the local reduction relation uses two auxiliary relations:

- $\Sigma[\![rhs]\!]$, which evaluates a right-hand side expression rhs in the local configuration Σ;
- $\Sigma, st \Downarrow \Sigma'$, which executes the statement st on the local configuration Σ to produce Σ'.

The most straightforward rule defining a local reduction step $\Sigma \rightsquigarrow \Sigma'$ just fetches the next statement st to run and performs a look-up on the auxiliary relation $\Sigma, st \Downarrow \Sigma'$. Formally, assuming a function $\mathit{get\text{-}stm}(\Sigma)$ fetching the next statement based on the program counter of the top-most local state in Σ, we have:

$$\textsc{(R-NextStm)}\quad \frac{\Sigma, \mathit{get\text{-}stm}(\Sigma) \Downarrow \Sigma'}{\Sigma \rightsquigarrow \Sigma'}$$

We show a subset of the new local reduction rules added to μ-Dalvik$_A$ in Table 4.8 and we explain them below.

Exception Rules In Dalvik, method bodies can contain special annotations for exception handling, specifying which exceptions are caught and where, as well as the program counter of the corresponding exception handler (handlers are part of the method body). In our formalism, we assume the existence of a partial map $\mathsf{ExcptTable}(pp, c) = pc$ which provides, for all program points pp where exceptions can be thrown and for all classes c extending the $\mathsf{Throwable}$ interface, the program counter pc of the corresponding exception handler. If no handler exists, then $\mathsf{ExcptTable}(pp, c) = \bot$. Moreover, all local states contain a special register r_{excpt} that is only accessed by the exception handling rules: this stores the location of the last thrown exception.

An exception object stored in r_e can be thrown by the statement `throw` r_e using rule (R-Throw): it checks that r_e contains the location of a (throwable) object, stores this location into the register r_{excpt} and moves the local configuration into an abnormal state. After entering an abnormal state, there are two possibilities: if there exists a handler for the thrown exception, we exit the abnormal state and jump to the program counter of the exception handler using rule (R-Caught); otherwise, the exception is thrown back to the method caller using rule (R-UnCaught). Finally, the location of the last thrown exception object can be copied from the register r_{excpt} into the register r_e by the statement `move-except` r_e, as formalized by rule (R-MoveException)

Thread Rules Our formalism covers the core methods of the Java Thread API [Javb]: they enable thread spawning and thread communication by means of interruptions and synchronizations. Rule (R-StartThread) models the statement `start-thread` r_t: it allows a thread to be started by simply pushing the location of the thread object stored in r_t on the pending thread stack. The actual execution of the thread is left to the virtual machine, which will spawn it at an unpredictable point in time, as we discuss in the next section. The statement `interrupt` r_t sets the interrupt field (named inte) of the thread object whose location is stored in r_t to **`true`**, as formalized by rule (R-InterruptThread). We now describe the semantics of thread synchronizations. If the thread t' calling `join` r_t was not interrupted at some point, rule (R-JoinThread) checks whether the thread whose location is stored in r_t has finished; if this is the case, it resumes the execution of t', otherwise t' remains stuck. If instead t' was interrupted before calling `join` r_t, rule (R-InterruptJoin) performs the following operations: the

(R-Throw)

$$\frac{\ell = \Sigma[\![r_e]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\}}{\Sigma, \mathtt{throw}\ r_e \Downarrow \Sigma[\alpha \mapsto \mathtt{AbNormal}(\alpha)][r_{\mathsf{excpt}} \mapsto \ell]}$$

(R-Caught)

$$\frac{\begin{array}{c}\ell = \Sigma_A[\![r_{\mathsf{excpt}}]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\} \\ \mathsf{ExcptTable}(c, m, pc, c') = pc' \qquad \alpha_c = \langle c, m, pc' \cdot u^* \cdot st^* \cdot R\rangle :: \alpha'\end{array}}{\Sigma_A \rightsquigarrow \Sigma_A[\alpha_A \mapsto \alpha_c]}$$

(R-UnCaught)

$$\frac{\begin{array}{c}\ell = \Sigma_A[\![r_{\mathsf{excpt}}]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\} \\ \mathsf{ExcptTable}(c, m, pc, c') = \bot\end{array}}{\Sigma_A \rightsquigarrow \Sigma_A[\alpha_A \mapsto \mathtt{AbNormal}(\alpha')][r_{\mathsf{excpt}} \mapsto \ell]}$$

(R-MoveException)

$$\frac{\ell = \Sigma[\![r_{\mathsf{excpt}}]\!]}{\Sigma, \mathtt{move\text{-}except}\ r_e \Downarrow \Sigma^+[r_e \mapsto \ell]}$$

(R-StartThread)

$$\frac{\ell = \Sigma[\![r_t]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\} \qquad \gamma' = \ell :: \gamma}{\Sigma, \mathtt{start\text{-}thread}\ r_t \Downarrow \Sigma^+[\gamma \mapsto \gamma']}$$

(R-InterruptThread)

$$\frac{\begin{array}{c}\ell = \Sigma[\![r_t]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \mathsf{inte} \mapsto \_|\!\} \\ H' = H[\ell \mapsto \{\!|c'; (f \mapsto v)^*, \mathsf{inte} \mapsto \mathtt{true}|\!\}]\end{array}}{\Sigma, \mathtt{interrupt}\ r_t \Downarrow \Sigma^+[H \mapsto H']}$$

(R-JoinThread)

$$\frac{\begin{array}{c}H(\ell_r) = \{\!|c_r; (f_r \mapsto v_r)^*, \mathsf{inte} \mapsto \mathtt{false}|\!\} \\ \ell = \Sigma[\![r_t]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \mathsf{finished} \mapsto \mathtt{true}|\!\}\end{array}}{\Sigma, \mathtt{join}\ r_t \Downarrow \Sigma^+}$$

(R-InterruptJoin)

$$\frac{\begin{array}{c}H(\ell_r) = \{\!|c_r; (f_r \mapsto v_r)^*, \mathsf{inte} \mapsto \mathtt{true}|\!\} \\ o = \{\!|c_r; (f_r \mapsto v_r)^*, \mathsf{inte} \mapsto \mathtt{false}|\!\} \qquad p_{c,m,pc} \notin dom(H) \\ H' = H, p_{c,m,pc} \mapsto \{\!|\mathsf{IntExcpt};|\!\} \qquad \alpha_c = \mathtt{AbNormal}(\alpha[r_{\mathsf{excpt}} \mapsto p_{c,m,pc}])\end{array}}{\Sigma, \mathtt{join}\ r_t \Downarrow \Sigma[\alpha \mapsto \alpha_c, H \mapsto H'[\ell_r \mapsto o]]}$$

Convention: let $\Sigma = \ell_r \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha = \langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle :: \alpha'$ and $\Sigma_A = \ell_r \cdot \alpha_A \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha_A = \mathtt{AbNormal}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle :: \alpha')$.

Table 4.8: Small step semantics of extended μ-Dalvik$_A$ - Excerpt

inte field of t' is reset to `false`, an IntExcpt exception is thrown (this creates a new exception object) and the local configuration enters an abnormal state.

4.3.3 Global Reduction

Notation Table 4.9 introduces the main semantic domains used in the present section. First, we assume the existence of a set of activity states *ActStates*, which is used to model the Android activity lifecycle (see [PS14]). Then we have two kinds of *frames*, modelling running processes. An *activity frame* $\varphi = \langle \ell, s, \pi, \gamma, \alpha \rangle$ describes the state of an activity: it includes a location ℓ, pointing to the activity object; the activity state s; a pending activity stack π, representing other activities started by the activity; a pending thread stack γ, representing threads spawned by the activity; and a call stack α. A *thread frame* $\psi = \langle\!\langle \ell, \ell', \pi, \gamma, \alpha \rangle\!\rangle$ describes a running thread: it includes a location ℓ, pointing to the activity object that started the thread; a location ℓ' pointing to the thread object; a pending activity stack π, representing activities started by the thread; a pending thread stack γ, representing other threads spawned by the thread; and a call stack α.

Activity frames are organized in an *activity stack* Ω, containing all the running activities; one of the activities may be singled out as *active*, represented by an underline, and it is scheduled for execution. We assume that each Ω contains at most one underlined activity frame. Thread frames, instead, are organized in a *thread pool* Ξ, containing all the running threads. A *configuration* $\Psi = \Omega \cdot \Xi \cdot H \cdot S$ includes an activity stack Ω, a thread pool Ξ, a heap H and a static heap S. It represents the full state of an Android application.

Activity states	s	$\in$	*ActStates*
Activity frames	φ	$::=$	$\langle \ell, s, \pi, \gamma, \alpha \rangle \mid \underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}$
Activity stacks	Ω	$::=$	$\varphi \mid \varphi :: \Omega$
Thread frames	ψ	$::=$	$\langle\!\langle \ell, \ell', \pi, \gamma, \alpha \rangle\!\rangle$
Thread pools	Ξ	$::=$	$\emptyset \mid \psi :: \Xi$
Configurations	Ψ	$::=$	$\Omega \cdot \Xi \cdot H \cdot S$

Table 4.9: Semantic Domains for Global Reduction

Global Reduction Relation The *global reduction* relation $\Psi \Rightarrow \Psi'$ models the evolution of a configuration Ψ into a new configuration Ψ', either by executing a statement in a thread or activity according to the local reduction rules, or as the result of processing lifecycle events of the Android platform, including user inputs, system callbacks, inter-component communication, etc.

Before presenting the global reduction rules, we define a few auxiliary notions. First, we let *lookup* be the function such that $lookup(c, m) = (c', st^*)$ iff c' is the class obtained when performing dispatch resolution of the method m on an object of type c and st^* is the corresponding method body. Then, we assume a function *sign* such that

$sign(c, m) = \tau^* \xrightarrow{n} \tau$ iff there exists a class cls_i such that $cls_i = \texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{fld^*;$ $mtd^*, m : \tau^* \xrightarrow{n} \tau\ \{st^*\}\}$. Finally, we let a *successful* call stack be the call stack of an activity or thread which has completed its computation, as formalized by the following definition.

Definition 11. A call stack α is *successful* if and only if $\alpha = \langle pp \cdot u^* \cdot \texttt{return} \cdot R \rangle :: \varepsilon$ for some pp, u^* and R. We let $\overline{\alpha}$ range over successful call stacks.

The core of the global reduction rules is taken from Table 2.5 and Table 2.6, extended with a few simple rules used, e.g., to manage the thread pool. The main new rules are given in Table 4.10 and the full set can be found in § 5.1. We start by describing rule (A-ThreadStart), which models the starting of a new thread by some activity. Let ℓ' be a pointer to a pending thread spawned by an activity identified by the pointer ℓ, the rule instantiates a new thread frame $\psi = \langle\!\langle \ell, \ell', \varepsilon, \varepsilon, \alpha' \rangle\!\rangle$ with empty pending activity stack and empty pending thread stack, executing the run method of the thread object referenced by ℓ'. We then have two other rules: rule (T-Reduce) allows the reduction of any thread in the thread pool, using the reduction relation for local configurations; rule (T-Kill) allows the system to remove a thread which has finished its computations, by checking that its call stack is successful.

(A-ThreadStart)

$$\frac{\begin{array}{c}\varphi = \underline{\langle \ell, s, \pi, \gamma :: \ell' :: \gamma', \alpha \rangle} \qquad \varphi' = \underline{\langle \ell, s, \pi, \gamma :: \gamma', \alpha \rangle} \\ \psi = \langle\!\langle \ell, \ell', \varepsilon, \varepsilon, \alpha' \rangle\!\rangle \qquad H(\ell') = \{\!| c'; (f \mapsto v)^* |\!\} \qquad lookup(c', \mathsf{run}) = (c'', st^*) \\ sign(c'', \mathsf{run}) = \mathsf{Thread} \xrightarrow{loc} \mathsf{Void} \qquad \alpha' = \langle c'', \mathsf{run}, 0 \cdot \ell' \cdot st^* \cdot (r_k \mapsto \mathbf{0})^{k \leq loc}, r_{loc+1} \mapsto \ell' \rangle\end{array}}{\Omega :: \varphi :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \varphi' :: \Omega' \cdot \psi :: \Xi \cdot H \cdot S}$$

(T-Reduce)

$$\frac{\ell_t \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S \rightsquigarrow \ell_t \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S'}{\Omega \cdot \Xi :: \langle\!\langle \ell, \ell_t, \pi, \gamma, \alpha \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega \cdot \Xi :: \langle\!\langle \ell, \ell_t, \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H' \cdot S'}$$

(T-Kill)

$$\frac{H(\ell') = \{\!| c; (f \mapsto v)^*, \mathsf{finished} \mapsto \_ |\!\} \qquad H' = H[\ell' \mapsto \{\!| c; (f \mapsto v)^*, \mathsf{finished} \mapsto \texttt{true} |\!\}]}{\Omega \cdot \Xi :: \langle\!\langle \ell, \ell', \varepsilon, \varepsilon, \overline{\alpha} \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega \cdot \Xi :: \Xi' \cdot H' \cdot S}$$

Table 4.10: New Global Reduction Rules - Excerpt

4.4 Abstract Semantics

Our analysis takes as input a program P and generates a set of Horn clauses $(\!| P |\!)$ that over-approximate the concrete semantics of P. We can then use an automated theorem prover such as *z3* [dMB08b] to show that $(\!|P|\!)$, together with a set of facts Δ over-approximating the initial state of the program, does not entail a formula ϕ representing the reachability of some undesirable program state (e.g., leaking sensitive information). By the over-approximation, the unsatisfiability of the formula ensures that also P does not reach such a program state.

4.4.1 Syntax of Terms

We assume two disjoint countable sets of variables *Vars* and *BVars*. The syntax of the *terms* of the abstract semantics is defined in Table 4.11 and described below.

Boolean variables	x_b	$\in$	BVars
Variables	x	$\in$	Vars
Abstract elements	$\hat{d}$	$\in$	$\hat{D}$
Booleans	bb	$::=$	$0 \mid 1 \mid x_b$
Abstract locations	$\hat{\lambda}$	$::=$	$\mathsf{FS}(\lambda) \mid \mathsf{NFS}(\lambda)$
Abstract values	$\hat{u}, \hat{v}$	$::=$	$\hat{d} \mid x \mid f(\hat{v}^*)$
Abstract objects	$\hat{o}$	$::=$	$\{\!|c; (f_\tau \mapsto \hat{v})^*|\!\}$
Abstract arrays	$\hat{a}$	$::=$	$\tau[\hat{v}]$
Abstract intents	$\hat{i}$	$::=$	$\{\!|@c; \hat{v}|\!\}$
Abstract blocks	$\hat{b}$	$::=$	$\hat{o} \mid \hat{a} \mid \hat{i}$
Abstract flow-sensitive blocks	$\hat{l}$	$::=$	$\hat{b} \mid \bot$
Abstract flow-sensitive heap	$\hat{h}$	$::=$	$(pp \mapsto \hat{l})^*$
Abstract filter	$\hat{k}$	$::=$	$(pp \mapsto bb)^*$

Table 4.11: Syntax of Terms

Each location p_λ is abstracted by an *abstract location* $\hat{\lambda}$, which is either an abstract flow-sensitive location $\mathsf{FS}(\lambda)$ or an abstract flow-insensitive location $\mathsf{NFS}(\lambda)$. Recall the syntax of annotations: in the concrete semantics, $\lambda = c$ means that p_λ stores an activity of class c; $\lambda = in(c)$ means that p_λ stores an intent received by an activity of class c; and $\lambda = pp$ means that p_λ stores a memory block (object, array or intent) created at program point pp. Only the latter elements are amenable for a sound flow-sensitive analysis, since activity objects are shared by all the activity callbacks and received intents are shared between at least two activities, but the analysis assumes the concurrent execution of all callbacks and activities.

The analysis assumes a bounded lattice $(\hat{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ for approximating concrete values such that the abstract domain $\hat{D}$ contains at least all the abstract locations $\hat{\lambda}$ and the abstractions $\widehat{prim}$ of any primitive value $prim$. We also assume a set of interpreted functions f, containing at least sound over-approximations $\hat{\odot}, \hat{\oplus}, \hat{\otimes}$ of the unary, binary and comparison operators $\odot, \oplus, \otimes$. Abstract values $\hat{v}$ are elements $\hat{d}$ of the abstract domain $\hat{D}$, variables x from *Vars* or function applications of the form $f(\hat{v}^*)$.

The abstraction of objects $\hat{o}$ is field-sensitive, while the abstraction of arrays $\hat{a}$ and intents $\hat{\imath}$ is field-insensitive. The reason is that the structure of objects is statically known thanks to their type, while array lengths and intent fields (strings) may only be known at runtime. It would clearly be possible to use appropriate abstract domains to have a more precise representation of array lengths and intent fields, but we do not do it for the sake of simplicity. An *abstract block* $\hat{b}$ can be an abstract object $\hat{o}$, an abstract array $\hat{a}$ or an abstract intent $\hat{\imath}$. An abstract *flow-sensitive* heap $\hat{h}$ is a total mapping from the set of allocation sites pp to abstract memory blocks $\hat{b}$ or the symbol $\perp$, representing the lack of a flow-sensitive abstraction of the memory blocks created at pp.

There is just one syntactic element in Table 4.11 which we did not discuss yet: *abstract filters*. Abstract filters $\hat{k}$ are total mappings from the set of allocation sites pp to boolean flags bb. They are technically needed to keep track of the allocation sites whose memory blocks must be downgraded to a flow-insensitive analysis when returning from a method call. The downgrading mechanism, called *lifting* of an allocation site, is explained in Section 4.4.3.

4.4.2 Ingredients of the Analysis

Overview Our analysis is *context-sensitive*, which means that the abstraction of the elements in the call stack keeps track of a representation of their calling context. In this work, contexts are defined as tuples $(\hat{\lambda}_t, \hat{u}^*)$, where $\hat{\lambda}_t$ is an abstraction of the location storing the thread or activity which called the method, while $\hat{u}^*$ is an abstraction of the method arguments. Abstracting the calling thread or activity increases the precision of the analysis, in particular when dealing with the `join` r_t statement for thread synchronization.

Moreover, our analysis is *flow-sensitive* and computes a different over-approximation $\hat{h}$ of the state of the heap at each reachable program point, satisfying the following invariant: for each allocation site pp, if $\hat{h}(pp) = \hat{b}$, then $\hat{b}$ is an over-approximation of the most-recently allocated memory block at pp and this memory block is local to the allocating thread or activity. Otherwise, $\hat{h}(pp) = \perp$ and the memory blocks allocated at pp, if any, do not admit a flow-sensitive analysis. These memory blocks are then abstracted by an abstract *flow-insensitive* heap, defining an over-approximation of the state of the heap which is valid at all reachable program points. As such, the abstract flow-insensitive heap is not indexed by a program point.

We present selected excerpts of the analysis in the remaining of this section: the full analysis specification is given in § 5.2.

Analysis Facts The syntax of the analysis *facts* f is defined in Table 4.12. The fact $\mathsf{LState}_{\mathsf{c,m,pc}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ is used to abstract local states: it denotes that, if the method m of the class c is invoked in the context $(\hat{\lambda}_t, \hat{u}^*)$, the state of the registers at the pc-th statement is over-approximated by $\hat{v}^*$, while $\hat{h}$ provides a flow-sensitive abstraction of the state of the heap and $\hat{k}$ tracks the set of the allocation sites which must be lifted after returning from the method. The fact $\mathsf{AState}_{\mathsf{c,m,pc}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ has an analogous meaning, but it abstracts local states trying to recover from an exception. The fact $\mathsf{Res}_{\mathsf{c,m}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}; \hat{h}; \hat{k})$ states that, if the method m of the class c is invoked in the context $(\hat{\lambda}_t, \hat{u}^*)$, its return value is over-approximated by $\hat{v}$; the information $\hat{h}$ and $\hat{k}$ has the same meaning as before and it is used to update the abstract state of the caller after returning from the method m. The fact $\mathsf{Uncaught}_{\mathsf{c,m,pc}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}; \hat{h}; \hat{k})$ ensures that, if the method m of the class c is invoked in the context $(\hat{\lambda}_t, \hat{u}^*)$, it throws an uncaught exception at the pc-th statement and the location of the exception object is over-approximated by $\hat{v}$; here, $\hat{h}$ and $\hat{k}$ are needed to update the abstract state of the caller of m, which becomes in charge of handling the uncaught exception. The fact $\mathsf{RHS}_{\mathsf{pp}}(\hat{v})$ states that $\hat{v}$ over-approximates the right-hand side of a `move` *lhs rhs* statement at program point *pp*.

f ::=	
$\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}, \hat{v}^*); \hat{v}^*; \hat{h}; \hat{k})$	Abstract local state
$\mathsf{AState}_{\mathsf{pp}}((\hat{\lambda}, \hat{v}^*); \hat{v}^*; \hat{h}; \hat{k})$	Abstract abnormal state
$\mathsf{Res}_{\mathsf{c,m}}((\hat{\lambda}, \hat{v}^*); \hat{v}; \hat{h}; \hat{k})$	Abstract result of method call
$\mathsf{Uncaught}_{\mathsf{pp}}((\hat{\lambda}, \hat{v}^*); \hat{v}; \hat{h}; \hat{k})$	Abstract uncaught exception
$\mathsf{RHS}_{\mathsf{pp}}(\hat{v})$	Abstract value of right-hand side
$\mathsf{LiftHeap}(\hat{h}; \hat{k})$	Abstract heap lifting
$\mathsf{Reach}(\hat{v}; \hat{h}; \hat{k})$	Abstract heap reachability
$\mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \hat{\lambda}; \hat{b})$	Abstract heap look-up
$\mathsf{H}(\lambda, \hat{b})$	Abstract flow-insensitive heap entry
$\mathsf{S}_{\mathsf{c,f}}(\hat{v})$	Abstract static field
$\mathsf{I}_{\mathsf{c}}(\hat{i})$	Abstract pending activity
$\mathsf{T}(\lambda, \hat{o})$	Abstract pending thread
$\hat{u} \sqsubseteq \hat{v}$	Partial ordering on abstract values
$\tau \leq \tau'$	Subtyping fact

Table 4.12: Analysis Facts

We then have a few facts used to abstract the heap and lift the allocation sites. The facts $\mathsf{LiftHeap}(\hat{h}; \hat{k})$, $\mathsf{Reach}(\hat{v}; \hat{h}; \hat{k})$ and $\mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \hat{\lambda}; \hat{b})$ are the most complicated and peculiar, so they are explained in detail later on. The fact $\mathsf{H}(\lambda, \hat{b})$ models the abstract flow-insensitive heap: it states that the location p_λ stores a memory block over-approximated by $\hat{b}$ at some point in the program execution. The fact $\mathsf{S}_{\mathsf{c,f}}(\hat{v})$ states that the static field f of class c contains a value over-approximated by $\hat{v}$ at some point of the program execution.

Finally, the fact $\mathsf{I}_{\mathsf{c}}(\hat{i})$ tracks that an activity of class c has sent an intent over-approximated by $\hat{i}$. The fact $\mathsf{T}(\lambda, \hat{o})$ tracks that an activity or thread has started a new thread stored at some location p_λ and over-approximated by $\hat{o}$. We then have standard partial order facts $\hat{u} \sqsubseteq \hat{v}$ and subtyping facts $\tau \leq \tau'$.

Horn Clauses We define *Horn clauses* as logical formulas of the form $\forall x_1, \ldots, \forall x_m.\mathsf{f}_1 \wedge \ldots \wedge \mathsf{f}_n \implies \mathsf{f}$ without free variables. In order to improve readability, we always omit the universal quantifiers in front of Horn clauses and we distinguish constants from universally quantified variables by using a $\mathsf{sans\ serif}$ font for constants, e.g., we write c to denote some specific class c. When an element in a Horn clause is unimportant, we just replace it with an underscore (_). Also, we write $\forall x_1, \ldots, \forall x_m.\mathsf{f}_1 \wedge \ldots \wedge \mathsf{f}_n \implies \mathsf{f}'_1 \wedge \ldots \wedge \mathsf{f}'_k$ for the set $\{\forall x_1, \ldots, \forall x_m.\mathsf{f}_1 \wedge \ldots \wedge \mathsf{f}_n \implies \mathsf{f}'_i \mid i \in [1, k]\}$.

Abstract Programs We define *abstract programs* Δ as sets of facts and Horn clauses, where facts over-approximate program states, while Horn clauses over-approximate the concrete semantics of the analysed program.

4.4.3 The Lifting Mechanism

The *lifting* mechanism is the central technical contribution of the static analysis. It is convenient to abstract for a moment from the technical details and explain it in terms of three separate sequential steps, even though in practice these steps are interleaved together upon Horn clause resolution.

Computing the Abstract Filter Let pp_a be the allocation site to lift, i.e., assume that the most-recently-allocated memory block b at pp_a must be downgraded to a flow-insensitive analysis, for example, because it was shared with another activity or thread. Hence, all the memory blocks which can be reached by following a chain of locations (pointers) starting from any location in b must also be downgraded for soundness. In the analysis, we over-approximate this set of locations with facts of the form $\mathsf{Reach}(\hat{v}; \hat{h}; \hat{k})$, meaning that the abstract filter $\hat{k}$ represents a subset of the flow-sensitive abstract locations which are reachable along $\hat{h}$ from any flow-sensitive abstract location over-approximated by $\hat{v}$. The Horn clauses deriving $\mathsf{Reach}(\hat{v}; \hat{h}; \hat{k})$ are in Table 4.13 and should be read as a recursive computation, whose goal is to find the set of all the abstract flow-sensitive locations reachable from $\hat{v}$ and hence a sound over-approximation of the set of the allocation sites which need to be lifted. The definition uses the function $\hat{k} \,\hat{\sqcup}\, \hat{k}'$, computing the point-wise maximum between $\hat{k}$ and $\hat{k}'$.

Performing the Lifting Once $\mathsf{Reach}(\mathsf{FS}(pp_a); \hat{h}; \hat{k})$ has been recursively computed, the analysis introduces a fact $\mathsf{LiftHeap}(\hat{h}; \hat{k})$ to force the lifting of the allocation sites pp such that $\hat{k}(pp) = 1$, moving their abstract blocks from the abstract flow-sensitive heap $\hat{h}$ to the abstract flow-insensitive heap. The lifting is formalized by the following Horn clause:

$$\mathsf{LiftHeap}(\hat{h}; \hat{k}) \wedge \hat{k}(\mathsf{pp}) = 1 \wedge \hat{h}(\mathsf{pp}) = \hat{b} \implies \mathsf{H}(\mathsf{pp}; \hat{b})$$

Housekeeping Finally, we need to update the data structures used by the analysis to reflect the lifting, using the computed abstract filter $\hat{k}$ to update:

$$\mathsf{Reach}(\widehat{prim}; \hat{h}; 0^*) \qquad \mathsf{Reach}(\mathsf{NFS}(\lambda); \hat{h}; 0^*) \qquad \mathsf{Reach}(\mathsf{FS}(\mathsf{pp}); \hat{h}; 0^*[\mathsf{pp} \mapsto 1])$$

$$\mathsf{Reach}(\hat{u}; \hat{h}; \hat{k}) \wedge \hat{u} \sqsubseteq \hat{v} \implies \mathsf{Reach}(\hat{v}; \hat{h}; \hat{k})$$

$$\mathsf{Reach}(\hat{v}; \hat{h}; \hat{k}) \wedge \mathsf{Reach}(\hat{v}; \hat{h}; \hat{k}') \implies \mathsf{Reach}(\hat{v}; \hat{h}; \hat{k} \mathbin{\hat{\sqcup}} \hat{k}')$$

$$\left.\begin{array}{r} \hat{h}(\mathsf{pp}) = \{\!|c; \_, f \mapsto \hat{v}|\!\} \\ \hat{h}(\mathsf{pp}) = \tau[\hat{v}] \\ \hat{h}(\mathsf{pp}) = \{\!|@c; \hat{v}|\!\} \end{array}\right\} \wedge \mathsf{Reach}(\hat{v}; \hat{h}; \hat{k}) \implies \mathsf{Reach}(\mathsf{FS}(\mathsf{pp}); \hat{h}; \hat{k})$$

Table 4.13: Horn Clauses Used to Derive the Predicate $\mathsf{Reach}(\hat{v}; \hat{h}; \hat{k})$

1. the current abstraction of the registers $\hat{v}^*$. This is done by using a function $\mathsf{lift}(\hat{v}^*; \hat{k})$, which updates $\hat{v}^*$ so that all the abstract flow-sensitive locations $\mathsf{FS}(pp)$ such that $\hat{k}(pp) = 1$ are changed to $\mathsf{NFS}(pp)$. This ensures that the next abstract heap accesses via the register abstractions perform a look-up on the abstract flow-insensitive heap for lifted allocation sites. Formally, we require the lift function to satisfy the axioms in Table 4.14;

$$\frac{\hat{k}(pp) = 0}{\mathsf{lift}(\mathsf{FS}(pp); \hat{k}) = \mathsf{FS}(pp)} \qquad \frac{\hat{k}(pp) = 1}{\mathsf{lift}(\mathsf{FS}(pp); \hat{k}) = \mathsf{NFS}(pp)} \qquad \mathsf{lift}(\mathsf{NFS}(\lambda); \hat{k}) = \mathsf{NFS}(\lambda)$$

$$\mathsf{lift}(\widehat{prim}; \hat{k}) = \widehat{prim} \qquad \frac{\hat{u} \sqsubseteq \hat{v}}{\mathsf{lift}(\hat{u}; \hat{k}) \sqsubseteq \mathsf{lift}(\hat{v}; \hat{k})} \qquad \frac{\forall i : \mathsf{lift}(\hat{v}_i; \hat{k})) = \hat{u}_i}{\mathsf{lift}(\hat{v}^*; \hat{k}) = \hat{u}^*}$$

Table 4.14: Axioms Required on the Function $\mathsf{lift}(\hat{v}^*; \hat{k})$

2. the current abstract flow-sensitive heap $\hat{h}$. This is done by the function $\mathsf{hlift}(\hat{h}; \hat{k})$, which replaces all the entries of the form $pp \mapsto \hat{b}$ in $\hat{h}$ with $pp \mapsto \perp$ if $\hat{k}(pp) = 1$, thus invalidating their flow-sensitive abstraction. If $\hat{k}(pp) = 0$, instead, the function calls $\mathsf{lift}(\hat{v}; \hat{k})$ on all the abstract values $\hat{v}$ occurring in $\hat{b}$, so that $\hat{b}$ itself is still analysed in a flow-sensitive fashion, but it is correctly updated to reflect the lifting of its sub-components;

3. the current abstract filter $\hat{k}'$. This is done by the function $\hat{k} \mathbin{\hat{\sqcup}} \hat{k}'$, computing the point-wise maximum between $\hat{k}$ and $\hat{k}'$. This tracks the allocation sites which must be lifted upon returning from the current method call, so that also the caller can correctly update the abstraction of its registers by using the lift function.

For simplicity, we just say that we lift some abstract value $\hat{v}$ when we lift all the allocation sites pp such that $\mathsf{FS}(pp) \sqsubseteq \hat{v}$.

Example Assume integers are abstracted by their sign and consider the following abstract flow-sensitive heap:

$$\begin{aligned}\hat{h} = \quad & pp_1 \mapsto \tau[\mathsf{FS}(pp_2)], pp_2 \mapsto \{\!|c; g \mapsto \mathsf{FS}(pp_1), g' \mapsto +|\!\}\\ & pp_3 \mapsto \{\!|c'; f \mapsto \mathsf{NFS}(pp_2), f' \mapsto \mathsf{FS}(pp_4)|\!\}\\ & pp_4 \mapsto \{\!|c'; f \mapsto \mathsf{FS}(pp_1), f' \mapsto \mathsf{FS}(pp_3)|\!\}\end{aligned}$$

Assume we want to lift the allocation site pp_1, the computation of the abstract filter gives: $\hat{k} = pp_1 \mapsto 1, pp_2 \mapsto 1, pp_3 \mapsto 0, pp_4 \mapsto 0$. The result of the lifting is then the following:

$$\begin{aligned}\mathsf{hlift}(\hat{h}; \hat{k}) \quad = \quad & pp_1 \mapsto \bot, pp_2 \mapsto \bot,\\ & pp_3 \mapsto \{\!|c'; f \mapsto \mathsf{NFS}(pp_2), f' \mapsto \mathsf{FS}(pp_4)|\!\}\\ & pp_4 \mapsto \{\!|c'; f \mapsto \mathsf{NFS}(pp_1), f' \mapsto \mathsf{FS}(pp_3)|\!\}\end{aligned}$$

4.4.4 Abstracting Local Reduction

Accessing the Abstract Heaps We observe that in the concrete semantics one often needs to read a location stored in a register and then access the contents of that location on the heap. In the abstract semantics we rely on a similar mechanism, adapted to read from the correct abstract heap. The fact $\mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \hat{\lambda}; \hat{b})$ states that if $\hat{v}^*$ is an over-approximation of the content of the registers and $\hat{h}$ is an abstract flow-sensitive heap, then $\hat{\lambda}$ is an abstract location over-approximated by $\hat{v}_i$ and $\hat{b}$ is an abstract block over-approximating the memory block that register i is pointing to. Formally, this fact can be proved by the two Horn clauses below, discriminating on the flow-sensitivity of $\hat{\lambda}$:

$$\begin{aligned}\mathsf{FS}(\lambda) \sqsubseteq \hat{v}_i \wedge \hat{h}(\lambda) = \hat{b} \quad & \Longrightarrow \quad \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{FS}(\lambda); \hat{b})\\ \mathsf{NFS}(\lambda) \sqsubseteq \hat{v}_i \wedge \mathsf{H}(\lambda, \hat{b}) \quad & \Longrightarrow \quad \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{NFS}(\lambda); \hat{b})\end{aligned}$$

Evaluation of Right-Hand Sides The abstract semantics needs to be able to over-approximate the evaluation of right-hand sides. This is done via a translation $\langle\!\langle rhs \rangle\!\rangle_{pp}$ generating a set of Horn clauses, which over-approximate the value of rhs at program point pp. For example, the following translation rule generates one Horn clause which approximates the content of the register r_i at pp, based on the information stored in the corresponding local state abstraction:

$$\langle\!\langle r_i \rangle\!\rangle_{pp} = \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \_; \_) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{v}_i)\}$$

Standard Statements The abstract semantics defines, for each possible form of statement st, a translation $(\!| st |\!)_{pp}$ into a set of Horn clauses which over-approximate the semantics of st at program point pp. We start by discussing the top part of Table 4.15, presenting the abstract semantics of some statements considered in § 2.4. We focus in particular on the main additions needed to generalize their abstraction to implement a flow-sensitive heap analysis:

- $(\!|\texttt{new}\ r_d\ c'|\!)_{c,m,pc} =$
  $\{\mathsf{LState}_{c,m,pc}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{Reach}(\mathsf{FS}(c,m,pc);\hat{h};\hat{k}')$
  $\implies \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{c,m,pc+1}(\_;\mathsf{lift}(\hat{v}^*;\hat{k}')[d\mapsto \mathsf{FS}(c,m,pc)];\mathsf{hlift}(\hat{h};\hat{k}')[c,m,pc \mapsto$
  $\{\!|c';(f\mapsto \hat{\mathbf{0}}_\tau)^*|\!\}];\hat{k}\,\hat{\sqcup}\,\hat{k}')\}$

- $(\!|\texttt{move}\ r_o.f\ rhs|\!)_{c,m,pc} =$
  $\langle\!\langle rhs\rangle\!\rangle_{c,m,pc} \cup \{\mathsf{RHS}_{c,m,pc}(\hat{v}'') \wedge \mathsf{LState}_{c,m,pc}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*;\hat{h};\mathsf{FS}(\lambda);\{\!|c';(f'\mapsto$
  $\hat{u}')^*, f\mapsto \hat{v}'|\!\}) \implies$
  $\mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*;\hat{h}[\lambda\mapsto \{\!|c';(f'\mapsto\hat{u}')^*, f\mapsto\hat{v}''|\!\};\hat{k})\} \cup$
  $\{\mathsf{RHS}_{c,m,pc}(\hat{v}'') \wedge \mathsf{LState}_{c,m,pc}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*;\hat{h};\mathsf{NFS}(\lambda);\{\!|c';(f'\mapsto\hat{u}')^*, f\mapsto$
  $\hat{v}'|\!\}) \wedge \mathsf{Reach}(\hat{v}'';\hat{h};\hat{k}') \implies$
  $\mathsf{H}(\lambda,\{\!|c';(f'\mapsto\hat{u}')^*, f\mapsto\hat{v}'')|\!\}) \wedge \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge$
  $\mathsf{LState}_{c,m,pc+1}(\_;\mathsf{lift}(\hat{v}^*;\hat{k}');\mathsf{hlift}(\hat{h};\hat{k}');\hat{k}\,\hat{\sqcup}\,\hat{k}')\}$

- $(\!|\texttt{return}|\!)_{c,m,pc} = \{\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t,\hat{v}^*_{call});\hat{v}^*;\hat{h};\hat{k}) \implies \mathsf{Res}_{c,m}((\hat{\lambda}_t,\hat{v}^*_{call});\hat{v}_{\mathsf{res}};\hat{h};\hat{k})\}$

- $(\!|\texttt{invoke}\ r_o\ m'\ (r_{i_j})^{j\leq n}|\!)_{c,m,pc} =$
  $\{\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t,\_);\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*;\hat{h};\_;\{\!|c';(f\mapsto\hat{u})^*|\!\}) \wedge c'\leq c'' \implies$
  $\mathsf{LState}_{c'',m',0}((\hat{\lambda}_t,(\hat{v}_{i_j})^{j\leq n});(\hat{\mathbf{0}}_k)^{k\leq loc},(\hat{v}_{i_j})^{j\leq n};\hat{h};0^*) \mid c'' \in \widehat{lookup}(m') \wedge sign(c'',m') =$
  $(\tau_j)^{j\leq n} \xrightarrow{loc} \tau\} \cup$ **(1)**
  $\{\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t,\_);\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*;\hat{h};\_;\{\!|c';(f\mapsto\hat{u})^*|\!\}) \wedge c'\leq c'' \wedge$
  $\mathsf{Res}_{c'',m'}((\hat{\lambda}'_t,\hat{w}^*);\hat{v}'_{\mathsf{res}};\hat{h}_{\mathsf{res}};\hat{k}_{\mathsf{res}})$
  $\wedge\hat{\lambda}_t = \hat{\lambda}'_t \wedge \left(\bigwedge_{j\leq n}\hat{v}_{i_j}\sqcap\hat{w}_j \not\sqsubseteq \bot\right) \implies \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t,\_);\mathsf{lift}(\hat{v}^*;\hat{k}_{\mathsf{res}})[\mathsf{res}\mapsto\hat{v}'_{\mathsf{res}}];\hat{h}_{\mathsf{res}};\hat{k}\,\hat{\sqcup}$
  $\hat{k}_{\mathsf{res}}) \mid c''\in\widehat{lookup}(m')\} \cup$ **(2)**
  $\{\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t,\_);\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*;\hat{h};\_;\{\!|c';(f\mapsto\hat{u})^*|\!\}) \wedge c'\leq c'' \wedge$
  $\mathsf{Uncaught}_{c'',m'}((\hat{\lambda}'_t,\hat{w}^*));\hat{v}'_{\mathsf{excpt}};\hat{h}_{\mathsf{res}};\hat{k}_{\mathsf{res}})$
  $\wedge\ \hat{\lambda}_t = \hat{\lambda}'_t \wedge \left(\bigwedge_{j\leq n}\hat{v}_{i_j}\sqcap\hat{w}_j \not\sqsubseteq \bot\right) \implies \mathsf{AState}_{c,m,pc}((\hat{\lambda}_t,\_);\mathsf{lift}(\hat{v}^*;\hat{k}_{\mathsf{res}})[\mathsf{excpt}\mapsto$
  $\hat{v}'_{\mathsf{excpt}}];\hat{h}_{\mathsf{res}};\hat{k}\,\hat{\sqcup}\,\hat{k}_{\mathsf{res}}) \mid c''\in\widehat{lookup}(m')\}$ **(3)**

- $(\!|\texttt{throw}\ r_i|\!)_{c,m,pc} = \{\mathsf{LState}_{c,m,pc}(\_;\hat{v}^*;\hat{h};\hat{k}) \implies \mathsf{AState}_{c,m,pc}(\_;\hat{v}^*[\mathsf{excpt}\mapsto\hat{v}_i];\hat{h};\hat{k})\}$

- $(\!|\texttt{start-thread}\ r_i|\!)_{c,m,pc} =$
  $\{\mathsf{LState}_{c,m,pc}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{NFS}(\lambda);\{\!|c';(f\mapsto\hat{u})^*|\!\}) \wedge c'\leq \mathsf{Thread}$
  $\implies \mathsf{T}(\lambda,\{\!|c';(f\mapsto\hat{u})^*|\!\}) \wedge \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*;\hat{h};\hat{k})\} \cup$
  $\{\mathsf{LState}_{c,m,pc}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{FS}(\lambda);\{\!|c';(f\mapsto\hat{u})^*|\!\}) \wedge c'\leq \mathsf{Thread} \wedge$
  $\mathsf{Reach}(\mathsf{FS}(\lambda);\hat{h};\hat{k}')$
  $\implies \mathsf{T}(\lambda,\{\!|c';(f\mapsto\hat{u})^*|\!\}) \wedge \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{c,m,pc+1}(\_;\mathsf{lift}(\hat{v}^*;\hat{k}');\mathsf{hlift}(\hat{h};\hat{k}');\hat{k}\,\hat{\sqcup}\,\hat{k}')\}$

- $(\!|\texttt{join}\ r_i|\!)_{c,m,pc} =$
  $\{\mathsf{LState}_{c,m,pc}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{H}(\lambda_t,\{\!|c';(f\mapsto\hat{u})^*,\mathsf{inte}\mapsto\hat{v}'|\!\}) \wedge \widehat{\texttt{false}}\sqsubseteq\hat{v}' \implies$
  $\mathsf{LState}_{c,m,pc+1}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k})\} \cup$
  $\{\mathsf{LState}_{c,m,pc}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{H}(\lambda_t,\{\!|c';(f\mapsto\hat{u})^*,\mathsf{inte}\mapsto\hat{v}'|\!\}) \wedge \widehat{\texttt{true}}\sqsubseteq\hat{v}' \implies$
  $\mathsf{H}(c,m,pc;\{\!|\mathsf{IntExcpt};|\!\}) \wedge \mathsf{AState}_{c,m,pc}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*[\mathsf{excpt}\mapsto\mathsf{NFS}(c,m,pc)];\hat{h};\hat{k}) \wedge$
  $\mathsf{H}(\lambda_t,\{\!|c';(f\mapsto\hat{u})^*,\mathsf{inte}\mapsto\widehat{\texttt{false}}|\!\})\}$

Table 4.15: Abstract Semantics of Statements - Excerpt

- $(\!|\texttt{new}\ r_d\ c'|\!)_{pp}$: When allocating a new object at pp, the abstraction of the object that was the most-recently allocated one before the new allocation, if any, must be downgraded to a flow-insensitive analysis. Therefore, we lift the allocation site pp by computing an abstract filter $\hat{k}'$ via the Reach predicate and using it to perform the lifting as described in Section 4.4.3. We then put in the resulting abstract flow-sensitive heap a new abstract object $\{\!|c'; (f \mapsto \hat{\mathbf{0}}_\tau)^*|\!\}$ initialized to default values ($\hat{\mathbf{0}}_\tau$ represents the abstraction of the default value used to populate fields of type τ). The abstraction of the register r_d is set to the abstract flow-sensitive location $\mathsf{FS}(pp)$ to enable a flow-sensitive analysis of the new most-recently-allocated object;

- $(\!|\texttt{move}\ r_o.f\ rhs|\!)_{pp}$: We first use $\langle\!\langle rhs\rangle\!\rangle_{pp}$ to generate the Horn clauses over-approximating the value of rhs at program point pp. Assume then we have the over-approximation $\hat{v}''$ in a RHS fact. We have two possibilities, based on the abstract value $\hat{v}_o$ over-approximating the content of the register r_o. If GetBlk_o returns an abstract flow-sensitive location $\mathsf{FS}(\lambda)$, then we perform a strong update on the corresponding element of the abstract flow-sensitive heap. If GetBlk_o returns an abstract flow-insensitive location $\mathsf{NFS}(\lambda)$, we use λ to get an abstract heap fact $\mathsf{H}(\lambda, \{\!|c'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'|\!\})$ and we update the field f of this object in a new heap fact: this implements a weak update, since the old fact is still valid. The abstract value $\hat{v}''$ moved to the flow-insensitive heap fact may contain abstract flow-sensitive locations, which must be downgraded by lifting $\hat{v}''$ when propagating the local state abstraction to the next program point;

- $(\!|\texttt{return}|\!)_{pp}$: The callee generates a return fact Res containing the calling context $(\hat{\lambda}_t, \hat{v}^*_{call})$, the abstract value $\hat{v}_{\mathsf{res}}$ over-approximating the return value, its abstract flow-sensitive heap $\hat{h}$ and its abstract filter $\hat{k}$ recording which allocation sites were lifted during its computation. All this information is propagated to the analysis of the caller, as we explain in the next item;

- $(\!|\texttt{invoke}\ r_o\ m'\ (r_{i_j})^{j\leq n}|\!)_{pp}$: We statically know the name m' of the invoked method, but not the class of the receiver object in the register r_o. In part (**1**) we over-approximate dynamic dispatching as follows: we collect all the abstract objects accessible via the abstraction $\hat{v}_o$ of the content of the register r_o, but we only consider as possible receivers the ones whose type is a subtype of a class $c'' \in \widehat{lookup}(m')$, where $\widehat{lookup}(m')$ just returns the set of classes which define or inherit a method named m'. For all of them, we introduce an abstract local state fact LState over-approximating the local state of the invoked method, instantiating it with the calling context, the abstract flow-sensitive heap of the caller and an empty abstract filter.

  Part (**2**) handles the propagation of the abstraction of the return value from the callee to the caller. This is done by using the Res fact generated by the `return` statement of the callee: the caller matches appropriate callees by checking the context of the Res fact. Specifically, the caller checks that: (i) its own abstraction $\hat{\lambda}_t$ matches the abstraction $\hat{\lambda}'_t$ in the context of the callee, and (ii) that the meet of its arguments $\hat{v}_{i_j}$ and the context arguments $\hat{w}_j$ is not $\perp$. This prevents a callee from returning to a caller that could not have invoked it, in case (i) because different threads are executing caller and callee, and in case (ii) because the over-approximation of the arguments used by the caller and the over-approximation of the arguments supplied to the callee are disjoint. We then instantiate the abstract local state of the next program point by inheriting the abstract flow-sensitive heap of the callee $\hat{h}_{\mathsf{res}}$, lifting the abstraction of the caller registers, joining the caller abstract filter $\hat{k}$ with the callee abstract filter $\hat{k}_{\mathsf{res}}$, and storing the abstraction of the returned value $\hat{v}'_{\mathsf{res}}$ in the abstraction of the return register.

Finally, part (**3**) of the rule is used to handle the propagation of uncaught exceptions from the callee to the caller. It uses an abstract uncaught exception fact $\mathsf{Uncaught}$, generated by the exception rules explained below: it tries to throw back the exceptions to an appropriate caller, by matching the context of the $\mathsf{Uncaught}$ fact with the abstract local state of the caller.

Exceptions and Threads The bottom part of Table 4.15 presents the abstract semantics of some selected new statements of the concrete semantics:

- $(\!|\texttt{throw } r_i|\!)_{pp}$: We generate an abstract *abnormal* local state fact AState from the abstract local state throwing the exception, and we set the abstraction of the special exception register accordingly;

- $(\!|\texttt{start-thread } r_i|\!)_{pp}$: We create an abstract pending thread fact T, tracking that a new thread was started. The actual instantiation of the abstract thread object is done by the abstract counterpart of the global reduction rules, which we discuss later. Observe that, if the abstract location pointing to the abstract thread object has the form $\mathsf{FS}(\lambda)$, then λ is lifted, since the parent thread can access the state of the new thread, but the two threads are concurrently executed;

- $(\!|\texttt{join } r_i|\!)_{pp}$: We just check whether the inte field of the abstract object over-approximating the running thread or activity is over-approximating $\widehat{\texttt{true}}$, in which case an abstract abnormal local state throwing an $\mathsf{IntExcpt}$ exception is generated, or $\widehat{\texttt{false}}$, in which case the abstract local state is propagated to the next program point.

Example We show in Table 4.16 a (simplified) bytecode program corresponding to the code snippet in Table 4.1. A few comments about the bytecode: the activity constructor `<init>` is explicitly defined; by convention, the first register after the local registers of a method is used to store a pointer to the activity object and the register `ret` is used to store the result of the last invoked method.

We assume that the class `Leaky` extends `Activity` and implements at least the methods `send` and `getDeviceId`, whose code is not shown here. We also use line numbers to refer to program points, which makes the notation lighter. Notice that there are only two allocation points, lines 7 and 9, therefore the abstract flow-sensitive heap will contain only two entries and have the form $7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2$.

We selected three bytecode instructions and we give for each of them the Horn clauses generated by our analysis. We briefly comment on the clauses: the `new` instruction at line 7 computes all the abstract flow-sensitive locations reachable from $\mathsf{FS}(7)$ with the predicate Reach: bb'_1 (resp. bb'_2) is set to 1 iff the location 7 (resp. 9) needs to be lifted. These abstract flow-sensitive locations are then lifted, if needed, using:

$$\mathsf{LiftHeap}(7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2),$$

and the abstract flow-sensitive heap is updated by putting a fresh `Storage` object in 7 and by lifting 9, if needed:

$$7 \mapsto \{\!|\texttt{Storage}; \texttt{s} \mapsto \text{""}|\!\}, 9 \mapsto \mathsf{hlift}(\hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2).$$

The `invoke` instruction at line 18 has two clauses: the first clause retrieves the callee's class c' and performs an abstract virtual method dispatch (here there is only one class implementing `getDeviceId`, hence this step is trivial); the second clause gets the result from the called method and returns it to the caller, checking that the caller's abstract thread pointer $\hat{\lambda}_t$ and supplied argument $\hat{v}$ match the callee's context $(\hat{\lambda}'_t, \hat{v}')$ with the constraint $\hat{\lambda}_t = \hat{\lambda}'_t \wedge \hat{v} \sqcap \hat{v}' \not\sqsubseteq \bot$. We removed the exception handling clauses, as they are not relevant here.

Finally, the `move` instruction at line 20 is abstracted by four Horn clauses: the first one evaluates the right-hand side of the `move`; the two subsequent clauses execute the move in case the left-hand side is the field `s` of, respectively, the abstract flow-sensitive location 7 or 9; finally, the last clause is used if the left-hand side is the field `s` of an abstract flow-insensitive location, in which case a new abstract flow-insensitive heap entry is created.

4.4.5 Abstracting Global Reduction

The abstract counterpart of the global reduction rules is a set of Horn clauses over-approximating system events and the Android activity lifecycle. We extended the original rules of HornDroid presented in Table 2.5 and Table 2.6 with some new rules needed to support our richer concrete semantics including threads and exceptions. Table 4.17 shows two of these rules to exemplify, the other rules are in § 5.2. Rule *Tstart* over-approximates the spawning of new threads by generating an abstract local state executing the run method of the corresponding thread object. Rule *AbState* abstracts the mechanism by which a method recovers from an exception: part (**A**) turns an abstract abnormal state into an abstract local state if the abstraction of the exception register contains the abstract location of an object of class c extending the Throwable interface and if there exists an appropriate entry for exception handling in the exception table; part (**B**) is triggered if no such entry exists, and generates an abstract uncaught exception fact,

Bytecode Example:

```
1 .class public Leaky
2 .super Activity
3 .field st:Storage
4 .field st2:Storage
5 .method constructor
         <init>()
6 .1 local register
7   new r0 Storage
8   move r1.st r0
9   new r0 Storage
10  move r1.st2 r0
11 .end method
```

```
12 .method onRestart()
13 .1 local register
14   move r1.st2 r1.st
15 .end method
16 .method onResume()
17 .1 local register
18   invoke r1
         getDeviceId()
19   move r0 r1.st2
20   move r0.s ret
21 .end method
```

```
22 .method onPause()
23 .2 local registers
24   move r0 r2.st
25   move r1 r0.s
26   move r0 "http://myapp
         .com/"
27   invoke r2
         send() r1 r0
28 .end method
```

Generated Horn Clauses for Line 7:

- $\mathsf{LState}_7(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge$
  $\mathsf{Reach}(\mathsf{FS}(7); 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2) \implies$
  $\mathsf{LiftHeap}(7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2) \wedge$
  $\mathsf{LState}_8(\_; r_0 \mapsto \mathsf{FS}(7), r_1 \mapsto \mathsf{lift}(\hat{u}; 7 \mapsto bb'_1, 9 \mapsto bb'_2);$
  $7 \mapsto \{\!|\mathtt{Storage}; \mathtt{s} \mapsto \text{""}|\!\}, 9 \mapsto \mathsf{hlift}(\hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2); 7 \mapsto bb_1 \hat{\sqcup}\, bb'_1, 9 \mapsto bb_2 \hat{\sqcup}\, bb'_2)$

Generated Horn Clauses for Line 18:

- $\mathsf{LState}_{18}((\hat{\lambda}_t, \_); r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge$
  $\mathsf{GetBlk}_1(r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; \_; \{\!|c'; \_|\!\}) \wedge c' \leq \mathtt{Leaky} \implies$
  $\mathsf{LState}_0((\hat{\lambda}_t, \hat{v}); r_0 \mapsto \hat{v}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto 0, 9 \mapsto 0)$
- $\mathsf{LState}_{18}((\hat{\lambda}_t, \_); r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge$
  $\mathsf{GetBlk}_1(r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; \_; \{\!|c'; \_|\!\}) \wedge c' \leq \mathtt{Leaky} \wedge$
  $\mathsf{Res}_{\mathtt{getDeviceId}}((\hat{\lambda}'_t, \hat{v}'); \hat{u}'_{\mathsf{res}}; 7 \mapsto \hat{l}'_1, 9 \mapsto \hat{l}'_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2) \wedge \hat{\lambda}_t = \hat{\lambda}'_t \wedge \hat{v} \sqcap \hat{v}' \not\sqsubseteq \bot \implies$
  $\mathsf{LState}_{19}((\hat{\lambda}_t, \_); r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{u}'_{\mathsf{res}}; 7 \mapsto \hat{l}'_1, 9 \mapsto \hat{l}'_2; 7 \mapsto bb_1 \hat{\sqcup}\, bb'_1, 9 \mapsto bb_2 \hat{\sqcup}\, bb'_2)$

Generated Horn Clauses for Line 20:

- $\mathsf{LState}_{20}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \implies \mathsf{RHS}_{20}(\hat{w})$
- $\mathsf{LState}_{20}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge$
  $\mathsf{RHS}_{20}(\hat{u}') \wedge \mathsf{GetBlk}_0(r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; \mathsf{FS}(7); \{\!|\mathtt{Storage}; \mathtt{s} \mapsto \hat{v}'|\!\}) \implies$
  $\mathsf{LState}_{21}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \{\!|\mathtt{Storage}; \mathtt{s} \mapsto \hat{u}'|\!\}, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2)$
- $\mathsf{LState}_{20}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge$
  $\mathsf{RHS}_{20}(\hat{u}') \wedge \mathsf{GetBlk}_0(r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; \mathsf{FS}(9); \{\!|\mathtt{Storage}; \mathtt{s} \mapsto \hat{v}'|\!\}) \implies$
  $\mathsf{LState}_{21}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \{\!|\mathtt{Storage}; \mathtt{s} \mapsto \hat{u}'|\!\}; 7 \mapsto bb_1, 9 \mapsto bb_2)$
- $\mathsf{LState}_{20}(\_; r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge \mathsf{RHS}_{20}(\hat{u}') \wedge$
  $\mathsf{GetBlk}_0(r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; \mathsf{NFS}(\mathsf{pp}); \{\!|\mathtt{Storage}; \mathtt{s} \mapsto \hat{v}'|\!\}) \wedge$
  $\mathsf{Reach}(\hat{u}'; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2) \implies$
  $\mathsf{LiftHeap}(7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2) \wedge \mathsf{H}(\mathsf{pp}, \{\!|\mathtt{Storage}; \mathtt{s} \mapsto \hat{u}'|\!\}) \wedge$
  $\mathsf{LState}_{21}(\_; r_0 \mapsto \mathsf{lift}(\hat{u}; 7 \mapsto bb'_1, 9 \mapsto bb'_2),$
  $r_1 \mapsto \mathsf{lift}(\hat{v}; 7 \mapsto bb'_1, 9 \mapsto bb'_2), \mathtt{ret} \mapsto \mathsf{lift}(\hat{w}; 7 \mapsto bb'_1, 9 \mapsto bb'_2);$
  $7 \mapsto \mathsf{hlift}(\hat{l}_1; 7 \mapsto bb'_1, 9 \mapsto bb'_2), 9 \mapsto \mathsf{hlift}(\hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2);$
  $7 \mapsto bb_1 \hat{\sqcup}\, bb'_1, 9 \mapsto bb_2 \hat{\sqcup}\, bb'_2)$

Table 4.16: Example of Dalvik Bytecode and Excerpt of the Corresponding Horn Clauses

$$
\begin{array}{lcl}
\mathit{Tstart} & = & \{\mathsf{T}(\lambda, \{\!|c; (f \mapsto \_)^* |\!\}) \wedge c \leq \mathsf{c}' \wedge c \leq \mathsf{Thread} \implies \\
& & \mathsf{LState}_{\mathsf{c}',\mathsf{run},0}((\mathsf{NFS}(\lambda), \mathsf{NFS}(\lambda)); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(\lambda); (\bot)^*; 0^*) \mid \mathsf{c}' \in \widehat{lookup}(\mathsf{run}) \\
& & \wedge sign(\mathsf{c}', \mathsf{run}) = \mathsf{Thread} \xrightarrow{loc} \mathsf{Void}\} \\
\mathit{AbState} & = & \{\mathsf{AState}_{\mathsf{c},\mathsf{m},\mathsf{pc}}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}^*; \hat{h}; \_; \{\!|\mathsf{c}'; \_|\!\}) \wedge \mathsf{c}' \leq \mathsf{Throwable} \implies \\
& & \mathsf{LState}_{\mathsf{c},\mathsf{m},\mathsf{pc}'}(\_; \hat{v}^*; \hat{h}; \hat{k}) \mid \mathsf{ExcptTable}(\mathsf{c}, \mathsf{m}, \mathsf{pc}, \mathsf{c}') = \mathsf{pc}'\} \cup \qquad (\mathbf{A}) \\
& & \{\mathsf{AState}_{\mathsf{c},\mathsf{m},\mathsf{pc}}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}^*; \hat{h}; \_; \{\!|\mathsf{c}'; \_|\!\}) \wedge \mathsf{c}' \leq \mathsf{Throwable} \implies \\
& & \mathsf{Uncaught}_{\mathsf{c},\mathsf{m}}(\_; \hat{v}_{\mathsf{excpt}}; \hat{h}; \hat{k}) \mid \mathsf{ExcptTable}(\mathsf{c}, \mathsf{m}, \mathsf{pc}, \mathsf{c}') = \bot\} \qquad (\mathbf{B})
\end{array}
$$

Table 4.17: Global Rules of the Abstract Semantics - Excerpt

which is then used in the abstract semantics of the method invocation performed by the caller.

Let $\mathcal{R}$ denote the set of all the Horn clauses defining the auxiliary facts, like GetBlk_i, plus the Horn clauses abstracting system events and the activity lifecycle. We define the translation of a program P into Horn clauses, noted as $(\!|P|\!)$, by adding to $\mathcal{R}$ the translation of the individual statements of P.

4.4.6 Formal Results

The soundness of the analysis is proved by using *representation functions* [NNH99]: we define a function β_{Cnf} mapping each concrete configuration Ψ to a set of abstract configurations over-approximating it. We then define a partial order $<:$ between abstract configurations, where $\Delta <: \Delta'$ should be interpreted as: Δ is no coarser than Δ'. The soundness theorem can be stated as follows; its proof is given in § 5.3.

Theorem 3 (Global Preservation)**.** *If $\Psi \Rightarrow^* \Psi'$ under a given program P, then for any $\Delta_1 \in \beta_{\mathit{Cnf}}(\Psi)$ and $\Delta_2 :> \Delta_1$ there exist $\Delta'_1 \in \beta_{\mathit{Cnf}}(\Psi')$ and $\Delta'_2 :> \Delta'_1$ s.t. $(\!|P|\!) \cup \Delta_2 \vdash \Delta'_2$.*

We now discuss how a sound static taint analysis can be implemented on top of our formal result. First, we extend the syntax of concrete values as follows:

$$
\begin{array}{llcl}
\text{Taint} & t & ::= & \mathsf{public} \mid \mathsf{secret} \\
\text{Values} & u, v & ::= & prim^t \mid \ell
\end{array}
$$

The set of taints is a two-valued lattice, and we use $\sqsubseteq^{\mathsf{t}}$ and $\sqcup^{\mathsf{t}}$ to denote respectively the standard ordering on taints (where $\mathsf{public} \sqsubseteq^{\mathsf{t}} \mathsf{secret}$) and their join. When performing unary and binary operations, taints are propagated by having the taint of the result be the join of the taints of the arguments.

We then define the taint extraction function taint_Ψ which satisfies the following relations:

$$\mathsf{taint}_\Psi(v) = \begin{cases} \sqcup^{\mathsf{t}}{}_i\ \mathsf{taint}_\Psi(v_i) & \text{if } v = \ell \wedge H(\ell) = \{\!|c; (f_i \mapsto v_i)^*|\!\} \\ \sqcup^{\mathsf{t}}{}_i\ \mathsf{taint}_\Psi(v_i) & \text{if } v = \ell \wedge H(\ell) = \tau[v^*] \\ \sqcup^{\mathsf{t}}{}_i\ \mathsf{taint}_\Psi(v_i) & \text{if } v = \ell \wedge H(\ell) = \{\!|@c; (k_i \mapsto v_i)^*|\!\} \\ t & \text{if } v = prim^t \end{cases}$$

Informally, given a value v, it extracts its taint by doing a recursive computation: if v is a primitive value this is straightforward; if v is a pointer it recursively computes the join of all the taint accessible from v in the heap of Ψ.

We describe in Table 4.18 the abstract counter-part of taint_Ψ: intuitively $\mathsf{Taint}(\hat{v}, \hat{h}, \hat{t})$ holds when $\hat{v}$ has taint $\hat{t}$ in the abstract local heap $\hat{h}$. The rules defining Taint are similar to the rules defining Reach, since both predicates need to perform a fix-point computation in the abstract heap.

$$\mathsf{Taint}(\widehat{prim^t}, \hat{h}, t) \qquad\qquad \mathsf{Taint}(\hat{u}, \hat{h}, \hat{t}) \wedge \hat{u} \sqsubseteq \hat{v} \implies \mathsf{Taint}(\hat{v}, \hat{h}, \hat{t})$$

$$\mathsf{Taint}(\hat{v}, \hat{h}, \hat{t}) \wedge \mathsf{Taint}(\hat{v}, \hat{h}, \hat{t}') \implies \mathsf{Taint}(\hat{v}, \hat{h}, \hat{t} \sqcup^{\mathsf{t}} \hat{t}')$$

$$\mathsf{GetBlk}_0(\hat{u}; \hat{h}; \_; \hat{b}) \wedge \left\{ \begin{array}{r} \hat{b} = \{\!|c; \_, f \mapsto \hat{v}|\!\} \\ \hat{b} = \tau[\hat{v}] \\ \hat{b} = \{\!|@c; \hat{v}|\!\} \end{array} \right\} \wedge \mathsf{Taint}(\hat{v}, \hat{h}, \hat{t}) \implies \mathsf{Taint}(\hat{u}, \hat{h}, \hat{t})$$

Table 4.18: Horn Clauses Rules used to Derive $\mathsf{Taint}(\hat{v}, \hat{h}, \hat{t})$.

Finally, we assume two sets *Sinks* and *Sources*, where *Sinks* (resp. *Sources*) contains a pair (c, m) if and only if a method m of a class c is a sink (resp. a source). We assume that when a source returns a value, it always has the secret taint.

Definition 12. A program P *leaks* starting from a configuration Ψ if there exists $(c, m) \in \mathit{Sinks}$ such that $\Psi \Rightarrow^* \Omega \cdot \Xi \cdot H \cdot S$ and there exists $\langle \ell, s, \pi, \gamma, \alpha \rangle \in \Omega$ or $\langle\!\langle \ell, \ell', \pi, \gamma, \alpha \rangle\!\rangle \in \Xi$ such that $\alpha = \langle c, m, 0 \cdot u^* \cdot st^* \cdot R \rangle :: \alpha'$, $R(r_k) = v$ and $\mathsf{taint}_\Psi(v) = \mathsf{secret}$ for some r_k and v.

We then state the soundness of our taint tracking analysis in the following lemma: its proof can be found in Section 5.3.10.

Lemma 10. *If for all sinks* $(c, m) \in \mathit{Sinks}$, $\Delta \in \beta_{\mathit{Cnf}}(\Psi)$*:*

$$(\!|P|\!) \cup \Delta \vdash \mathsf{LState}_{c,m,0}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{Taint}(\hat{v}_i, \hat{h}, \mathsf{secret})$$

is unsatisfiable for each i, then P does not leak from Ψ.

4.5 Experiments

We implemented a prototype of our flow-sensitive analysis as an extension of HornDroid presented in Chapter 2. Our tool encodes the application to analyse as a set of Horn clauses, as we detailed in the previous section, and then uses the SMT solver *z3* [dMB08b] to statically detect information leaks. More specifically, the tool automatically generates a set of queries for the analysed application based on a public database of Android sources and sinks [RAB14]; if no query is satisfiable according to *z3*, no information leak may occur by the soundness results of our analysis.

4.5.1 Testing on DroidBench

We tested our flow-sensitive extension of HornDroid (called fsHornDroid) against DroidBench [ARF+14], a common benchmark of 115 small applications proposed by the research community to test information flow analysers for Android[3]. In our experiments we compared with the most popular and advanced static taint trackers for Android applications: FlowDroid [ARF+14], AmanDroid [WROR14], DroidSafe [GKP+15] and the original version of HornDroid described in Chapter 2. For all the tools, we computed standard validity measures (sensitivity for soundness and specificity for precision) and we tracked the analysis times on the 115 applications included in DroidBench: the experimental results are summarised in Table 4.19.

Validity Measures on DroidBench:

	FlowDroid	AmanDroid	DroidSafe	HornDroid	fsHornDroid
Sensitivity	0.67	0.74	0.92	1	1
Specificity	0.58	0.74	0.47	0.68	0.79
F-Measure	0.62	0.74	0.62	0.81	0.88

$Sensitivity = tp/(tp + fn) \sim$ Soundness
$Specificity = tn/(tn + fp) \sim$ Precision
$F\text{-}Measure = 2 * (sens * spec)/(sens + spec) \sim$ Aggregate

Analysis Times on DroidBench:

	FlowDroid	AmanDroid	DroidSafe	HornDroid	fsHornDroid
Average	22s	11s	2m92s	1s	14s
1st Quartile	13s	9s	2m38s	1s	1s
2nd Quartile	14s	10s	3m1s	1s	2s
3rd Quartile	15s	11s	3m26s	1s	5s

Table 4.19: Validity Measures and Analysis Times on DroidBench

[3]We removed from DroidBench 4 applications testing implicit information flows, since none of the available tools aims at supporting them.

Like the original version of HornDroid, fsHornDroid detects all the information leaks in DroidBench, since its sensitivity is 1. However, fsHornDroid turns out to be the most precise static analysis tool to date, with a value of specificity which is strictly higher than the one of all its competitors. In particular, fsHornDroid produces only 4 false positives on DroidBench: a leak inside an exception that is never thrown; a leak inside an unregistered callback which cannot be triggered; a leak inside an undeclared activity which cannot be started; and a leak of a public element of a list which also contains a confidential element. The last two cases should be easy to fix: the former by parsing the application manifest and the latter by implementing field-sensitivity for lists.

We also evaluated the analysis times of the applications in DroidBench for the different tools. In terms of performances, the original version of HornDroid is better than fsHornDroid as expected. However, the performances of fsHornDroid are satisfying: the median analysis time does not change too much with respect to HornDroid, which is the fastest tool, while the average analysis time is comparable with other flow-sensitive analysers like FlowDroid and AmanDroid.

4.5.2 Testing on Real Applications

In order to test the scalability of fsHornDroid, we picked the top 4 applications from 16 categories in a publicly available snapshot of the Google Play market [Theb]. For each application, we run fsHornDroid setting a timeout of 3 hours for finding the first information leak. In the end, we managed to get the analysis results within the timeout for 62 applications, whose average and median sizes were 7.4 Mb and 5 Mb respectively. The tool reported 47 applications as leaky and found no direct information leaks for 15 applications. Unfortunately, the absence of a ground truth makes it hard to evaluate the validity of the reported leaks. To preliminarily assess the improvement in precision due to flow-sensitivity, however, we sampled 3 of the potentially leaky applications and we checked all their possible information leaks. On these applications, fsHornDroid eliminated 17 false positives with respect to HornDroid, which amount to 18% of all the checked flows.

In terms of performances, fsHornDroid spent 17 minutes on average to perform the analysis, with a median analysis time of 2 minutes on an Intel Xeon E5-4650L 2.60 GHz. The updated experimental evaluation is available online, along with the sources of the tool [fsh]. Our results demonstrate that fsHornDroid scales to real applications, despite the increased performance overhead with respect to the original HornDroid.

4.5.3 Limitations

Our implementation of fsHornDroid does not aim at solving a few important limitations of HornDroid. First, a comprehensive implementation of *analysis stubs* for unknown methods is missing: this issue was thoroughly discussed by the authors of DroidSafe [GKP+15] and we think their research may be beneficial to improve on this. Moreover, the analysis does not capture *implicit* information flows, but only direct information leaks, and it does not cover native code, but only Dalvik bytecode. Finally, the analysis has no way of being less conservative on *intended* information flows: implementing declassification mechanisms would be important to analyse real applications without raising a high number of false alarms.

4.6 Related Work

There are several static information flow analysers for Android applications (see, e.g., [YY12, ZO12, MS12, GCEC12, KYY$^+$12, ARF$^+$14, WROR14, GKP$^+$15, CGM16]). We thoroughly compared with the current state of the art in the rest of the chapter, so we focus here on other related works.

Sound Analysis of Android Applications The first paper proposing a formally sound static analysis of Android applications is a seminal work by Chaudhuri [Cha09]. The paper presented a type-based analysis to reason on the data-flow security properties of Android applications modeled in an idealised calculus. A variant of the analysis was implemented in a prototype tool, SCanDroid [FCF09]. Unfortunately, SCanDroid is in an early prototype phase and it cannot analyse the applications in DroidBench [ARF$^+$14].

Sound type systems for Android applications have also been proposed in [LMS$^+$14] to prove non-interference and in [BCS13] to prevent privilege escalation attacks. In both cases, the considered formal models are significantly less detailed than ours and the purpose of the static analyses is different. Though the framework in [LMS$^+$14] can be used to prevent implicit information flows, unlike our approach, the analysis proposed there is not fully automatic, it does not approximate runtime value, thus sacrificing precision, and it was not experimentally evaluated.

Julia is a static analysis tool based on abstract interpretation, first developed for Java and recently extended to Android [PS12]. It is a commercial product and supports many useful features, including class analysis, nullness analysis and termination analysis for Android applications, but it does not track information flows. Moreover, Julia does not handle multi-threading and we are not aware of the existence of a soundness proof for its extension to Android.

Pointer Analysis Pointer analysis aims at over-approximating the set of objects that a program variable can refer to, and it is a well-established and rich research field [KK14, SCD$^+$13, SB15]. The most prominent techniques in pointer analysis are variants of the classical Andersen algorithm [And94], including flow-insensitive analyses [Das00, PB09, HL07, KS13] and flow-sensitive analyses [CBC93, EGH94, Kah08, LC11]; light-weight analyses in the flavor of the unification-based Steensgaard analysis [Ste96], which are flow-insensitive and very efficient; and shape analysis techniques [SRW99], which can be used to prove complex properties about the heap, often at the price of efficiency.

Although pointer analysis of sequential programs is well-studied, much less attention has been paid to pointer analysis of concurrent programs. Most flow-insensitive analyses for sequential programs remain sound for concurrent programs [RR99], because flow-insensitivity forces a sound analysis to consider all the possible interleavings of reads and writes to the heap. Designing a sound flow-sensitive pointer analysis for concurrent programs is more complicated and most flow-sensitive analyses for sequential programs cannot be easily adapted to concurrent programs. Still, flow-sensitive sound analyses for concurrent programs exist. The approach of Rugina and Rinard [RR99] handles concurrent programs with an unbounded number of threads, recursion and dynamic allocations, but it does not allow strong updates on dynamically allocated heap objects. Gotsman *et al.* [GBCS07] proposed a framework to prove complex properties about programs with dynamic allocations by using shape analysis and separation logic, but their approach requires users or external tools to provide annotations, and it is restricted to a bounded number of threads.

CHAPTER 5

Proofs of Chapter 4

Chapter Outline: In Section 5.1 we give the small-step semantics of the local states reduction for the Dalvik bytecode, as well as the reduction rules for activities and threads; in Section 5.2 we give the full abstract semantics; in Section 5.3 we give the soundness proof.

5.1 Concrete Semantics

As in introduced in Definition 1, we require that Dalvik programs are *well-formed*.

From now on, we always consider a fixed well-formed program $P = cls^*$. We give in Table 5.1 the syntax and an informal explanation of the Dalvik statements that were omitted in Table 4.6. The extensions with respect to Chapter 2 are in bold.

5.1.1 Extensions : Waiting Sets and Monitors

`sinvoke` $c\ m\ r^*$	invoke the static method m of the class c with args r^*
`checkcast` $r_s\ \tau$	jump to the next statement if the value of r_s has type τ
`instof` $r_d\ r_s\ \tau$	put **`true`** in r_d iff the value of r_s has type τ
`interrupted` r_t	read and reset the interrupt field of the thread in r_t
`is-interrupted` r_t	read the interrupt field of the thread in r_t
`monitor-enter` r_o	acquire the monitor of the object in r_o
`monitor-exit` r_o	release the monitor of the object in r_o
`wait` r_o	enter the waiting set of the object in r_o

Table 5.1: Syntax and Informal Semantics of Additional Statements

In order to give a full account of Java concurrency we extended our model to include waiting sets and monitors [Java], as well as two other interrupting methods of the Java Thread API. We start by extending the concrete semantics to handle the `wait` statement: we introduce a new semantic domain for waiting states and extend the local state lists domain: we use a special type of state, called *waiting state* and denoted by $\omega = \mathsf{waiting}(j, \ell)$, to model that the thread running the method is currently waiting on some object stored at location ℓ; the integer parameter j stores how many times the object monitor was acquired prior to entering the waiting state. A *local state list* $L^{\#}$ is now a list of local states and waiting states. Since a thread entering a waiting state is paused until it is ready to resume its execution, we assume that a local state list never contains more than one waiting state. Moreover, we assume this waiting state is always the head of the local state list (if present).

Waiting states	ω	$::=$	$\mathsf{waiting}(\ell, j)$
Local state lists	$L^{\#}$	$::=$	$\varepsilon \mid L :: L^{\#} \mid \omega :: L^{\#}$

Statements Description A monitor is a synchronization construct attached to an object, which can be acquired and released by threads, but cannot be acquired by more than one thread at once. Any thread holding an object monitor can start waiting on the object: this makes the thread enter the object waiting set, release the monitor, and pause until it is woken-up, notified or interrupted by another thread. Since we do not model timing aspects in our formalism and *spurious* wake-ups may happen in practice, we make the conservative assumption that waiting threads can non-deterministically wake up at any time. Moreover, we assume that all objects contain two special fields: the acquired field storing the location of the thread currently holding the object monitor, and the m-cnt field counting the number of monitor acquisitions. These fields can only be accessed by the monitor and wait rules.

When `monitor-enter` r_o is called, there are two possibilities. If the m-cnt field of the monitor of the object whose location is stored in r_o is set to 0, it is immediately set to 1 and the corresponding acquired field is set to the location of the acquiring thread. Otherwise, we check that the acquired field points to the location of the acquiring thread: if this is the case, the m-cnt field is incremented by 1 to reflect the presence of multiple acquisitions. A monitor is released only when all its acquisitions have been released via the statement `monitor-exit` r_o, which checks that the running thread holds the monitor of the object whose location is stored in r_o and decrements the monitor counter m-cnt by 1.

The statement `wait` r_o checks that the running thread holds the monitor of the object o whose location is stored in r_o, releases the monitor and pushes on the call stack a waiting state $\mathsf{waiting}(\ell, j)$, where ℓ is the location of o and j tracks how many times the released monitor was acquired before calling `wait` r_o. An uninterrupted thread can exit a waiting state and reacquire back the released monitor j times, provided that another thread does not hold the monitor. If a thread in a waiting state gets interrupted, an IntExcpt exception is thrown, the thread wakes up and starts recovering from the exception.

Finally `interrupted` r_t and `is-interrupted` r_t are simple write or read operations on the interrupt field (inte) of the thread object whose location is stored in r_t.

5.1.2 Local Reduction Relation

Type System

Local registers are untyped in Dalvik, and have default value $\mathbf{0}$. We also assume that for all type τ, there exists a default value $\mathbf{0}_\tau$ that will be used for field initialization. Before giving the concrete semantics of the Dalvik bytecode, we need some definitions. First we define a function $\mathit{type}_H(v)$ that retrieves from the heap H the type of the memory block v is pointing to.

Definition 13. Given a heap H, we let the partial function $\mathit{type}_H(v)$ be defined as follows:

$$\mathit{type}_H(v) = \begin{cases} c & \text{if } v = \ell \wedge H(\ell) = \{\!|c; (f \mapsto v)^*|\!\} \\ \texttt{array}[\tau] & \text{if } v = \ell \wedge H(\ell) = \tau[v^*] \\ \texttt{Intent} & \text{if } v = \ell \wedge H(\ell) = \{\!|@c; (k \mapsto v)^*|\!\} \\ \tau_{\mathit{prim}} & \text{if } v = \mathit{prim} \end{cases}$$

where τ_{prim} is the type of the primitive value prim.

Given a class name c, we let $\mathit{super}(c) = c'$ if there exists a class cls_i such that $\mathit{cls}_i = \texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{\mathit{fld}^*; \mathit{mtd}^*\}$, and $\mathit{inter}(c) = \{c^*\}$ iff there exists a class cls_i such that $\mathit{cls}_i = \texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{\mathit{fld}^*; \mathit{mtd}^*\}$. The subtyping relation is quite simple: a class c is a subclass of its super class $\mathit{super}(c)$ and of the interfaces $\mathit{inter}(c)$ it implements (plus reflexive and transitive closure). There is also a co-variant subtyping rule for array, which is unsound in presence of side-effects (types are checked dynamically at runtime to avoid errors). The typing rules are summarized below.

$$\text{(Sub-Refl)}\ \frac{}{\tau \leq \tau} \qquad \text{(Sub-Trans)}\ \frac{\tau \leq \tau' \qquad \tau' \leq \tau''}{\tau \leq \tau''} \qquad \text{(Sub-Ext)}\ \frac{}{c \leq \mathit{super}(c)} \qquad \text{(Sub-Impl)}\ \frac{c' \in \mathit{inter}(c)}{c \leq c'}$$

$$\text{(Sub-Array)}\ \frac{\tau \leq \tau'}{\texttt{array}[\tau] \leq \texttt{array}[\tau']}$$

Right-Hand Side Evaluation

Let $a[i] = v_i$ whenever $a = \tau[v^*]$ and $o.f = v$ whenever $o = \{\!|c; (f_i \mapsto v_i)^*, f \mapsto v|\!\}$. We define in Table 5.2 the relation $\Sigma[\![\mathit{rhs}]\!]$ that evaluates a right-hand side expression in a given local configuration Σ.

$$\text{(Rhs-Register)}\ \frac{}{\Sigma[\![r]\!] = R(r)} \qquad \text{(Rhs-Array)}\ \frac{\ell = \Sigma[\![r_a]\!] \quad a = H(\ell) \quad j = \Sigma[\![r_{idx}]\!]}{\Sigma[\![r_a[r_{idx}]]\!] = a[j]} \qquad \text{(Rhs-Object)}\ \frac{\ell = \Sigma[\![r_o]\!] \quad o = H(\ell)}{\Sigma[\![r_o.f]\!] = o.f} \qquad \text{(Rhs-Static)}\ \frac{}{\Sigma[\![c.f]\!] = S(c.f)}$$

$$\text{(Rhs-Prim)}\ \frac{}{\Sigma[\![prim]\!] = prim}$$

Convention: in all the rules, let $\Sigma = \ell_r \cdot \alpha_c \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha_c = \langle pp \cdot \_ \cdot st^* \cdot R \rangle :: \alpha'$ or $\alpha_c = \texttt{AbNormal}(\langle pp \cdot \_ \cdot st^* \cdot R \rangle :: \alpha')$.

Table 5.2: Evaluation of Right-hand Sides ($\Sigma[\![rhs]\!] = v$)

Instruction Fetching

We recall that the definition of the local reduction relation uses an auxiliary relation $\Sigma, st \Downarrow \Sigma'$, which means that the execution of the statement st in Σ produces Σ'. The simplest rule defining a local reduction $\Sigma \rightsquigarrow \Sigma'$ just fetches the next statement st to run and performs a look-up on the auxiliary relation $\Sigma, st \Downarrow \Sigma'$. Formally:

$$\text{(R-NextStm)}\ \frac{\Sigma, \mathit{get\text{-}stm}(\Sigma) \Downarrow \Sigma'}{\Sigma \rightsquigarrow \Sigma'}$$

We are finally ready to give the semantics of the Dalvik bytecode relation: the standard operation are in Table 5.3 and Table 5.4, while the new operations are given in Table 5.5 and Table 5.6.

5.1.3 Global Rules Descriptions

Serialization

All the activities running on some Android device are *sand-boxed*, in order to provide some security guarantees. Inter-component communications are still allowed through the intent mechanism: activities can exchange objects using intents, which are a special kind of object storing data in a dictionary-like structure. When an activity sends an intent to some activity, a copy of this intent is given to the receiver activity. This copying is performed by a recursive *serialization* procedure, and there is therefore no object-sharing between different activities.

We give the first (simplified) formal account to the serialization procedures in § 2.3.4, here we extend it. We model serialization using a set of derivation rules for fact of the form $\Gamma \vdash ser^H_{Val}(v) = (v', H', \Gamma')$ and $\Gamma \vdash ser^H_{Blk}(b) = (b', H', \Gamma')$, where Γ and Γ'

(R-GOTO)
$$\frac{}{\Sigma, \texttt{goto}\ pc' \Downarrow \Sigma[pc \mapsto pc']}$$

(R-TRUE)
$$\frac{\Sigma[\![r_1]\!] \olessthan \Sigma[\![r_2]\!]}{\Sigma, \texttt{if}_{\olessthan}\ r_1\ r_2\ \texttt{then}\ pc' \Downarrow \Sigma[pc \mapsto pc']}$$

(R-FALSE)
$$\frac{\neg(\Sigma[\![r_1]\!] \olessthan \Sigma[\![r_2]\!])}{\Sigma, \texttt{if}_{\olessthan}\ r_1\ r_2\ \texttt{then}\ pc' \Downarrow \Sigma^+}$$

(R-MOVEREG)
$$\frac{v = \Sigma[\![rhs]\!] \qquad R' = R[r \mapsto v]}{\Sigma, \texttt{move}\ r\ rhs \Downarrow \Sigma^+[R \mapsto R']}$$

(R-MOVEFLD)
$$\frac{v = \Sigma[\![rhs]\!] \qquad \ell = \Sigma[\![r_o]\!] \qquad o = H(\ell) \qquad H' = H[\ell \mapsto o[f \mapsto v]]}{\Sigma, \texttt{move}\ r_o.f\ rhs \Downarrow \Sigma^+[H \mapsto H']}$$

(R-MOVEARR)
$$\frac{\begin{array}{c} v = \Sigma[\![rhs]\!] \qquad \ell = \Sigma[\![r_a]\!] \qquad type_H(\ell) = \texttt{array}[\tau] \\ type_H(v) \leq \tau \qquad a = H(\ell) \qquad j = \Sigma[\![r_{idx}]\!] \qquad H' = H[\ell \mapsto a[j \mapsto v]] \end{array}}{\Sigma, \texttt{move}\ r_a[r_{idx}]\ rhs \Downarrow \Sigma^+[H \mapsto H']}$$

(R-MOVESFLD)
$$\frac{v = \Sigma[\![rhs]\!] \qquad S' = S[c'.f \mapsto v]}{\Sigma, \texttt{move}\ c'.f\ rhs \Downarrow \Sigma^+[S \mapsto S']}$$

(R-UNOP)
$$\frac{v = \odot\Sigma[\![r_s]\!] \qquad R' = [r_d \mapsto v]}{\Sigma, \texttt{unop}_{\odot}\ r_d\ r_s \Downarrow \Sigma^+[R \mapsto R']}$$

(R-BINOP)
$$\frac{v = \Sigma[\![r_1]\!] \oplus \Sigma[\![r_2]\!] \qquad R' = R[r_d \mapsto v]}{\Sigma, \texttt{binop}_{\oplus}\ r_d\ r_1\ r_2 \Downarrow \Sigma^+[R \mapsto R']}$$

(R-NEWOBJ)
$$\frac{\begin{array}{c} o = \{\!|c'; (f_\tau \mapsto \mathbf{0}_\tau)^*|\!\} \\ \ell = p_{c,m,pc} \notin dom(H) \qquad H' = H[\ell \mapsto o] \qquad R' = R[r_d \mapsto \ell] \end{array}}{\Sigma, \texttt{new}\ r_d\ c' \Downarrow \Sigma^+[H \mapsto H', R \mapsto R']}$$

(R-NEWARR)
$$\frac{\begin{array}{c} len = \Sigma[\![r_l]\!] \\ a = \tau[(\mathbf{0}_\tau)^{j \leq len}] \qquad \ell = p_{c,m,pc} \notin dom(H) \qquad H' = H[\ell \mapsto a] \qquad R' = R[r_d \mapsto \ell] \end{array}}{\Sigma, \texttt{newarray}\ r_d\ r_l\ \tau \Downarrow \Sigma^+[H \mapsto H', R \mapsto R']}$$

(R-CAST)
$$\frac{\ell = \Sigma[\![r_s]\!] \qquad type_H(\ell) \leq \tau}{\Sigma, \texttt{checkcast}\ r_s\ \tau \Downarrow \Sigma^+}$$

(R-INSTOFTRUE)
$$\frac{\ell = \Sigma[\![r_s]\!] \qquad type_H(\ell) \leq \tau \qquad R' = R[r_d \mapsto \texttt{true}]}{\Sigma, \texttt{instof}\ r_d\ r_s\ \tau \Downarrow \Sigma^+[R \mapsto R']}$$

(R-INSTOFFALSE)
$$\frac{\ell = \Sigma[\![r_s]\!] \qquad type_H(\ell) \not\leq \tau \qquad R' = R[r_d \mapsto \texttt{false}]}{\Sigma, \texttt{instof}\ r_d\ r_s\ \tau \Downarrow \Sigma^+[R \mapsto R']}$$

Convention: let $pp = c, m, pc$ and let $\Sigma = \_ \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha = \langle c, m, pc \cdot \_ \cdot \_ \cdot R \rangle :: \alpha'$. We recall that Σ^+ stands for Σ where pc is replaced by $pc + 1$.

Table 5.3: Small step semantics of μ-Dalvik$_A$ - Standard Statements (continued in Table 5.4)

(R-Return)

$$\frac{\begin{array}{c}\alpha = \langle c, m, pc \cdot \_ \cdot \_ \cdot R\rangle :: \langle c', m', pc' \cdot v^* \cdot st^* \cdot R'\rangle :: \alpha_0 \\ \alpha'' = \langle c', m', pc' + 1 \cdot v^* \cdot st^* \cdot R'[r_{\mathsf{res}} \mapsto \Sigma\llbracket r_{\mathsf{res}}\rrbracket]\rangle :: \alpha_0\end{array}}{\Sigma, \texttt{return} \Downarrow \Sigma[\alpha \mapsto \alpha'']}$$

(R-SCall)

$$\frac{\begin{array}{c}lookup(c', m') = (c', st^*) \qquad sign(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau \\ R' = ((r_j \mapsto \mathbf{0})^{j \le loc}, (r_{loc+k} \mapsto \Sigma\llbracket r'_k\rrbracket)^{k \le n}) \qquad \alpha'' = \langle c', m', 0 \cdot (\Sigma\llbracket r'_k\rrbracket)^{k \le n} \cdot st^* \cdot R'\rangle :: \alpha\end{array}}{\Sigma, \texttt{sinvoke}\ c'\ m'\ r'_1, \ldots, r'_n \Downarrow \Sigma[\alpha \mapsto \alpha'']}$$

(R-Call)

$$\frac{\begin{array}{c}\ell = \Sigma\llbracket r_o\rrbracket \qquad lookup(type_H(\ell), m') = (c', st^*) \\ sign(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau \qquad R' = ((r_j \mapsto \mathbf{0})^{j \le loc}, r_{loc+1} \mapsto \ell, (r_{loc+1+k} \mapsto \Sigma\llbracket r'_k\rrbracket)^{k \le n}) \\ \alpha'' = \langle c', m', 0 \cdot (\Sigma\llbracket r'_k\rrbracket)^{k \le n} \cdot st^* \cdot R'\rangle :: \alpha\end{array}}{\Sigma, \texttt{invoke}\ r_o\ m'\ r'_1, \ldots, r'_n \Downarrow \Sigma[\alpha \mapsto \alpha'']}$$

(R-NewIntent)

$$\frac{\begin{array}{c}i = \{\!|@c'; \cdot|\!\} \\ \ell = p_{c,m,pc} \notin dom(H) \qquad H' = H[\ell \mapsto i] \qquad R' = R[r_d \mapsto \ell]\end{array}}{\Sigma, \texttt{newintent}\ r_d\ c' \Downarrow \Sigma^+[H \mapsto H', R \mapsto R']}$$

(R-PutExtra)

$$\frac{\ell = \Sigma\llbracket r_i\rrbracket \qquad i = H(\ell) \qquad k = \Sigma\llbracket r_k\rrbracket \qquad v = \Sigma\llbracket r_v\rrbracket \qquad H' = H[\ell \mapsto i[k \mapsto v]]}{\Sigma, \texttt{put-extra}\ r_i\ r_k\ r_v \Downarrow \Sigma^+[H \mapsto H']}$$

(R-GetExtra)

$$\frac{\begin{array}{c}\ell = \Sigma\llbracket r_i\rrbracket \\ k = \Sigma\llbracket r_k\rrbracket \qquad H(\ell) = i \qquad type_H(i.k) \le \tau \qquad v = i.k \qquad R' = R[r_{\mathsf{res}} \mapsto v]\end{array}}{\Sigma, \texttt{get-extra}\ r_i\ r_k\ \tau \Downarrow \Sigma^+[R \mapsto R']}$$

(R-StartAct)

$$\frac{\ell = \Sigma\llbracket r_i\rrbracket \qquad H(\ell) = i \qquad \pi' = i :: \pi}{\Sigma, \texttt{start-act}\ r_i \Downarrow \Sigma^+[\pi \mapsto \pi']}$$

Convention: let $pp = c, m, pc$ and let $\Sigma = \_ \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha = \langle c, m, pc \cdot \_ \cdot \_ \cdot R\rangle :: \alpha'$. We recall that Σ^+ stands for Σ where pc is replaced by $pc + 1$.

Table 5.4: Small step semantics of μ-Dalvik$_A$ - Standard Statements (continuation of table Table 5.3)

Exception Rules

(R-Throw)
$$\frac{\ell = \Sigma\llbracket r_i \rrbracket \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\}}{\Sigma, \texttt{throw}\ r_i \Downarrow \Sigma[\alpha \mapsto \texttt{AbNormal}(\alpha)][r_{\mathsf{excpt}} \mapsto \ell]}$$

(R-MoveException)
$$\frac{\ell = \Sigma\llbracket r_{\mathsf{excpt}} \rrbracket}{\Sigma, \texttt{move-except}\ r_d \Downarrow \Sigma^+[r_d \mapsto \ell]}$$

(R-Caught)
$$\frac{\begin{array}{c}\ell = \Sigma_A\llbracket r_{\mathsf{excpt}} \rrbracket \\ H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\} \qquad \mathsf{ExcptTable}(c, m, pc, c') = pc' \\ \alpha_c = \langle c, m, pc' \cdot \_ \cdot \_ \cdot R \rangle :: \alpha' \end{array}}{\Sigma_A \rightsquigarrow \Sigma_A[\alpha_A \mapsto \alpha_c]}$$

(R-UnCaught)
$$\frac{\begin{array}{c}\ell = \Sigma_A\llbracket r_{\mathsf{excpt}} \rrbracket \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\} \\ \mathsf{ExcptTable}(c, m, pc, c') = \bot \end{array}}{\Sigma_A \rightsquigarrow \Sigma_A[\alpha_A \mapsto \texttt{AbNormal}(\alpha')][r_{\mathsf{excpt}} \mapsto \ell]}$$

Thread Rules

(R-StartThread)
$$\frac{\ell = \Sigma\llbracket r_i \rrbracket \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\} \qquad \gamma' = \ell :: \gamma}{\Sigma, \texttt{start-thread}\ r_i \Downarrow \Sigma^+[\gamma \mapsto \gamma']}$$

(R-InterruptThread)
$$\frac{\begin{array}{c}\ell = \Sigma\llbracket r_i \rrbracket \\ H(\ell) = \{\!|c'; (f \mapsto v)^*, \mathsf{inte} \mapsto \_|\!\} \qquad H' = H[\ell \mapsto \{\!|c'; (f \mapsto v)^*, \mathsf{inte} \mapsto \texttt{true}|\!\}] \end{array}}{\Sigma, \texttt{interrupt}\ r_i \Downarrow \Sigma^+[H \mapsto H']}$$

(R-InterruptedThread)
$$\frac{\begin{array}{c}\ell = \Sigma\llbracket r_i \rrbracket \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \mathsf{inte} \mapsto u|\!\} \\ H' = H[\ell \mapsto \{\!|c'; (f \mapsto v)^*, \mathsf{inte} \mapsto \texttt{false}|\!\}] \end{array}}{\Sigma, \texttt{interrupted}\ r_i \Downarrow \Sigma^+[r_{\mathsf{res}} \mapsto u, H \mapsto H']}$$

(R-IsInterruptedThread)
$$\frac{\ell = \Sigma\llbracket r_i \rrbracket \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \mathsf{inte} \mapsto u|\!\}}{\Sigma, \texttt{is-interrupted}\ r_i \Downarrow \Sigma^+[r_{\mathsf{res}} \mapsto u]}$$

(R-JoinThread)
$$\frac{\begin{array}{c}H(\ell_r) = \{\!|c_r; (f_r \mapsto v_r)^*, \mathsf{inte} \mapsto \texttt{false}|\!\} \\ \ell = \Sigma\llbracket r_i \rrbracket \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \mathsf{finished} \mapsto \texttt{true}|\!\} \end{array}}{\Sigma, \texttt{join}\ r_i \Downarrow \Sigma^+}$$

(R-InterruptJoin)
$$\frac{\begin{array}{c}H(\ell_r) = \{\!|c_r; (f_r \mapsto v_r)^*, \mathsf{inte} \mapsto \texttt{true}|\!\} \qquad o = \{\!|c_r; (f_r \mapsto v_r)^*, \mathsf{inte} \mapsto \texttt{false}|\!\} \\ p_{c,m,pc} \notin dom(H) \qquad H' = H, p_{c,m,pc} \mapsto \{\!|\mathsf{IntExcpt};|\!\} \qquad \alpha_c = \texttt{AbNormal}(\alpha[r_{\mathsf{excpt}} \mapsto p_{c,m,pc}]) \end{array}}{\Sigma, \texttt{join}\ r_i \Downarrow \Sigma[\alpha \mapsto \alpha_c, H \mapsto H'[\ell_r \mapsto o]]}$$

Convention: let $\Sigma = \ell_r \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha = \langle c, m, pc \cdot \_ \cdot \_ \cdot R \rangle :: \alpha'$ (apart when specified otherwise), and $\Sigma_A = \ell_r \cdot \alpha_A \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha_A = \texttt{AbNormal}(\alpha)$. We recall that Σ^+ stands for Σ where pc is replaced by $pc + 1$.

Table 5.5: Small step semantics of μ-Dalvik$_A$ - New Statements (continued in Table 5.6)

Monitor and Wait Rules

(R-MonitorEnter1)

$$\frac{\ell = \Sigma[\![r_i]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \textsf{acquired} \mapsto \_, \textsf{m-cnt} \mapsto 0|\!\} \qquad o' = \{\!|c'; (f \mapsto v)^*, \textsf{acquired} \mapsto \ell_r, \textsf{m-cnt} \mapsto 1|\!\}}{\Sigma, \texttt{monitor-enter}\ r_i \Downarrow \Sigma^+[H \mapsto H[\ell \mapsto o']]}$$

(R-MonitorEnter2)

$$\frac{\ell = \Sigma[\![r_i]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \textsf{acquired} \mapsto \ell_r, \textsf{m-cnt} \mapsto j|\!\} \qquad o' = \{\!|c'; (f \mapsto v)^*, \textsf{acquired} \mapsto \ell_r, \textsf{m-cnt} \mapsto j+1|\!\} \qquad j > 0}{\Sigma, \texttt{monitor-enter}\ r_i \Downarrow \Sigma^+[H \mapsto H[\ell \mapsto o']]}$$

(R-MonitorExit)

$$\frac{\ell = \Sigma[\![r_i]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \textsf{acquired} \mapsto \ell_r, \textsf{m-cnt} \mapsto j+1|\!\} \qquad o' = \{\!|c'; (f \mapsto v)^*, \textsf{acquired} \mapsto \ell_r, \textsf{m-cnt} \mapsto j|\!\} \qquad j \geq 0}{\Sigma, \texttt{monitor-exit}\ r_i \Downarrow \Sigma^+[H \mapsto H[\ell \mapsto o']]}$$

(R-StartWait)

$$\frac{\ell = \Sigma[\![r_i]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \textsf{acquired} \mapsto \ell_r, \textsf{m-cnt} \mapsto j|\!\} \qquad o' = \{\!|c'; (f \mapsto v)^*, \textsf{acquired} \mapsto \ell_r, \textsf{m-cnt} \mapsto 0|\!\} \qquad j > 0}{\Sigma, \texttt{wait}\ r_i \Downarrow \Sigma[\alpha \mapsto \textsf{waiting}(\ell, j) :: \alpha, H \mapsto H[\ell \mapsto o']]}$$

(R-StopWait)

$$\frac{H(\ell_r) = \{\!|c_r; (f_r \mapsto v_r)^*, \textsf{inte} \mapsto \texttt{false}|\!\} \qquad \alpha = \textsf{waiting}(\ell_o, j) :: \alpha_0 \qquad H(\ell_o) = \{\!|c'; (f \mapsto v)^*, \textsf{acquired} \mapsto \_, \textsf{m-cnt} \mapsto 0|\!\} \qquad o' = \{\!|c'; (f \mapsto v)^*, \textsf{acquired} \mapsto \ell_r, \textsf{m-cnt} \mapsto j|\!\}}{\Sigma \rightsquigarrow \Sigma^+[\alpha \mapsto \alpha_0, H \mapsto H[\ell_o \mapsto o']]}$$

(R-InterruptWait)

$$\frac{H(\ell_r) = \{\!|c_r; (f_r \mapsto v_r)^*, \textsf{inte} \mapsto \texttt{true}|\!\} \qquad \alpha = \textsf{waiting}(\_, \_) :: \alpha_0 \qquad p_{c,m,pc} \notin dom(H) \qquad o = \{\!|c_r; (f_r \mapsto v_r)^*, \textsf{inte} \mapsto \texttt{false}|\!\} \qquad o_e = \{\!|\textsf{IntExcpt}; |\!\}}{\Sigma \rightsquigarrow \Sigma[\alpha \mapsto \texttt{AbNormal}(\alpha_0[r_{\textsf{excpt}} \mapsto \ell_e]), H \mapsto H[p_{c,m,pc} \mapsto o_e, \ell_r \mapsto o]]}$$

Convention: let $\Sigma = \ell_r \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha = \langle c, m, pc \cdot \_ \cdot \_ \cdot R\rangle :: \alpha'$ (apart when specified otherwise), and $\Sigma_A = \ell_r \cdot \alpha_A \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha_A = \texttt{AbNormal}(\alpha)$. We recall that Σ^+ stands for Σ where pc is replaced by $pc + 1$.

Table 5.6: Small step semantics of μ-Dalvik$_A$ - New Statements (continuation of Table 5.5)

are serialization context consisting of a list of key-value bindings of locations of the form $(p_\lambda \mapsto p'_\lambda)$ (notice that both locations have the same annotation). Sersialization contexts store, for each already serialized location ℓ, the fresh location ℓ' that was used to replace ℓ. This way if the same location is encountered twice (or more) during the serialization process, it will be serialized by the same location each time. Intuitively, if $ser^H_{Val}(v) = (v', H', \Gamma')$ (resp. $\Gamma \vdash ser^H_{Blk}(b) = (b', H', \Gamma')$) is derivable then v' (resp. b') is the serialized version of the value v (resp. block b), H' is the heap containing all the serialized version of the objects encountered, and Γ' is the history of all serialized locations. We refer to Table 5.7 for the formal statement of the serialization rules.

$$\frac{}{\Gamma \vdash ser^H_{Val}(prim) = (prim, \cdot, \Gamma)} \qquad \frac{(p_\lambda \mapsto p'_\lambda) \in \Gamma}{\Gamma \vdash ser^H_{Val}(p_\lambda) = (p'_\lambda, \cdot, \Gamma)}$$

$$\frac{p'_\lambda \text{ fresh location} \qquad \begin{array}{c} p_\lambda \notin dom(\Gamma) \\ \Gamma, p_\lambda \mapsto p'_\lambda \vdash ser^H_{Blk}(H(p_\lambda)) = (b, H'', \Gamma') \end{array} \qquad H' = H'', p'_\lambda \mapsto b}{\Gamma \vdash ser^H_{Val}(p_\lambda) = (p'_\lambda, H', \Gamma')}$$

$$\frac{\Gamma_0 = \Gamma \qquad \forall i \in [1,n] : \Gamma_{i-1} \vdash ser^H_{Val}(v_i) = (u_i, H_i, \Gamma_i) \qquad H' = H_1, \dots, H_n}{\Gamma \vdash ser^H_{Blk}(\{\!|c'; (f_i \mapsto v_i)^{i \le n}|\!\}) = (\{\!|c'; (f_i \mapsto u_i)^{i \le n}|\!\}, H', \Gamma_n)}$$

$$\frac{\Gamma_0 = \Gamma \qquad \forall i \in [1,n] : \Gamma_{i-1} \vdash ser^H_{Val}(v_i) = (u_i, H_i, \Gamma_i) \qquad H' = H_1, \dots, H_n}{\Gamma \vdash ser^H_{Blk}(\tau[(v_i)^{i \le n}]) = (\tau[(u_i)^{i \le n}], H', \Gamma_n)}$$

$$\frac{\Gamma_0 = \Gamma \qquad \forall i \in [1,n] : \Gamma_{i-1} \vdash ser^H_{Val}(v_i) = (u_i, H_i, \Gamma_i) \qquad H' = H_1, \dots, H_n}{\Gamma \vdash ser^H_{Blk}(\{\!|@c'; (k_i \mapsto v_i)^{i \le n}|\!\}) = (\{\!|@c'; (k_i \mapsto u_i)^{i \le n}|\!\}, H', \Gamma_n)}$$

Conventions: environments (denoted by $\Gamma, \Gamma' \dots$) are partial mappings from the set of all locations to itself.

Table 5.7: Serialization rules

Threads and Activities

Before giving a global reduction relation, we need some definitions. We start by formally define what a thread class and an activity class are.

Definition 14. A class *cls* is a *thread class* if and only if $cls = \texttt{cls}\ c \le c'\ \texttt{imp}\ c^*\ \{fld^*; mtd^*\}$ for some $c' \le$ Thread. A *thread* is an instance of a thread class. We stipulate that each thread implements the method run, has a boolean field inte stating whether the thread was interrupted and a boolean field finished stating whether the thread has finished or not.

Definition 15. A class *cls* is an *activity class* if and only if $cls = \texttt{cls}\ c \leq c'\ \texttt{imp}\ c^*\ \{fld^*; mtd^*\}$ for some $c' \leq \mathsf{Activity}$. An *activity* is an instance of an activity class. We stipulate that each activity has the following fields: (1) $\mathsf{finished}$: a boolean flag stating whether the activity has finished or not; (2) intent: a location to the intent which started the activity; (3) result: a location to an intent storing the result of the activity computation; and (4) *parent*: a location to the parent activity, i.e., the activity which started the present one.

Each activity provides a set of *event handlers* which are callbacks methods used to respond to user inputs: for all activity class c, let $handlers(c) = \{m_1, \dots, m_n\}$ be the set of callback method names of c. We model the activity lifecycle (see [PS14]) by a set of activity states *ActStates* and a transition relation $Lifecycle \subseteq ActStates \times ActStates$. For each activity state s, we let $cb(c, s)$ be the set of callbacks for the activity c in the state s. Moreover we assume that for the *running* state, $cb(c, running) = handlers(c)$.

We also need the notion of *callback stack*: a callback stack is the initial call stack of a new activity frame, created upon a callback method invocation:

Definition 16. Given a location ℓ pointing to an activity of class c, we let $\alpha_{\ell.s}$ stand for an arbitrary *callback stack* for state s, i.e., any call stack $\langle c', m, 0 \cdots st^* \cdot R\rangle :: \varepsilon$, where $(c', st^*) = lookup(c, m)$ for some $m \in cb(c, s)$, $sign(c', m) = \tau_1, \dots, \tau_n \xrightarrow{loc} \tau$ and:

$$R = ((r_i \mapsto \mathbf{0})^{i \leq loc}, r_{loc+1} \mapsto \ell, (r_{loc+1+j} \mapsto v_j)^{j \leq n}),$$

for some values $v_1, \dots, v_n$ of the correct type $\tau_1, \dots, \tau_n$.

Global Reduction Relation

We are now ready to give the global reduction relation. First we will describe two new rules which were not given Chapter 4 and can be found in Table 5.8: rule (T-Intent) allows a thread to transfer an intent to the activity that spawned it, and rule (T-Thread) allows a thread to transfer a location in its pending thread stack to the activity that spawned it.

Table 5.9 and Table 5.10 recall the rules introduced in Table 2.5 and Table 2.6 to model the activity lifecycle mechanism, with only minor modifications to include the thread pool. Rule (A-Active) executes the statements of the active frame in the activity stack, using the reduction relation for local configurations. Rule (A-Deactivate) stops an activity frame from being active when it has completed its computations. Rule (A-Step) models the transition of the top-most activity frame from one activity state to one of its successor in the activity lifecycle, and executes a callback method from this new activity state, provided some side conditions related to the pending activity stack and the $\mathsf{finished}$ field of the activity object are met. Rule (A-Destroy) models the removal of a finished activity from the activity stack. Rule (A-Back) is used by the system to finished the top-most activity when the user hits the back button. Rule (A-Replace) models the screen orientation changing, by destroying and restarting the top-most activity. Rule

(T-Reduce)

$$\frac{\ell_t \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S \rightsquigarrow \ell_t \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S'}{\Omega \cdot \Xi :: \langle\!\langle \ell, \ell_t, \pi, \gamma, \alpha \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega \cdot \Xi :: \langle\!\langle \ell, \ell_t, \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H' \cdot S'}$$

(T-Kill)

$$\frac{\begin{array}{c} H(\ell') = \{\!| c; (f \mapsto v)^*, \mathsf{finished} \mapsto \_ |\!\} \\ H' = \ H[\ell' \mapsto \{\!| c; (f \mapsto v)^*, \mathsf{finished} \mapsto \mathtt{true} |\!\}] \end{array}}{\Omega \cdot \Xi :: \langle\!\langle \ell, \ell', \varepsilon, \varepsilon, \overline{\alpha} \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega \cdot \Xi :: \Xi' \cdot H' \cdot S}$$

(T-Intent)

$$\frac{(\varphi, \varphi') \in \{(\langle \ell, s, \pi, \gamma, \alpha \rangle, \langle \ell, s, i :: \pi, \gamma, \alpha \rangle), (\underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}, \underline{\langle \ell, s, i :: \pi, \gamma, \alpha \rangle})\}}{\begin{array}{c} \Omega :: \varphi :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', i :: \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \\ \Omega :: \varphi' :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S \end{array}}$$

(T-Thread)

$$\frac{(\varphi, \varphi') \in \{(\langle \ell, s, \pi, \gamma, \alpha \rangle, \langle \ell, s, \pi, \ell_t :: \gamma, \alpha \rangle), (\underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}, \underline{\langle \ell, s, \pi, \ell_t :: \gamma, \alpha \rangle})\}}{\begin{array}{c} \Omega :: \varphi :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \gamma' :: \ell_t :: \gamma'', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \\ \Omega :: \varphi' :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \gamma' :: \gamma'', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S \end{array}}$$

(A-ThreadStart)

$$\frac{\begin{array}{c} \varphi = \underline{\langle \ell, s, \pi, \gamma :: \ell' :: \gamma', \alpha \rangle} \\ \varphi' = \underline{\langle \ell, s, \pi, \gamma :: \gamma', \alpha \rangle} \qquad \psi = \langle\!\langle \ell, \ell', \varepsilon, \varepsilon, \alpha' \rangle\!\rangle \\ H(\ell') = \{\!| c'; (f \mapsto v)^* |\!\} \qquad \mathit{lookup}(c', \mathsf{run}) = (c'', st^*) \\ \mathit{sign}(c'', \mathsf{run}) = \mathsf{Thread} \xrightarrow{loc} \mathsf{Void} \\ \alpha' = \langle c'', \mathsf{run}, 0 \cdot \ell' \cdot st^* \cdot (r_k \mapsto \mathbf{0})^{k \leq loc}, r_{loc+1} \mapsto \ell' \rangle \end{array}}{\Omega :: \varphi :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \varphi' :: \Omega' \cdot \psi :: \Xi \cdot H \cdot S}$$

Table 5.8: New Global Reduction Rules

(A-Hidden) allows an activity in the background to take precedence over the foreground activity, stopping or destroying it. Rule (A-Start) allows to start a new activity: the top-most activity must be paused or stopped, and must have an intent i sent to some activity c in its pending activity stack: a new activity of class c is added to the top of the activity stack, its intent field is set to a serialized copy of i and its *parent* field is set to the starting activity. Rule (A-Swap) allows a parent activity to come back to the foreground, assuming the foreground activity is finished and is one of its child activities. Finally, rule (A-Result) allows the top-most activity to return the result of its computation to the parent activity, provided that the top-most activity is finished: a serialized copy of the result is sent to the parent activity, which becomes active and executes the *onActivityResult* callback.

(A-Active)

$$\frac{\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S \rightsquigarrow \ell \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S'}{\Omega :: \underline{\langle \ell, s, \pi, \gamma, \alpha \rangle} :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \underline{\langle \ell, s, \pi', \gamma', \alpha' \rangle} :: \Omega' \cdot \Xi \cdot H' \cdot S'}$$

(A-Deactivate)

$$\frac{}{\Omega :: \underline{\langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle} :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle :: \Omega' \cdot \Xi \cdot H \cdot S}$$

(A-Step)

$$\frac{\begin{array}{c}(s, s') \in \mathit{Lifecycle} \\ \pi \neq \varepsilon \Rightarrow (s, s') = (\mathit{running}, \mathit{onPause}) \\ H(\ell).\mathsf{finished} = \mathtt{true} \Rightarrow (s, s') \in \{(\mathit{running}, \mathit{onPause}), (\mathit{onPause}, \mathit{onStop}), (\mathit{onStop}, \mathit{onDestroy})\}\end{array}}{\langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \underline{\langle \ell, s', \pi, \gamma, \alpha_{\ell.s'} \rangle} :: \Omega \cdot \Xi \cdot H \cdot S}$$

(A-Destroy)

$$\frac{H(\ell).\mathsf{finished} = \mathtt{true}}{\Omega :: \langle \ell, \mathit{onDestroy}, \pi, \gamma, \overline{\alpha} \rangle :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \Omega' \cdot \Xi \cdot H \cdot S}$$

(A-Back)

$$\frac{H' = H[\ell \mapsto H(\ell)[\mathsf{finished} \mapsto \mathtt{true}]]}{\langle \ell, \mathit{running}, \varepsilon, \gamma, \overline{\alpha} \rangle :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \langle \ell, \mathit{running}, \varepsilon, \gamma, \overline{\alpha} \rangle :: \Omega \cdot \Xi \cdot H' \cdot S}$$

(A-Replace)

$$\frac{\begin{array}{c}H(\ell) = \{\!|c; (f_\tau \mapsto v)^*, \mathsf{finished} \mapsto u|\!\} \\ p_c \notin \mathit{dom}(H) \qquad o = \{\!|c; (f_\tau \mapsto \mathbf{0}_\tau)^*, \mathsf{finished} \mapsto \mathtt{false}|\!\} \qquad H' = H, p_c \mapsto o\end{array}}{\langle \ell, \mathit{onDestroy}, \pi, \gamma, \overline{\alpha} \rangle :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \quad \underline{\langle p_c, \mathit{constructor}, \pi, \gamma, \alpha_{p_c.\mathit{constructor}} \rangle} :: \Omega \cdot \Xi \cdot H' \cdot S}$$

(A-Hidden)

$$\frac{\begin{array}{c}\varphi = \langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle \\ s \in \{\mathit{onResume}, \mathit{onPause}\} \qquad (s', s'') \in \{(\mathit{onPause}, \mathit{onStop}), (\mathit{onStop}, \mathit{onDestroy})\}\end{array}}{\varphi :: \Omega :: \langle \ell', s', \pi', \gamma', \overline{\alpha}' \rangle :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \varphi :: \Omega :: \underline{\langle \ell', s'', \pi', \gamma', \alpha_{\ell'.s''} \rangle} :: \Omega' \cdot \Xi \cdot H \cdot S}$$

Conventions: the activity stack on the left-hand side does not contain underlined frames, with the exception of (A-Deactivate) and (A-Activate)

Table 5.9: Reduction Rules for Configurations ($\Omega \cdot \Xi \cdot H \cdot S \Rightarrow \Omega' \cdot \Xi' \cdot H' \cdot S'$), continued in Table 5.10

(A-START)

$$\frac{\begin{array}{c} s \in \{onPause, onStop\} \qquad i = \{\!| @c; (k \mapsto v)^* |\!\} \qquad \emptyset \vdash ser^{H}_{Blk}(i) = (i', H') \\ p_c, p'_{in(c)} \notin dom(H, H') \qquad o = \{\!| c; (f_\tau \mapsto \mathbf{0}_\tau)^*, \mathsf{finished} \mapsto \mathtt{false}, \mathsf{intent} \mapsto p'_{in(c)}, parent \mapsto \ell |\!\} \\ H'' = H, H', p_c \mapsto o, p'_{in(c)} \mapsto i' \end{array}}{\begin{array}{c} \langle \ell, s, i :: \pi, \gamma, \overline{\alpha} \rangle :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \\ \underline{\langle p_c, constructor, \varepsilon, \varepsilon, \alpha_{p_c.constructor} \rangle} :: \langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle :: \Omega \cdot \Xi \cdot H'' \cdot S \end{array}}$$

(A-SWAP)

$$\frac{\begin{array}{c} \varphi' = \langle \ell', onPause, \varepsilon, \gamma', \overline{\alpha}' \rangle \\ H(\ell').\mathsf{finished} = \mathtt{true} \qquad \varphi = \langle \ell, s, i :: \pi, \gamma, \overline{\alpha} \rangle \qquad s \in \{onPause, onStop\} \qquad H(\ell').parent = \ell \end{array}}{\varphi' :: \varphi :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \varphi :: \varphi' :: \Omega \cdot \Xi \cdot H \cdot S}$$

(A-RESULT)

$$\frac{\begin{array}{c} \varphi' = \langle \ell', onPause, \varepsilon, \gamma', \overline{\alpha}' \rangle \\ H(\ell').\mathsf{finished} = \mathtt{true} \qquad \varphi = \langle \ell, s, \varepsilon, \gamma, \overline{\alpha} \rangle \qquad s \in \{onPause, onStop\} \qquad H(\ell').parent = \ell \\ \emptyset \vdash ser^{H}_{Val}(H(\ell').\mathsf{result}) = (w', H') \qquad H'' = (H, H')[\ell \mapsto H(\ell)[\mathsf{result} \mapsto w']] \end{array}}{\varphi' :: \varphi :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \quad \underline{\langle \ell, s, \varepsilon, \gamma, \alpha_{\ell.onActivityResult} \rangle} :: \varphi' :: \Omega \cdot \Xi \cdot H'' \cdot S}$$

Conventions: the activity stack on the left-hand side does not contain underlined frames, with the exception of (A-DEACTIVATE) and (A-ACTIVATE)

Table 5.10: Reduction Rules for Configurations ($\Omega \cdot \Xi \cdot H \cdot S \Rightarrow \Omega' \cdot \Xi' \cdot H' \cdot S'$), continuation of Table 5.9

5.2 Abstract Semantics

Lifting functions

We first give the formal definition of the $\mathsf{hlift}(;)$ and $\hat{\sqcup}$ functions, that we informally described in § 4.4.

$$\hat{k} \mathbin{\hat{\sqcup}} \hat{k}' = \Big(pp \mapsto \mathtt{max}(\hat{k}(pp), \hat{k}'(pp))\Big)^*$$

$$\mathsf{hlift}(\hat{h}; \hat{k}) = \left(pp \mapsto \begin{cases} \{\!|c; (f \mapsto \mathsf{lift}(\hat{u}; \hat{k}))^*|\!\} & \text{if } \hat{k}(pp) = 0 \wedge \hat{h}(pp) = \{\!|c; (f \mapsto \hat{u})^*|\!\} \\ \{\!|@c; \mathsf{lift}(\hat{u}; \hat{k})|\!\} & \text{if } \hat{k}(pp) = 0 \wedge \hat{h}(pp) = \{\!|@c; \hat{u}|\!\} \\ \tau[\mathsf{lift}(\hat{u}; \hat{k})] & \text{if } \hat{k}(pp) = 0 \wedge \hat{h}(pp) = \tau[\hat{u}] \\ \bot & \text{otherwise} \end{cases} \right)^*$$

Right-Hand Side

We can now present the rules for the abstract evaluation of right-hand sides (a formal description is given in Table 5.11): to abstract a primitive value *prim* at a program point pp, we take the corresponding element $\widehat{prim}$ from the underlying abstract domain. To abstract the content of a register r_i at program point pp, we take the abstract local state fact $\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \_; \_)$ and we return the i-th abstract value $\hat{v}_i$. To abstract, at program point pp, the content of the field f of an object whose location is stored in register r_i, we retrieve the i-th abstract value $\hat{v}_i$ from the abstract fact $\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \_)$: if $\hat{v}_i$ contains any location abstraction $\hat{\lambda}$, we look whether it is an abstract flow-sensitive location $\mathsf{FS}(\lambda)$ or an abstract flow-insensitive location $\mathsf{NFS}(\lambda)$: in the former case, we get the entry $(\lambda \mapsto \hat{o})$ from the abstract flow-sensitive heap $\hat{h}$, and we return the abstract value stored in the field f of the abstract object $\hat{o}$; in the latter case, we try to find a matching flow-insensitive heap fact $\mathsf{H}(\lambda, \hat{o})$ and we return the *lifted* value of the field f of the abstract object $\hat{o}$ contained therein. We similarly abstract the content of array cells, but in a field-insensitive fashion. To abstract the content of a static field $\mathsf{c.f}$ at program point pp, we take any fact $\mathsf{S}_{\mathsf{c,f}}(\hat{v})$ and we return the *lifted* abstract value $\hat{v}$.

Remark 1. When getting an abstract value from a flow-insensitive heap fact, a static field fact or an array we lift it, by returning $\mathsf{lift}(\hat{v}; 1^*)$ [1]. This is due to the fact that, by definition, a flow-insensitive memory block cannot contain a location to a flow-sensitive memory block. Therefore we chose that instead of lifting abstract locations before putting them in abstract flow-insensitive facts, arrays or static fields, we lift abstract locations when performing look-ups. We believe this to (slightly) simplify the abstract semantics and the soundness proof.

[1]We abuse the notation here: 1^* should be interpreted as $(\_ \mapsto 1)^*$.

$$\langle\!\langle prim \rangle\!\rangle_{pp} = \{\mathsf{RHS}_{\mathsf{pp}}(\widehat{prim})\} \qquad \langle\!\langle r_i \rangle\!\rangle_{pp} = \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \_; \_) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{v}_i)\}$$

$$\langle\!\langle c.f \rangle\!\rangle_{pp} = \{\mathsf{S}_{\mathsf{c,f}}(\hat{v}) \implies \mathsf{RHS}_{\mathsf{pp}}(\mathsf{lift}(\hat{v}; 1^*))\}$$

$$\langle\!\langle r_i.f \rangle\!\rangle_{pp} = \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \_) \land \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{NFS}(\lambda); \{\!|c; (f' \mapsto \hat{v}')^*, f \mapsto \hat{u}|\!\}) \implies$$
$$\mathsf{RHS}_{\mathsf{pp}}(\mathsf{lift}(\hat{u}; 1^*))\}$$

$$\cup\, \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \_) \land \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{FS}(\lambda); \{\!|c; (f' \mapsto \hat{v}')^*, f \mapsto \hat{u}|\!\}) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{u})\}$$

$$\langle\!\langle r_i[r_j] \rangle\!\rangle_{pp} = \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \_) \land \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{NFS}(\lambda); \tau[\hat{u}]) \implies \mathsf{RHS}_{\mathsf{pp}}(\mathsf{lift}(\hat{u}; 1^*))\}$$

$$\cup\, \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \_) \land \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{FS}(\lambda); \tau[\hat{u}]) \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{u})\}$$

Table 5.11: Abstract Evaluation of Right-hand Sides

Activity Abstraction

We will now describe the rules abstracting the activity lifecycle and thread management mechanisms, which are given in Table 5.12. The rule (TSTART) over-approximates the spawning of a new thread $\mathsf{T}(\lambda, \{\!|c; (f \mapsto \_)^*|\!\})$ by generating an abstract local state running the method run of the corresponding thread object. The rule (CBK) abstracts the callback invocation by generating an abstract local heap fact for all the callbacks of a started activity. Observe that the initial arguments supplied are over-approximated by $\top$, since they depend on user-inputs and are not statistically known. The rule (FIN) roughly over-approximates whether an activity is finished or not: it always replaces the $\mathsf{finished}$ field of an activity object by $\top_{\texttt{bool}}$. The rule (REP) restarts abstract activity objects at any time, by re-setting their fields to their default initial abstract value $\hat{\mathbf{0}}_\tau$ (this over-approximates the restarting of an activity when the screen orientation changes). The rule (ACT) handles the starting of new activities: if an intent $\mathsf{I}_{\mathsf{c'}}(\{\!|@in(c); \hat{v}^*|\!\})$ has been sent to an activity c by an activity c', the rule creates a new abstract activity object of class c with properly bound and initialized fields. It also creates a new special abstract heap fact $\mathsf{H}(in(c), \{\!|@c; \hat{v}^*|\!\})$ that contains a copy of the sent intent: this over-approximates the serialization mechanism, and is sound because the intent contains only abstract flow-insensitive locations, that are updated with weak updates. The rule (RES) over-approximates the mechanism by which a child activity returns a result to its parent activity. Finally rule (SUB) contains subtyping judgments for classes, and rule (PO) contain partial ordering rules for abstract values.

Statement Abstraction

Before giving the abstract rule for Dalvik statements, we need to define the abstract counter-part of the $type_H(b)$ function:

$$
\begin{array}{lcl}
\mathit{Tstart} & = & \{\mathsf{T}(\lambda, \{\!|c; (f \mapsto \_)^* |\!\}) \wedge c \leq \mathsf{c}' \wedge c \leq \mathsf{Thread} \\
 & & \implies \mathsf{LState}_{\mathsf{c}',\mathsf{run},0}((\mathsf{NFS}(\lambda), \mathsf{NFS}(\lambda)); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(\lambda); (\bot)^*; 0^*) \mid \\
 & & \mathsf{c}' \in \widehat{lookup}(\mathsf{run}) \wedge \mathit{sign}(\mathsf{c}', \mathsf{run}) = \mathsf{Thread} \xrightarrow{loc} \mathsf{Void}\} \\
\mathit{Cbk} & = & \{\mathsf{H}(c, \{\!|c; (f \mapsto \_)^* |\!\}) \wedge c \leq \mathsf{c}' \implies \\
 & & \mathsf{LState}_{\mathsf{c}',\mathsf{m},0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*) \mid \\
 & & \mathsf{c}' \text{ is an activity class} \wedge \exists s : m \in cb(\mathsf{c}', s) \wedge \mathit{sign}(\mathsf{c}', m) = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau\} \\
\mathit{Fin} & = & \{\mathsf{H}(c, \{\!|c; (f \mapsto \_)^*, \mathsf{finished} \mapsto \_ |\!\}) \implies \mathsf{H}(c, \{\!|c; (f \mapsto \_)^*, \\
 & & \mathsf{finished} \mapsto \top_{\texttt{bool}} |\!\})\} \\
\mathit{Rep} & = & \{\mathsf{H}(c, \{\!|c; (f_\tau \mapsto \_)^* |\!\}) \implies \mathsf{H}(c, \{\!|c; (f_\tau \mapsto \hat{\mathbf{0}}_\tau)^* |\!\})\} \\
\mathit{Act} & = & \{\mathsf{I}_{\mathsf{c}'}(\{\!|@c; \hat{v} |\!\})) \implies \mathsf{H}(in(c), \{\!|@c; \hat{v} |\!\})\} \cup \\
 & & \{\mathsf{I}_{\mathsf{c}'}(\{\!|@c; \hat{v} |\!\})) \implies \mathsf{H}(c, \{\!|c; (f_\tau \mapsto \hat{\mathbf{0}}_\tau)^*, \\
 & & \mathsf{finished} \mapsto \widehat{\texttt{false}}, \mathit{parent} \mapsto c', \mathsf{intent} \mapsto in(c) |\!\})\} \\
\mathit{Res} & = & \{\mathsf{H}(c', \{\!|c'; (f' \mapsto \_)^*, \mathit{parent} \mapsto c, \mathsf{result} \mapsto \hat{w} |\!\} \wedge \mathsf{H}(c, \{\!|c; (f \mapsto \_)^*, \\
 & & \mathsf{result} \mapsto \_ |\!\} \implies \mathsf{H}(c, \{\!|c; (f \mapsto \_)^*, \mathsf{result} \mapsto \hat{w} |\!\}\} \\
\mathit{Sub} & = & \{\tau \leq \tau' \mid \tau \leq \tau' \text{ is a valid subtyping judgment}\} \\
\mathit{Po} & = & \{\hat{v} \sqsubseteq \hat{v}' \mid \hat{v} \sqsubseteq \hat{v}' \text{ is a valid partial ordering}\}
\end{array}
$$

Table 5.12: Abstract Semantics of μ-Dalvik$_A$ - Activity Rules

Definition 17. Given an abstract memory block $\hat{b}$, we define a function $\widehat{get\text{-}type}(\hat{b})$ as follows:

$$
\widehat{get\text{-}type}(\hat{b}) = \begin{cases} c & \text{if } \hat{b} = \{\!|c; (f \mapsto \hat{v})^* |\!\} \\ \texttt{array}[\tau] & \text{if } \hat{b} = \tau[\hat{v}] \\ \texttt{Intent} & \text{if } \hat{b} = \{\!|@c; \hat{v} |\!\} \end{cases}
$$

For all standard Dalvik statement *st* and program point *pp*, the rule $(\!|st|\!)_{pp}$ abstracts the action of *st* at program point *pp*. The most important rules have already been described in § 4.4, and the full set of rules is given in Table 5.13, Table 5.14, Table 5.15, Table 5.16, and Table 5.17. A few points are worth mentioning:

- $(\!|\texttt{wait } r_i|\!)_{pp}$: We just check whether the inte field of the abstract object over-approximating the running thread or activity is over-approximating $\widehat{\texttt{true}}$, in which case an abstract abnormal local state throwing an $\mathsf{IntExcpt}$ is generated, or $\widehat{\texttt{false}}$, in which case the abstract local state is propagated to the next program point;

- $(\!|\texttt{monitor-enter } r_i|\!)_{pp}$ and $(\!|\texttt{monitor-exit } r_i|\!)_{pp}$: Given that monitors are synchronization constructs, it is sound to ignore them when checking reachability properties, which is the target of the present work. There are of course more precise ways of abstracting monitors, but they would make the analysis more complicated and their practical benefits are unclear.

- $(\!|\texttt{start-act}\ r_i|\!)_{pp}$: When an abstract intent $\{\!|@c'; \hat{u}|\!\}$ stored in the flow-sensitive heap at program point $\hat{\lambda}$ is used to start a new (abstract) activity, every abstract flow-sensitive location reachable from $\hat{\lambda}$ in $\hat{h}$ (represented by the abstract filter $\hat{k}'$ computed by $\mathsf{Reach}(\mathsf{FS}(\lambda); \hat{h}; \hat{k}'))$ is being lifted, to make sure that these heap entries are abstract in a flow-insensitive fashion, since they are being shared between the parent and the started child activity.

$$\begin{array}{lcl}
(\!|\texttt{goto}\ pc'|\!)_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_; \hat{v}^*; _; _) \implies \mathsf{LState}_{\mathsf{c,m,pc'}}(_; \hat{v}^*; _; _)\} \\
(\!|\texttt{if}_{\otimes}\ r_i\ r_j\ \texttt{then}\ pc'|\!)_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_; \hat{v}^*; _; _) \wedge \hat{v}_i\ \hat{\otimes}\ \hat{v}_j \implies \\
 & & \mathsf{LState}_{\mathsf{c,m,pc'}}(_; \hat{v}^*; _; _)\} \cup \\
 & & \{\mathsf{LState}_{\mathsf{pp}}(_; \hat{v}^*; _; _) \wedge \hat{v}_i\ \hat{\not\otimes}\ \hat{v}_j \implies \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_; \hat{v}^*; _; _)\} \\
(\!|\texttt{binop}_{\oplus}\ r_d\ r_i\ r_j|\!)_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_; \hat{v}^*; _; _) \implies \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_; \hat{v}^*[d \mapsto \hat{v}_i\ \hat{\oplus}\ \hat{v}_j]; _; _)\} \\
(\!|\texttt{unop}_{\odot}\ r_d\ r_i|\!)_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_; \hat{v}^*; _; _) \implies \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_; \hat{v}^*[d \mapsto \hat{\odot}\, \hat{v}_i]; _; _)\} \\
(\!|\texttt{move}\ r_d\ rhs|\!)_{pp} & = & \langle\!\langle rhs \rangle\!\rangle_{pp} \cup \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}') \wedge \mathsf{LState}_{\mathsf{pp}}(_; \hat{v}^*; _; _) \implies \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_; \hat{v}^*[d \mapsto \hat{v}']; _; _)\} \\
(\!|\texttt{move}\ r_a[r_{idx}]\ rhs|\!)_{pp} & = & \langle\!\langle rhs \rangle\!\rangle_{pp} \cup \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{LState}_{\mathsf{pp}}(_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \\
 & & \mathsf{GetBlk}_a(\hat{v}^*; \hat{h}; \mathsf{NFS}(\lambda); \tau[\hat{v}']) \wedge \mathsf{Reach}(\hat{v}''; \hat{h}; \hat{k}') \\
 & & \implies \mathsf{H}(\lambda, \tau[\hat{v}' \sqcup \hat{v}'']) \wedge \mathsf{LiftHeap}(\hat{h}; \hat{k}') \wedge \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_; \mathsf{lift}(\hat{v}^*; \hat{k}'); \mathsf{hlift}(\hat{h}; \hat{k}'); \hat{k}\ \hat{\sqcup}\ \hat{k}')\} \cup \\
 & & \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{LState}_{\mathsf{pp}}(_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \\
 & & \mathsf{GetBlk}_a(\hat{v}^*; \hat{h}; \mathsf{FS}(\lambda); \tau[\hat{v}']) \\
 & & \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}(_; \hat{v}^*; \hat{h}[\lambda \mapsto \tau[\hat{v}' \sqcup \hat{v}'']; \hat{k})\} \\
(\!|\texttt{move}\ r_o.f\ rhs|\!)_{pp} & = & \langle\!\langle rhs \rangle\!\rangle_{pp} \cup \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{LState}_{\mathsf{pp}}(_; \hat{v}^*; \hat{h}; \hat{k}) \\
 & & \wedge \mathsf{GetBlk}_o(\hat{v}^*; \hat{h}; \mathsf{NFS}(\lambda); \{\!|c'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'|\!\}) \wedge \\
 & & \mathsf{Reach}(\hat{v}''; \hat{h}; \hat{k}') \implies \\
 & & \mathsf{H}(\lambda, \{\!|\mathsf{c}'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'')|\!\}) \wedge \mathsf{LiftHeap}(\hat{h}; \hat{k}') \wedge \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_; \mathsf{lift}(\hat{v}^*; \hat{k}'); \mathsf{hlift}(\hat{h}; \hat{k}'); \hat{k}\ \hat{\sqcup}\ \hat{k}')\} \cup \\
 & & \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{LState}_{\mathsf{pp}}(_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \\
 & & \mathsf{GetBlk}_o(\hat{v}^*; \hat{h}; \mathsf{FS}(\lambda); \{\!|c'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'|\!\}) \implies \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_; \hat{v}^*; \hat{h}[\lambda \mapsto \{\!|c'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}''|\!\}; \hat{k})\} \\
(\!|\texttt{move}\ c'.f\ rhs|\!)_{pp} & = & \langle\!\langle rhs \rangle\!\rangle_{pp} \cup \{\mathsf{RHS}_{\mathsf{pp}}(\hat{v}') \wedge \mathsf{LState}_{\mathsf{pp}}(_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \\
 & & \mathsf{Reach}(\hat{v}'; \hat{h}; \hat{k}') \implies \mathsf{S}_{\mathsf{c',f}}(\hat{v}') \wedge \mathsf{LiftHeap}(\hat{h}; \hat{k}') \wedge \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_; \mathsf{lift}(\hat{v}^*; \hat{k}'); \mathsf{hlift}(\hat{h}; \hat{k}'); \hat{k}\ \hat{\sqcup}\ \hat{k}')\}
\end{array}$$

Conventions: $\mathsf{pp} = \mathsf{c}, \mathsf{m}, \mathsf{pc}$

Table 5.13: Abstract Semantics of μ-Dalvik$_A$ - Standard Statements (continues in Table 5.14)

$$
\begin{array}{lcl}
\llparenthesis \texttt{instof}\ r_d\ r_s\ \tau \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_;\hat{v}^*;\hat{h};\hat{k}) \wedge \\
 & & \mathsf{GetBlk}_s(\hat{v}^*;\hat{h};_;\hat{b}) \wedge \widehat{\textit{get-type}}(\hat{b}) \leq \tau \\
 & & \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}(_;\hat{v}^*[d \mapsto \widehat{\texttt{true}}];\hat{h};\hat{k})\} \cup \\
 & & \{\mathsf{LState}_{\mathsf{pp}}(_;\hat{v}^*;\hat{h};\hat{k}) \wedge \\
 & & \mathsf{GetBlk}_s(\hat{v}^*;\hat{h};_;\hat{b}) \wedge \widehat{\textit{get-type}}(\hat{b}) \not\leq \tau \\
 & & \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}(_;\hat{v}^*[d \mapsto \widehat{\texttt{false}}];\hat{h};\hat{k})\} \\
\llparenthesis \texttt{checkcast}\ r_s\ \tau \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_;\hat{v}^*;\hat{h};\hat{k}) \wedge \\
 & & \mathsf{GetBlk}_s(\hat{v}^*;\hat{h};_;\hat{b}) \wedge \widehat{\textit{get-type}}(\hat{b}) \leq \tau \implies \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_;\hat{v}^*;\hat{h};\hat{k})\} \\
\llparenthesis \texttt{new}\ r_d\ \mathsf{c}' \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{Reach}(\mathsf{FS}(\mathsf{pp});\hat{h};\hat{k}') \implies \\
 & & \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{\mathsf{c,m,pc+1}}(_;\mathsf{lift}(\hat{v}^*;\hat{k}')[d \mapsto \mathsf{FS}(\mathsf{pp})]; \\
 & & \mathsf{hlift}(\hat{h};\hat{k}')[\mathsf{pp} \mapsto \{\!|\mathsf{c}';(f \mapsto \hat{\mathbf{0}}_\tau)^*|\!\}];\hat{k} \mathbin{\hat{\sqcup}} \hat{k}')\} \\
\llparenthesis \texttt{newintent}\ r_d\ \mathsf{c}' \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{Reach}(\mathsf{FS}(\mathsf{pp});\hat{h};\hat{k}') \\
 & & \implies \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{\mathsf{c,m,pc+1}}(_;\mathsf{lift}(\hat{v}^*;\hat{k}')[d \mapsto \mathsf{FS}(\mathsf{pp})]; \\
 & & \mathsf{hlift}(\hat{h};\hat{k}')[\mathsf{pp} \mapsto \{\!|@\mathsf{c}';\bot|\!\})];\hat{k} \mathbin{\hat{\sqcup}} \hat{k}')\} \\
\llparenthesis \texttt{newarray}\ r_d\ r_l\ \tau \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{Reach}(\mathsf{FS}(\mathsf{pp});\hat{h};\hat{k}') \\
 & & \implies \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{\mathsf{c,m,pc+1}}(_;\mathsf{lift}(\hat{v}^*;\hat{k}')[d \mapsto \mathsf{FS}(\mathsf{pp})]; \\
 & & \mathsf{hlift}(\hat{h};\hat{k}')[\mathsf{pp} \mapsto \tau[\hat{\mathbf{0}}_\tau])];\hat{k} \mathbin{\hat{\sqcup}} \hat{k}')\} \\
\llparenthesis \texttt{start-act}\ r_i \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{NFS}(\lambda);\{\!|@c';\hat{u}|\!\}) \\
 & & \implies \mathsf{I}_{\mathsf{c}}(\{\!|@c';\hat{u}|\!\}) \wedge \mathsf{LState}_{\mathsf{c,m,pc+1}}(_;\hat{v}^*;\hat{h};\hat{k})\} \cup \\
 & & \{\mathsf{LState}_{\mathsf{pp}}(_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{FS}(\lambda);\{\!|@c';\hat{u}|\!\}) \wedge \\
 & & \mathsf{Reach}(\mathsf{FS}(\lambda);\hat{h};\hat{k}') \implies \mathsf{I}_{\mathsf{c}}(\{\!|@c';\hat{u}|\!\}) \wedge \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_;\mathsf{lift}(\hat{v}^*;\hat{k}');\mathsf{hlift}(\hat{h};\hat{k}');\hat{k} \mathbin{\hat{\sqcup}} \hat{k}')\} \\
\llparenthesis \texttt{put-extra}\ r_i\ r_k\ r_j \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{NFS}(\lambda);\{\!|@c';\hat{v}'|\!\}) \\
 & & \wedge \mathsf{Reach}(\hat{v}_j;\hat{h};\hat{k}') \implies \mathsf{H}(\lambda,\{\!|@c';\hat{v}' \sqcup \hat{v}_j|\!\}) \wedge \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_;\mathsf{lift}(\hat{v}^*;\hat{k}');\mathsf{hlift}(\hat{h};\hat{k}');\hat{k} \mathbin{\hat{\sqcup}} \hat{k}')\} \cup \\
 & & \{\mathsf{LState}_{\mathsf{pp}}(_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{FS}(\lambda);\{\!|@c';\hat{v}'|\!\}) \\
 & & \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}(_;\hat{v}^*;\hat{h}[\lambda \mapsto \{\!|@c';\hat{v}' \sqcup \hat{v}_j|\!\}];\hat{k})\} \\
\llparenthesis \texttt{get-extra}\ r_i\ r_k\ \tau \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};_;\{\!|@c';\hat{v}'|\!\}) \implies \\
 & & \mathsf{LState}_{\mathsf{c,m,pc+1}}(_;\hat{v}^*[\mathsf{res} \mapsto \hat{v}'];\hat{h};\hat{k})\} \\
\llparenthesis \texttt{return} \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}_t,\hat{v}^*_{call});\hat{v}^*;\hat{h};\hat{k}) \implies \mathsf{Res}_{\mathsf{c,m}}((\hat{\lambda}_t,\hat{v}^*_{call});\hat{v}_{\mathsf{res}};\hat{h};\hat{k})\}
\end{array}
$$

Conventions: $\mathsf{pp} = \mathsf{c},\mathsf{m},\mathsf{pc}$

Table 5.14: Abstract Semantics of μ-Dalvik$_A$ - Standard Statements (continuation of Table 5.13)

- $\llparenthesis \texttt{invoke}\ r_o\ m'\ (r_{i_j})^{j \leq n} \rrparenthesis_{pp} =$
  $\{\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}_t, \_); \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*; \hat{h}; \_; \{\!|c'; (f \mapsto \hat{u})^*|\!\}) \wedge c' \leq \mathsf{c''}$
  $\implies \mathsf{LState}_{\mathsf{c''},\mathsf{m'},0}((\hat{\lambda}_t, (\hat{v}_{i_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, (\hat{v}_{i_j})^{j \leq n}; \hat{h}; 0^*) \mid \mathsf{c''} \in \widehat{lookup}(m') \wedge sign(\mathsf{c''}, m') = (\tau_j)^{j \leq n} \xrightarrow{loc} \tau\} \cup$
  $\{\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}_t, \_); \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*; \hat{h}; \_; \{\!|c'; (f \mapsto \hat{u})^*|\!\}) \wedge c' \leq \mathsf{c''} \wedge \mathsf{Res}_{\mathsf{c''},\mathsf{m'}}((\hat{\lambda}'_t, \hat{w}^*); \hat{v}'_{\mathsf{res}}; \hat{h}_{\mathsf{res}}; \hat{k}_{\mathsf{res}})$
  $\wedge \hat{\lambda}_t = \hat{\lambda}'_t \wedge \left(\bigwedge_{j \leq n} \hat{v}_{i_j} \sqcap \hat{w}_j \not\sqsubseteq \bot\right) \implies \mathsf{LState}_{\mathsf{c},\mathsf{m},\mathsf{pc}+1}((\hat{\lambda}_t, \_); \mathsf{lift}(\hat{v}^*; \hat{k}_{\mathsf{res}})[\mathsf{res} \mapsto \hat{v}'_{\mathsf{res}}]; \hat{h}_{\mathsf{res}}; \hat{k} \mathbin{\hat{\sqcup}} \hat{k}_{\mathsf{res}}) \mid \mathsf{c''} \in \widehat{lookup}(\mathsf{m'})\}$
  $\{\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}_t, \_); \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_o(\hat{v}^*; \hat{h}; \_; \{\!|c'; (f \mapsto \hat{u})^*|\!\}) \wedge c' \leq \mathsf{c''} \wedge \mathsf{Uncaught}_{\mathsf{c''},\mathsf{m'}}((\hat{\lambda}'_t, \hat{w}^*)); \hat{v}'_{\mathsf{excpt}}; \hat{h}_{\mathsf{res}}; \hat{k}_{\mathsf{res}})$
  $\wedge \hat{\lambda}_t = \hat{\lambda}'_t \wedge \left(\bigwedge_{j \leq n} \hat{v}_{i_j} \sqcap \hat{w}_j \not\sqsubseteq \bot\right) \implies \mathsf{AState}_{\mathsf{c},\mathsf{m},\mathsf{pc}}((\hat{\lambda}_t, \_); \mathsf{lift}(\hat{v}^*; \hat{k}_{\mathsf{res}})[\mathsf{excpt} \mapsto \hat{v}'_{\mathsf{excpt}}]; \hat{h}_{\mathsf{res}}; \hat{k} \mathbin{\hat{\sqcup}} \hat{k}_{\mathsf{res}}) \mid \mathsf{c''} \in \widehat{lookup}(\mathsf{m'})\}$
- $\llparenthesis \texttt{sinvoke}\ c'\ m'\ (r_{i_j})^{j \leq n} \rrparenthesis_{pp} =$
  $\{\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}_t, \_); \hat{v}^*; \hat{h}; \hat{k}) \implies \mathsf{LState}_{\mathsf{c'},\mathsf{m'},0}((\hat{\lambda}_t, (\hat{v}_{i_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, (\hat{v}_{i_j})^{j \leq n}; \hat{h}; 0^*) \mid sign(\mathsf{c'}, m') = (\tau_j)^{j \leq n} \xrightarrow{loc} \tau\} \cup$
  $\{\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}_t, \_); \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{Res}_{\mathsf{c'},\mathsf{m'}}((\hat{\lambda}'_t, \hat{w}^*); \hat{v}'_{\mathsf{res}}; \hat{h}_{\mathsf{res}}; \hat{k}_{\mathsf{res}}) \wedge \hat{\lambda}_t = \hat{\lambda}'_t \wedge \left(\bigwedge_{j \leq n} \hat{v}_{i_j} \sqcap \hat{w}_j \not\sqsubseteq \bot\right)$
  $\implies \mathsf{LState}_{\mathsf{c},\mathsf{m},\mathsf{pc}+1}((\hat{\lambda}_t, \_); \mathsf{lift}(\hat{v}^*; \hat{k}_{\mathsf{res}})[\mathsf{res} \mapsto \hat{v}'_{\mathsf{res}}]; \hat{h}_{\mathsf{res}}; \hat{k} \mathbin{\hat{\sqcup}} \hat{k}_{\mathsf{res}})\}$
  $\{\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}_t, \_); \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{Uncaught}_{\mathsf{c'},\mathsf{m'}}((\hat{\lambda}'_t, \hat{w}^*); \hat{v}'_{\mathsf{excpt}}; \hat{h}_{\mathsf{res}}; \hat{k}_{\mathsf{res}}) \wedge \hat{\lambda}_t = \hat{\lambda}'_t \wedge \left(\bigwedge_{j \leq n} \hat{v}_{i_j} \sqcap \hat{w}_j \not\sqsubseteq \bot\right)$
  $\implies \mathsf{AState}_{\mathsf{c},\mathsf{m},\mathsf{pc}}((\hat{\lambda}_t, \_); \mathsf{lift}(\hat{v}^*; \hat{k}_{\mathsf{res}})[\mathsf{excpt} \mapsto \hat{v}'_{\mathsf{excpt}}]; \hat{h}_{\mathsf{res}}; \hat{k} \mathbin{\hat{\sqcup}} \hat{k}_{\mathsf{res}})\}$

Conventions: $\mathsf{pp} = \mathsf{c}, \mathsf{m}, \mathsf{pc}$

Table 5.15: Abstract Semantics of μ-Dalvik$_A$ - Invoke Statements

Statement Abstractions:

$$
\begin{array}{lcl}
\llparenthesis \texttt{start-thread}\ r_i \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{NFS}(\lambda); \{\!|c'; (f \mapsto \hat{u})^*|\!\})\wedge \\
& & c' \leq \mathsf{Thread} \\
& & \implies \mathsf{T}(\lambda, \{\!|c'; (f \mapsto \hat{u})^*|\!\}) \wedge \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_; \hat{v}^*; \hat{h}; \hat{k})\} \cup \\
& & \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{FS}(\lambda); \{\!|c'; (f \mapsto \hat{u})^*|\!\})\wedge \\
& & c' \leq \mathsf{Thread} \wedge \mathsf{Reach}(\mathsf{FS}(\lambda); \hat{h}; \hat{k}') \\
& & \implies \mathsf{T}(\lambda, \{\!|c'; (f \mapsto \hat{u})^*|\!\}) \wedge \mathsf{LiftHeap}(\hat{h}; \hat{k}')\wedge \\
& & \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_; \mathsf{lift}(\hat{v}^*; \hat{k}'); \mathsf{hlift}(\hat{h}; \hat{k}'); \hat{k} \mathbin{\hat{\sqcup}} \hat{k}')\} \\
\llparenthesis \texttt{interrupt}\ r_i \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \hat{k})\wedge \\
& & \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{NFS}(\lambda); \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \_|\!\}) \\
& & \implies \mathsf{H}(\lambda, \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \widehat{\texttt{true}}|\!\}\wedge \\
& & \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_; \hat{v}^*; \hat{h}; \hat{k})\} \cup \\
& & \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \hat{k})\wedge \\
& & \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{FS}(\lambda); \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \_|\!\}) \implies \\
& & \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_; \hat{v}^*; \hat{h}[\lambda \mapsto \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \widehat{\texttt{true}}|\!\}]; \hat{k})\} \\
\llparenthesis \texttt{interrupted}\ r_i \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \hat{k})\wedge \\
& & \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{NFS}(\lambda); \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \hat{v}'|\!\}) \\
& & \implies \mathsf{H}(\lambda, \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \widehat{\texttt{false}}|\!\}\wedge \\
& & \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_; \hat{v}^*[\mathsf{res} \mapsto \hat{v}']; \hat{h}; \hat{k})\}\cup \\
& & \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \hat{k})\wedge \\
& & \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{FS}(\lambda); \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \hat{v}'|\!\}) \\
& & \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_; \hat{v}^*[\mathsf{res} \mapsto \hat{v}']; \hat{h}[\lambda \mapsto \{\!|c'; (f \mapsto \hat{u})^*, \\
& & \mathsf{inte} \mapsto \widehat{\texttt{false}}|\!\}]; \hat{k})\} \\
\llparenthesis \texttt{is-interrupted}\ r_i \rrparenthesis_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(\_; \hat{v}^*; \hat{h}; \hat{k})\wedge \\
& & \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \_; \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \hat{v}'|\!\}) \\
& & \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_; \hat{v}^*[\mathsf{res} \mapsto \hat{v}']; \hat{h}; \hat{k})\}
\end{array}
$$

Conventions: $\mathsf{pp} = \mathsf{c}, \mathsf{m}, \mathsf{pc}$

Table 5.16: Abstract Semantics of μ-Dalvik$_A$ - Rules for New Statements (continues in Table 5.17)

Statement Abstractions:

$$
\begin{array}{lcl}
(\!|\texttt{join } r_i|\!)_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k})\wedge \\
& & \mathsf{H}(\lambda_t,\{\!|c';(f\mapsto\hat{u})^*,\mathsf{inte}\mapsto\hat{v}'|\!\})\wedge\widehat{\texttt{false}}\sqsubseteq\hat{v}' \\
& & \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k})\}\cup \\
& & \{\mathsf{LState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k})\wedge \\
& & \mathsf{H}(\lambda_t,\{\!|c';(f\mapsto\hat{u})^*,\mathsf{inte}\mapsto\hat{v}'|\!\})\wedge\widehat{\texttt{true}}\sqsubseteq\hat{v}' \implies \\
& & \mathsf{H}(\mathsf{pp};\{\!|\mathsf{IntExcpt};|\!\})\wedge \\
& & \mathsf{AState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*[\mathsf{excpt}\mapsto\mathsf{NFS}(\mathsf{pp})];\hat{h};\hat{k})\wedge \\
& & \mathsf{H}(\lambda_t,\{\!|c';(f\mapsto\hat{u})^*,\mathsf{inte}\mapsto\widehat{\texttt{false}}|\!\})\} \\
(\!|\texttt{wait } r_i|\!)_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k})\wedge \\
& & \mathsf{H}(\lambda_t,\{\!|c';(f\mapsto\hat{u})^*,\mathsf{inte}\mapsto\hat{v}'|\!\})\wedge\widehat{\texttt{false}}\sqsubseteq\hat{v}' \\
& & \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k})\}\cup \\
& & \{\mathsf{LState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*;\hat{h};\hat{k})\wedge \\
& & \mathsf{H}(\lambda_t,\{\!|c';(f\mapsto\hat{u})^*,\mathsf{inte}\mapsto\hat{v}'|\!\})\wedge\widehat{\texttt{true}}\sqsubseteq\hat{v}' \implies \\
& & \mathsf{H}(\mathsf{pp};\{\!|\mathsf{IntExcpt};|\!\})\wedge \\
& & \mathsf{AState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_t),\_);\hat{v}^*[\mathsf{excpt}\mapsto\mathsf{NFS}(\mathsf{pp})];\hat{h};\hat{k})\wedge \\
& & \mathsf{H}(\lambda_t,\{\!|c';(f\mapsto\hat{u})^*,\mathsf{inte}\mapsto\widehat{\texttt{false}}|\!\})\} \\
(\!|\texttt{monitor-enter } r_i|\!)_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(\_;\hat{v}^*;\hat{h};\hat{k}) \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*;\hat{h};\hat{k})\} \\
(\!|\texttt{monitor-exit } r_i|\!)_{pp} & = & \{\mathsf{LState}_{\mathsf{pp}}(\_;\hat{v}^*;\hat{h};\hat{k}) \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*;\hat{h};\hat{k})\} \\
(\!|\texttt{throw } r_i|\!)_{pp} & = & \{\mathsf{LState}_{\mathsf{c,m,pc}}(\_;\hat{v}^*;\hat{h};\hat{k}) \\
& & \implies \mathsf{AState}_{\mathsf{c,m,pc'}}(\_;\hat{v}^*[\mathsf{excpt}\mapsto\hat{v}_i];\hat{h};\hat{k})\} \\
(\!|\texttt{move-except } r_d|\!)_{pp} & = & \{\mathsf{LState}_{\mathsf{c,m,pc}}(\_;\hat{v}^*;\hat{h};\hat{k}) \\
& & \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}(\_;\hat{v}^*[d\mapsto\hat{v}_{\mathsf{excpt}}];\hat{h};\hat{k})\}
\end{array}
$$

Global Abstractions:

$$
\begin{array}{lcl}
\mathit{AbState} & = & \{\mathsf{AState}_{\mathsf{c,m,pc}}(\_;\hat{v}^*;\hat{h};\hat{k})\wedge\mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}^*;\hat{h};\_;\{\!|\mathsf{c}';\_|\!\})\wedge\mathsf{c}'\leq\mathsf{Throwable} \\
& & \implies \mathsf{LState}_{\mathsf{c,m,pc'}}(\_;\hat{v}^*;\hat{h};\hat{k}) \mid \mathsf{ExcptTable}(\mathsf{c},\mathsf{m},\mathsf{pc},\mathsf{c}')=\mathsf{pc}'\} \\
& & \{\mathsf{AState}_{\mathsf{c,m,pc}}(\_;\hat{v}^*;\hat{h};\hat{k})\wedge\mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}^*;\hat{h};\_;\{\!|\mathsf{c}';\_|\!\})\wedge\mathsf{c}'\leq\mathsf{Throwable} \\
& & \implies \mathsf{Uncaught}_{\mathsf{c,m}}(\_;\hat{v}_{\mathsf{excpt}};\hat{h};\hat{k}) \mid \mathsf{ExcptTable}(\mathsf{c},\mathsf{m},\mathsf{pc},\mathsf{c}')=\bot\}
\end{array}
$$

Conventions: $\mathsf{pp}=\mathsf{c},\mathsf{m},\mathsf{pc}$

Table 5.17: Abstract Semantics of μ-Dalvik$_A$ - Rules for New Statements (continuation of Table 5.16)

5.3 Proofs

Before entering in the formalism, we are going to give an informal description of the difficulties. The main problem is that knowing which locations are going to be abstracted as abstract flow-sensitive locations and which locations are going to be abstracted as abstract flow-insensitive locations is *dynamically* determined by the analysis: this is not a property of the concrete semantics that is abstracted. That is, given a snapshot of an execution (a configuration Ψ), there is no *unique* correct way of choosing which locations should be handled in a flow-sensitive fashion, since the information about which are the most-recently allocated locations is not stored in Ψ. Therefore there are several ways of abstracting a configuration: there is one possible abstraction of a configuration for each decomposition of the set of locations into locations that are handled in a flow-sensitive fashion and location that are handled in a flow-insensitive fashion, and for each *history* of the heap. An *history* is a record of which locations used to be abstracted as abstract flow-sensitive locations, and when they were lifted. To see why it is necessary to take into account the history, consider the following example.

Example 1. Consider the following call-stack: $\alpha = \langle c, m, pc \cdot R \cdot st^* \cdot u \rangle :: \langle c', m', pc' \cdot R' \cdot st'^* \cdot \_ \rangle$ with $R = (r_1 \mapsto p_{\mathsf{pp}}, r_2 \mapsto p'_{\mathsf{pp}})$, $u = p_{\mathsf{pp}}$ and $R' = (r \mapsto p_{\mathsf{pp}})$.

Here there are several possible abstractions of this call-stack: for example, p_{pp} could have been lifted before c', m' invoked c, m, and c, m could have just allocated a new object at location p'_{pp}, in which case p_{pp} is abstracted in a flow-insensitive fashion in both c, m and c', m'.

But another possibility is that, when c', m' invoked c, m, the location p_{pp} was abstracted in a flow-sensitive fashion. Then later on c, m allocated a new object with location p'_{pp} at program point pp, and p_{pp} was lifted. In that case, p_{pp} would be abstracted in a flow-sensitive fashion in c', m' and in a flow-insensitive fashion in c, m. Therefore we need to record that p_{pp} used to be abstract in a flow-sensitive fashion, and that lifting occurred somewhere between c', m' and c, m: this will be done using *filters* (which are the concrete counter-part of abstract filters).

5.3.1 Heap Decompositions

We are now going to define formally what is the decomposition of a heap between a sub-heap (that will be handled in a flow-insensitive fashion) and local heaps (that will be handled in a flow-sensitive fashion). To do so we first need several definitions.

Heap Formally we defined heaps as finite sequences of key-value bindings between a location and a memory block. We can then state that some location ℓ maps to b by $(\ell \mapsto b) \in H$. The active domain of a heap H, denoted by $dom(H)$, is the finite set of locations having a mapping in H.

For convenience reasons, we would like to see a heap H as a function from the set of locations to memory block: to do so we use the special symbol $\perp$ that we introduced for abstract flow-sensitive heap entries. We will see the heap as a function that maps any location to a memory block or $\bot$. Since the heap is a finite sequence of key-value bindings between a location and a memory block, this function has a finite support. To summarize, if one reads $(\ell \mapsto b) \in H$ then we know that ℓ is in the active domain of H and that it points to the memory block b, whereas $H(\ell)$ may be either a memory block, or the empty block $\bot$.

Local Heap Intuitively a local heap K is a heap such that for all pp, there is at most one memory block b such that $(\mathsf{pp} \mapsto b) \in K$. For technical reasons we will consider a slightly different definition: a local heap is a finite sequence of key-value bindings from locations to memory block or $\bot$ such that there is *exactly* one key-value binding for all pp. Formally we have:

Definition 18. A heap K is a local heap if and only if it satisfies the following equations:

- $\forall \mathsf{pp}, p, p'.\ p_{\mathsf{pp}} \in dom(K) \wedge p'_{\mathsf{pp}} \in dom(K) \Rightarrow p = p'$
- $\forall \mathsf{pp}. \exists p. (p_{\mathsf{pp}} \mapsto \_) \in K$

Remark 2. Observe that if a heap H and some local heaps $(K_i)_{i \leq n}$ have disjoint domains then we can easily define their union.

We define the relation $H \rightarrow_{\mathsf{ref}} G$ between two heaps (local or not), to holds if the heap H contains a memory block storing a location to an element of G.

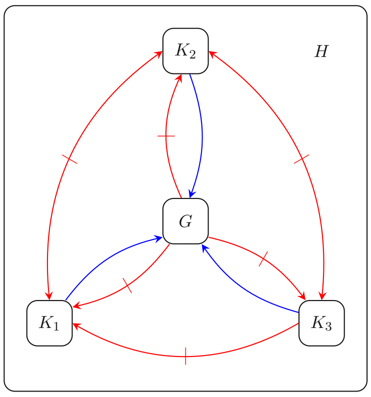
Definition 19. $H \rightarrow_{\mathsf{ref}} G$ if and only if there exists $(\_ \mapsto b) \in H$ such that one of the following cases holds:

- $b = \{\!|c; (f_i \mapsto v_i)^*|\!\} \in H$ and there exists j such that $v_j \in dom(G)$.
- $b = \{\!|@c; (f_i \mapsto v_i)^*|\!\} \in H$ and there exists j such that $v_j \in dom(G)$.
- $b = \tau[v^*] \in H$ and there exists j such that $v_j \in dom(G)$.

Now we can define what the heap decomposition of a heap together with a static heap is. Intuitively it is a partitioning of the heap H into a heap G and a finite set of local heaps $(K_i)_{i\leq n}$ such we have no locations going from G to any K_i, or from K_i to K_j for any $i \neq j$ (we allow locations from K_i to K_i or to G, and locations from G to itself). Formally:

Definition 20. $(G,(K_i)_{i\leq n})$ is a heap decomposition of $H \cdot S$ if and only if:

- $H = G \cup \bigcup_{i\leq n} K_i$
- $\forall i. dom(G) \cap dom(K_i) = \emptyset$
- $\forall i \neq j. dom(K_i) \cap dom(K_j) = \emptyset$
- $\forall i. G \cup S \not\rightarrow_{\mathsf{ref}} K_i$ and $\forall j \neq i. K_i \not\rightarrow_{\mathsf{ref}} K_j$



Example: a local heap decomposition with three local heaps.

5.3.2 Filter History

We are now going to define formally what the history of a configuration is. As we mentioned earlier, this is used to determine which locations were lifted, and when (in a given call-stack). It turns out that this definition is quite technical, because we need to make sure that the history of a configuration respected some properties: no locations should have been lifted twice, and a location to an object cannot appear in a local state that is situated in the call-stack *before* the local state that allocated this object.

First, we are going to define what a filter is. Filters are going to be used to represent one *layer* of the history, that is which locations were lifted between two local states.

Definition 21. A filter lk is a mapping from locations to $\{0,1\}$ such that for all pp, there exists at most one p such that $\mathsf{lk}(p_{\mathsf{pp}}) = 1$. Besides we define the following function:

$$\mathsf{lk} \sqcup^{\mathsf{loc}} \mathsf{lk}' = \left(p_{\mathsf{pp}} \mapsto \begin{cases} 1 \text{ if } \mathsf{lk}'(p_{\mathsf{pp}}) = 1 \\ 1 \text{ if } \mathsf{lk}(p_{\mathsf{pp}}) = 1 \text{ and } \forall p'_{\mathsf{pp}}, \mathsf{lk}'(p'_{\mathsf{pp}}) = 0 \\ 0 \text{ otherwise} \end{cases} \right)^*$$

Proposition 1. *The binary operation $\sqcup^{\mathsf{loc}}$ admits $(\mathsf{pp} \mapsto 0)^*$ as left and right neuter and is associative.*

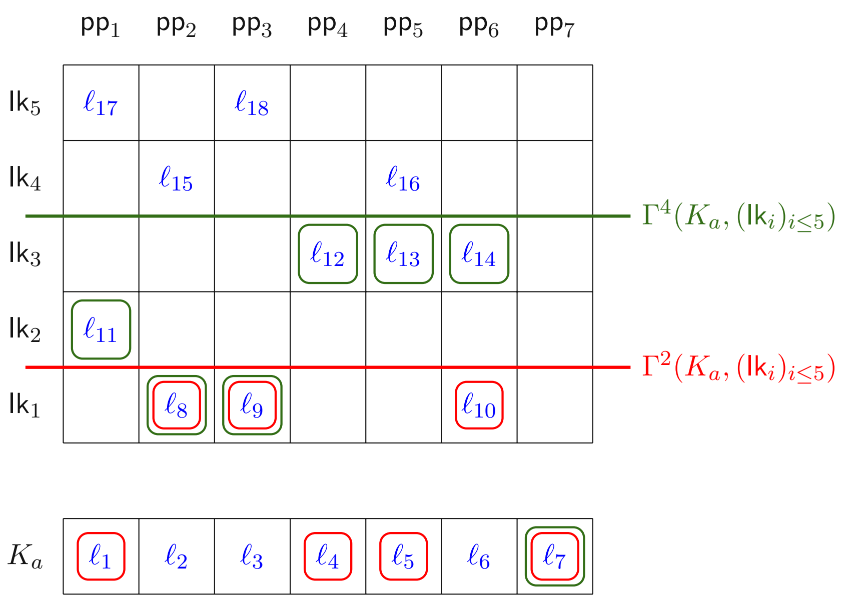
Remark 3. $\sqcup^{\mathsf{loc}}$ is **not** commutative.

The history of a call-stack $\alpha = L_1 :: \cdots :: L_n$ is going to be recorded using a list of filters $(\mathsf{lk}^j)_j$, such that for all i, lk_i records which locations were lifted between L_i and L_{i+1}. We then define, for all i, the function $\Gamma^i(K_a, (\mathsf{lk}^j)_j)$ that, given a local heap and an history, give us which for all program point pp the location which is handled in a flow-sensitive fashion in the local state L_i.

Definition 22. For all $i \in \mathbb{N} \cup \{+\infty\}$, $\Gamma^i(K_a, (\mathsf{lk}^j)_j)$ is the function defined as follows: let $\mathsf{lk} = \mathsf{lk}^1 \sqcup^{\mathsf{loc}} \dots \sqcup^{\mathsf{loc}} \mathsf{lk}^{i-1}$, then

$$\Gamma^i(K_a, (\mathsf{lk}^j)_j) = \left(\mathsf{pp} \mapsto \begin{cases} p_{\mathsf{pp}} \text{ if } \mathsf{lk}(p_{\mathsf{pp}}) = 1 \\ p_{\mathsf{pp}} \text{ if } p_{\mathsf{pp}} \in dom(K_a) \wedge \forall p'_{\mathsf{pp}}, \mathsf{lk}(p'_{\mathsf{pp}}) = 0 \end{cases} \right)^*$$

A graphical representation of Γ on an example can be found in Figure 5.18.



Convention: Each line of the table represents one local filter, by having a pointer ℓ in position $(\mathsf{lk}_i, \mathsf{pp}_j)$ if and only if there exists p such that $\ell = p_{\mathsf{pp}}$ and $\mathsf{lk}_i(\ell) = 1$. The last line represent the domain of the local heap K_a.
The pointer framed by red (resp. green) in column pp_i is the image of pp_i by $\Gamma^2(K_a(\mathsf{lk}_i)_{i\leq 5})$ (resp. $\Gamma^4(K_a, (\mathsf{lk}_i)_{i\leq 5})$).

Table 5.18: Graphical representation of the $\Gamma^j(K_a, (\mathsf{lk}_i)_{i\leq n})$ functions

Proposition 2 (Properties of Γ)**.** *For all $(K_a, (\mathsf{lk}_i)_{1\leq i\leq n})$ we have :*

1. *For all $i \in \{n+1, n+2, \dots\} \cup \{\infty\}$, $\Gamma^i(K_a, (\mathsf{lk}_j)_{1\leq j\leq n}) = \Gamma^{n+1}(K_a, (\mathsf{lk}_j)_{1\leq j\leq n})$*

2. *If $n \geq 2$, then*
   for all $i > 1$, $\Gamma^{i+1}(K_a, (\mathsf{lk}_j)_{1\leq j\leq n}) = \Gamma^i(K_a, (\mathsf{lk}_1 \sqcup^{\mathsf{loc}} \mathsf{lk}_2) :: (\mathsf{lk}_j)_{3\leq j\leq n})$

3. *For all* $i \geq 0$, $\Gamma^i(K_a, (\mathsf{lk}_j)_{1\leq j\leq n}) = \Gamma^{i+1}(K_a, (\mathsf{pp} \mapsto 0)^* :: (\mathsf{lk}_j)_{1\leq j\leq n})$

4. *Let* K'_a *be a local heap such that* $dom(K_a) = dom(K'_a)$*. Then for all* j *we have:*

$$\Gamma^i(K_a, (\mathsf{lk}_j)_{1\leq j\leq n}) = \Gamma^i(K'_a, (\mathsf{lk}_j)_{1\leq j\leq n})$$

5. *Let* lk_a *be a filter such that* $\forall \ell, \mathsf{lk}_a(\ell) = 1 \implies \ell \in dom(K_a)$*. Let* K'_a *be a local heap such that :*

$$dom(K'_a)\backslash \left\{p_{\mathsf{pp}} \in dom(K'_a) \mid \exists p', \mathsf{lk}_a(p'_{\mathsf{pp}}) = 1\right\} \subseteq dom(K_a)$$

Then for all $i \geq 2$ *we have:*

$$\Gamma^i(K_a, (\mathsf{lk}_j)_{1\leq j\leq n}) = \Gamma^i(K'_a, (\mathsf{lk}_a \sqcup^{\mathsf{loc}} \mathsf{lk}_1) :: (\mathsf{lk}_j)_{2\leq j\leq n})$$

We can now define when $(K, (\mathsf{lk}^j)_j)$ is a filter history of a call-stack α. Equation (5.1) expresses that a location never appears before it was allocated: this is done by stating that if, for a given pp, the location p_{pp} being handled in a flow-sensitive fashion in the local state L_i is not the same one than in local state L_j (where L_j appears before L_i in the call-stack), then no object was stored at location p_{pp} when L_j was the top-most element of the call-stack. Therefore p_{pp} cannot appear in any of the local state $L_j :: \dots L_n$. Equation (5.2) expresses the fact that no location was lifted twice, and that if a location is in the local heap then it was never lifted.

Definition 23. $(K, (\mathsf{lk}^j)_j)$ is a filter history of $\alpha = L_1 :: \dots :: L_n$ if and only if for all $1 \leq i < l \leq n$ and for all pp we have:

$$\Gamma^i(K, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^l(K, (\mathsf{lk}^j)_j)(\mathsf{pp}) \implies \Gamma^i(K, (\mathsf{lk}^j)_j)(\mathsf{pp}) \notin dom(L_l :: \dots :: L_n) \quad (5.1)$$
$$\forall i, \forall p_{\mathsf{pp}}, \Big((i = 0 \land p_{\mathsf{pp}} \in dom(K)) \lor \mathsf{lk}^i(p_{\mathsf{pp}}) = 1\Big) \implies \forall j \neq i, \mathsf{lk}^j(p_{\mathsf{pp}}) = 0 \quad (5.2)$$

The following (rather technical) lemma gives sufficient conditions to show that $(K'_a, (\mathsf{lk}'^j)_j)$ is a filter history, knowing that $(K_a, (\mathsf{lk}^j)_j)$ is a filter history and that $(K_a, (\mathsf{lk}^j)_j)$ and $(K'_a, (\mathsf{lk}'^j)_j)$ coincide everywhere except on the top-most filter and on the local heap.

Lemma 11. *Let* $(K, (\mathsf{lk}^j)_j)$ *be a filter history of* $\alpha = L_1 :: \alpha_t$*. Let* $\alpha' = L'_1 :: \alpha_t$*, and* $(K'_a, (\mathsf{lk}'^j)_j)$ *be such that* $(\mathsf{lk}'^j)_j = \mathsf{lk}'^1 :: (\mathsf{lk}^j)_{j>1}$*, and let* n *be the length of* α'*. If the four following conditions hold:*

$$\forall i > 1, \forall \mathsf{pp}, \Gamma^i(K, (\mathsf{lk}^j)_j)(\mathsf{pp}) = \Gamma^i(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \quad (5.3)$$
$$(dom(K')\backslash dom(K)) \cap dom(\alpha_t) = \emptyset \quad (5.4)$$
$$(dom(K')\backslash dom(K)) \cap \{\ell \mid \exists j, \mathsf{lk}^j(\ell) = 1\} = \emptyset \quad (5.5)$$
$$\{\ell \mid \mathsf{lk}'^1(\ell) = 1 \land \mathsf{lk}'^1(\ell) \neq \mathsf{lk}^1(\ell)\} \subseteq dom(K)\backslash dom(K') \quad (5.6)$$

then $(K', (\mathsf{lk}'^j)_j)$ *is a filter history of* α'*.*

Proof. This proof is done in two steps:

- First we are going to show that for all $1 \leq i < j < n$ we have:

$$\begin{aligned} &\Gamma^i(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \implies \\ &\Gamma^i(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \notin dom(\alpha'_l :: \ldots :: \alpha'_n) \end{aligned} \tag{5.7}$$

  - For $1 < i < l \leq n$, using Equation (5.3) we have that $\Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp})$ implies that $\Gamma^i(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^l(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})$. Since $(K_a, (\mathsf{lk}^j)_j)$ is a filter history of $L_1 :: \alpha_t$, this implies that $\Gamma^i(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \notin dom(\alpha_l :: \ldots :: \alpha_n)$. Since $l > 1$, $dom(\alpha_l :: \ldots :: \alpha_n) = dom(\alpha'_l :: \ldots :: \alpha'_n)$. Moreover using Equation (5.3) again we know that $\Gamma^i(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) = \Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp})$, therefore Equation (5.7) holds.
  - For $i = 1$, and $1 < l \leq n$. If $\Gamma^1(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) = \Gamma^1(K, (\mathsf{lk}^j)_j)(\mathsf{pp})$ then the same argument works. If $\Gamma^1(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^1(K, (\mathsf{lk}^j)_j)(\mathsf{pp})$, then since locations are annotated by their allocation point, and each local heap domain contains at most one location for each allocation point, we have $\Gamma^1(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \in (dom(K') \backslash dom(K))$. Therefore by applying Equation (5.4) we get that $\Gamma^1(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \notin dom(\alpha_t)$, which shows that Equation (5.7) holds.

- Now we are going to show that:

$$\forall i, \forall p_{\mathsf{pp}}, \Big((i = 0 \land p_{\mathsf{pp}} \in dom(K')) \lor \mathsf{lk}'^i(p_{\mathsf{pp}}) = 1\Big) \implies \forall j \neq i, \mathsf{lk}'^j(p_{\mathsf{pp}}) = 0$$

  Since we know that $(K, (\mathsf{lk}^j)_j)$ is a filter history, we just need to show it for $i = 0$ and $i = 1$.

  - $i = 0$. Let $\ell = p_{\mathsf{pp}} \in dom(K')$. In a first time assume that $\ell \in dom(K)$. Since $(K, (\mathsf{lk})^j)_j$ is a filter history we know that for all $j > 2, \mathsf{lk}'^j(\ell) = \mathsf{lk}^j(\ell) = 0$. It remains to show that $\mathsf{lk}'^1(\ell) = \mathsf{lk}^1(\ell) = 0$: if $\mathsf{lk}'^1(\ell) = 0$ then we have nothing to prove, and if $\mathsf{lk}'^1(\ell) \neq 0$ then since $\ell \in dom(K')$, Equation (5.6) gives us that $\mathsf{lk}^1(\ell) = \mathsf{lk}'^1(\ell) \neq 0$, which contradicts the fact that $(K, (\mathsf{lk})^j)_j$ is a filter history.

    Now assume that $\ell \notin dom(K)$. Then by Equation (5.5) we know that $\forall j > 2, \mathsf{lk}'^j(\ell) = \mathsf{lk}^j(\ell)$. Besides by Equation (5.6) we know that either $\mathsf{lk}'^1(\ell) = 0$, in which case we have nothing to prove, or that $\mathsf{lk}'^1(\ell) = \mathsf{lk}^1(\ell) = 1$, which contradict Equation (5.5).
  - $i = 1$. Let $\ell = p_{\mathsf{pp}}$ be such that $\mathsf{lk}'^1(\ell) = 1$. If $\mathsf{lk}'^1(\ell) = \mathsf{lk}^1(\ell)$ then since $(K, (\mathsf{lk})^j)_j$ is a filter history we know that for all $j > 2, \mathsf{lk}'^j(\ell) = \mathsf{lk}^j(\ell) = 0$. If $\mathsf{lk}'^1(\ell) \neq \mathsf{lk}^1(\ell)$ then by Equation (5.6) we know that $\ell \in dom(K)$ and we conclude again by using the fact that $(K, (\mathsf{lk})^j)_j$ is a filter history.

□

5.3.3 Configuration Decomposition

The heap decomposition notion is relative to a *heap*, and the filter history notion is relative to a *call-stack*. We then link these two notions into the local configuration decomposition notion, that is relative to a *local configuration.*

Definition 24. $(G, (K_i)_i, K, (\mathsf{lk}^j)_j)$ is a local configuration decomposition of $\Sigma = \ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ if and only if:

- $G, (K_i)_i$ is a heap decomposition of $H \cdot S$ and $K \in (K_i)_i$
- $\mathit{dom}(\alpha) \subseteq \mathit{dom}(G) \cup \mathit{dom}(K)$
- $(K, (\mathsf{lk}^j)_j)$ is a filter history of α
- $\forall i \in \pi, \exists p_\lambda, (p_\lambda \mapsto i) \in G$
- $\forall \ell \in \gamma, \ell \in \mathit{dom}(G)$
- $\ell \in \mathit{dom}(G)$

Finally we use the local configuration decomposition notion to define what is a *configuration decomposition.*

Definition 25. Let $\Omega = \phi_1 :: \cdots :: \phi_n$ and $\Xi = \psi_1 :: \cdots :: \psi_m$. Then $(G, (K_i, (\mathsf{lk}^{i,j})_j)_{i \leq n+m})$ is a configuration decomposition of $\Omega \cdot \Xi \cdot H \cdot S$ if and only if:

- $G, (K_i)_i$ is a heap decomposition of $H \cdot S$.
- for all $i \leq n$, if $\phi_i \in \{\langle \ell, s, \pi, \gamma, \alpha \rangle, \underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}\}$ then $(G, (K_j)_j, K_i, (\mathsf{lk}^{i,j})_j)$ is a heap decomposition history of $\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ with local heap K_i.
- for all $n+1 \leq i \leq m+n$, if $\psi_i = \langle\!\langle \ell, \ell', \pi, \gamma, \alpha \rangle\!\rangle$ then $(G, (K_j)_j, K_i, (\mathsf{lk}^{i,j})_j)$ is a heap decomposition history of $\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ with local heap K_i.

5.3.4 Well-Formedness

First we are going to make some assumptions on the program P, which are guaranteed by the Java type system: we assume that the exception table built by the compiler only contain entries for exception class, and that the compiler guarantee type soundness for the thread and exception rules.

Assumption 4 (Exception Table Correction)**.** If $\mathsf{ExcptTable}(c, m, pc, c')$ is defined (i.e is equal to some pc' or to $\perp$) then $c' \leq \mathsf{Throwable}$.

Assumption 5 (Type Soundness Guarantee)**.**

- If $\Sigma, \texttt{throw}\ r_e \Downarrow \Sigma'$ and $H(\Sigma[\![r_e]\!]) = \{\!|c'; (f \mapsto v)^*|\!\}$ then $c' \leq \mathsf{Throwable}$.
- If $\Sigma, st \Downarrow \Sigma'$ where $st \in \{\texttt{start-thread}\ r_t, \texttt{interrupt}\ r_t, \texttt{join}\ r_t\}$ and $H(\Sigma[\![r_t]\!]) = \{\!|c'; (f \mapsto v)^*|\!\}$ then $c' \leq \mathsf{Thread}$.

We are going to need some *well-formedness* properties in the proof, that are preserved by the local configuration and configuration reductions.

Definition 26. A local configuration $\Sigma = \ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ is *well-formed* if and only if, whenever $\alpha = L_1 :: \ldots :: L_n$ or $\alpha = \texttt{AbNormal}(L_1 :: \ldots :: L_n)$, we have:

- For all i, $L_i = \mathsf{waiting}(\_,\_)$ implies that $i = 1$ and $\alpha = \texttt{AbNormal}(L_1 :: \ldots :: L_n)$.
- If $L_1 = \mathsf{waiting}(\ell_o, \_)$ then $L_2 = \langle c, m, pc \cdot \_ \cdot st^* \cdot \_\rangle$ with $st_{pc} = \texttt{wait}\ r_i$ and $\ell_o = \Sigma[\![r_i]\!]$.
- For all $i \leq n$, if $L_i = \langle c, m, pc \cdot v^* \cdot st^* \cdot R\rangle$ and $R(r) = \ell$ then $\ell \in dom(H)$.
- For all $\ell \in \gamma$, if $H(\ell) = \{\!|c'; \_|\!\}$ then $c' \leq \mathsf{Thread}$.
- Either $n \in \{0, 1\}$, or $n \geq 2$ and for each $i \in [2, n]$, either of the following conditions hold true:
  - $L_i = \langle c', m', pc' \cdot v'^* \cdot st'^* \cdot R'\rangle$ and $L_{i-1} = \langle c, m, pc \cdot \_ \cdot st^* \cdot R\rangle$ with $st_{pc} = \texttt{invoke}\ r_o\ m'\ r_1, \ldots, r_n$,
    $lookup(type_H(R(r_o)), m') = (c', st'^*)$, $sign(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ and $v'^* = (R(r_k))^{k \leq n}$
  - $L_i = \langle c', m', pc' \cdot v'^* \cdot st'^* \cdot R'\rangle$ and $L_{i-1} = \langle c, m, pc \cdot \_ \cdot st^* \cdot R\rangle$ with $st_{pc} = \texttt{sinvoke}\ c'\ m'\ r_1, \ldots, r_n$,
    $lookup(c', m') = (c', st'^*)$, $sign(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ and $v'^* = (R(r_k))^{k \leq n}$.

Lemma 12 (Preserving Local Well-formation)**.** *If Σ is well-formed and $\Sigma \rightsquigarrow^* \Sigma'$, then Σ' is well-formed.*

Proof. By induction on the length of the reduction sequence and a case analysis on the last rule applied. □

Definition 27. A heap H is *well-typed* if and only if, whenever $H(\ell) = \{\!|c; (f_i \mapsto v_i)^{i \leq n}|\!\}$, for all $i \in [1, n]$ we have $type_H(v_i) \leq \tau_i$, where τ_i is the declared type of field f_i for an object of type c according to the underlying program.

Assumption 6 (Java Type Soundness)**.**

If $\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S \rightsquigarrow \ell' \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S'$, then for any value v we have $type_{H'}(v) \leq type_H(v)$. Moreover, if H is well-typed, then also H' is well-typed.

Definition 28. A configuration $\Psi = \Omega \cdot \Xi \cdot H \cdot S$ is *well-formed* if and only if:

- whenever $\Omega = \Omega_0 :: \varphi :: \Omega_1$ with $\varphi \in \{\langle \ell, s, \pi, \gamma, \alpha \rangle, \underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}\}$, we have
  - $H(\ell) = \{\!|c; (f \mapsto v)^*|\!\}$ for some activity class c and $\ell = p_c$ for some pointer p
  - $\Sigma = \ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ is a well-formed local configuration
- whenever $\langle\!\langle \ell, \ell', \pi, \gamma, \alpha \rangle\!\rangle \in \Xi$, we have
  - $H(\ell) = \{\!|c; (f \mapsto v)^*|\!\}$ for some activity class c and $\ell = p_c$ for some pointer p
  - $H(\ell') = \{\!|c'; (f' \mapsto v')^*|\!\}$ for some thread class c'
  - $\Sigma = \ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ is a well-formed local configuration
- H is a well-typed heap.

Lemma 13 (Preserving Well-formation)**.** *If Ψ is well-formed and $\Psi \Rightarrow^* \Psi'$, then Ψ' is well-formed.*

Proof. By induction on the length of the reduction sequence and a case analysis on the last rule applied, using Lemma 12 and Assumption 6 to deal with case (A-Active). □

From now on, we tacitly focus only on well-formed configurations. All the formal results only apply to them: notice that well-formed configurations always reduce to well-formed configurations by Lemma 13.

5.3.5 Representation Functions

From now on, we will consider only ground abstract values, and we will identify these values with their evaluation in the abstract domain $\hat{D}$.

We are now ready to define the *representation functions* that we will use in the proof. A representation function is a (possibly parametrized) function that takes as input a concrete value and returns an abstraction of this value. The final goal of this section is to define the representation function $\beta_{Cnf}(\Psi)$ that takes as input a configuration Ψ and returns a set of sets of abstract facts, where each set of abstract facts X in $\beta_{Cnf}(\Psi)$ is an abstraction of Ψ for a given configuration decomposition.

Basic Representation Functions

First we presuppose the existence of a representation function β_{Prim} which associates to each primitive value *prim* a corresponding abstract value $\{\widehat{prim}\}$. We then define the following representation function, that abstracts a filter lk into an abstract filter $\hat{k}$, where the $\hat{k}$ is the abstract filters that maps a program point pp to 1 iff there exists a locations ℓ annotated with pp (i.e. $\ell = p_{\mathsf{pp}}$) such that $\mathsf{lk}(\ell) = 1$.

$$\beta_{Filter}(\mathsf{lk}) = \left(\mathsf{pp} \mapsto \begin{cases} 1 & \text{if } \exists p_{\mathsf{pp}}, \mathsf{lk}(p_{\mathsf{pp}}) = 1 \\ 0 & \text{otherwise} \end{cases}\right)^*$$

We then define the flow-sensitive and flow-insensitive location and value representation functions. The flow-sensitive representation functions are going to be used when the analysis is flow-sensitive (for example one registers), and the flow-insensitive representation functions are going to be used when the analysis is *not* flow-sensitive (for example on the static heap).

	flow-sensitive abstraction	flow-insensitive abstraction
location	$\beta_{Loc}(p_\lambda, K_a, (\mathsf{lk}^j)_j) = \begin{cases} \mathsf{FS}(\lambda) & \text{if } \lambda = \mathsf{pp} \wedge p_{\mathsf{pp}} = \Gamma^\infty(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \\ \mathsf{NFS}(\lambda) & \text{otherwise} \end{cases}$	$\beta_{Lab}(p_\lambda) = \lambda$
value	$\beta_{LocVal}(v, K_a, (\mathsf{lk}^j)_j) = \begin{cases} \beta_{Prim}(v) & \text{if } v = prim \\ \beta_{Loc}(v, K_a, (\mathsf{lk}^j)_j) & \text{if } v = \ell \end{cases}$	$\beta_{Val}(v) = \begin{cases} \beta_{Prim}(v) & \text{if } v = prim \\ \mathsf{NFS}(\beta_{Lab}(v)) & \text{if } v = \ell \end{cases}$

We typically omit brackets around singleton abstract values, and we will write $\beta_{LocVal}(v, K_a)$ instead of the more verbose $\beta_{LocVal}(v, K_a, \varepsilon)$ when the filter list is empty.

Remark 4. Recall that by definition, only locations annotated with *program points* can be abstracted as flow-sensitive abstract locations. In particular activity object and their intents are always flow-insensitive.

With these representation functions, we can define the flow-sensitive representation function β_{LocBlk} for local blocks, and the flow-insensitive representation function β_{Blk} for blocks.

$$\beta_{LocBlk}(l, K_a) = \begin{cases} \{\!|c; (f \mapsto \hat{v})^*|\!\} & \text{if } l = \{\!|c; (f \mapsto v)^*|\!\} \text{ and } \forall i : \beta_{LocVal}(v_i, K_a) = \hat{v}_i \\ \{\!|@c; \hat{v}|\!\} & \text{if } l = \{\!|@c; (f \mapsto v)^*|\!\} \text{ and } \hat{v} = \sqcup_i \beta_{LocVal}(v_i, K_a) \\ \tau[\hat{v}] & \text{if } l = \tau[v^*] \text{ and } \hat{v} = \sqcup_i \beta_{LocVal}(v_i, K_a) \\ \bot & \text{if } l = \bot \end{cases}$$

$$\beta_{Blk}(b) = \begin{cases} \{\!|c; (f \mapsto \hat{v})^*|\!\} & \text{if } b = \{\!|c; (f \mapsto v)^*|\!\} \text{ and } \forall i : \beta_{Val}(v_i) = \hat{v}_i \\ \{\!|@c; \hat{v}|\!\} & \text{if } b = \{\!|@c; (f \mapsto v)^*|\!\} \text{ and } \hat{v} = \sqcup_i \beta_{Val}(v_i) \\ \tau[\hat{v}] & \text{if } b = \tau[v^*] \text{ and } \hat{v} = \sqcup_i \beta_{Val}(v_i) \end{cases}$$

Advanced Representation Functions

We define the representation function $\beta_{LHeap}(K_a)$ abstracting a local heap into an abstract flow-sensitive heap as follows:

$$\beta_{LHeap}(K_a) = \{(\mathsf{pp} \mapsto \beta_{LocBlk}(K_a(p_{\mathsf{pp}}), K_a)) \mid p_{\mathsf{pp}} \in dom(K_a)\}$$

We have three representation functions used to abstract a local state L taken from the call-stack α of a local configuration Σ, where ℓ is the pointer to the activity or thread object and $K_a, (\mathsf{lk}^n)_n$ is a filter history of Σ:

- If a local state L is not the top-most local state in its call-stack then we use $\beta^{\ell}_{LstInv}(L, n_0, c', K_a, (\mathsf{lk}^n)_n)$ where n_0 is the position is the call-stack and c' is the class of the object that L invoked a method upon.

$$\begin{aligned}\beta^{\ell}_{LstInv}(\langle \mathsf{pp}\cdot u^*\cdot st^*\cdot R\rangle, n_0, c', K_a, (\mathsf{lk}^n)_n) = \\ \Big\{\mathsf{Inv}^{c'}_{\mathsf{pp}}((\hat{\lambda}_t,\hat{u}^*);\hat{v}^*;\hat{k}) \mid \hat{k} = \beta_{Filter}(\mathsf{lk}^{n_0}) \\ \wedge \forall j : \hat{u}_j = \beta_{LocVal}(u_j, K_a, (\mathsf{lk}^n)_{\mathbf{n\leq n_0}}) \\ \wedge\, \hat{\lambda}_t = \beta_{Val}(\ell) \wedge \forall k : \hat{v}_k = \beta_{LocVal}(R(r_k), K_a, (\mathsf{lk}^n)_{\mathbf{n<n_0}})\Big\}\end{aligned}$$

- If L is the top-most local state, and α is **not** abnormal, then we use $\beta^{\ell}_{Lst}(L, K_a, (\mathsf{lk}^n)_n)$.

$$\begin{aligned}\beta^{\ell}_{Lst}(\langle \mathsf{pp}\cdot u^*\cdot st^*\cdot R\rangle, K_a, (\mathsf{lk}^n)_n) = \Big\{\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}_t,\hat{u}^*);\hat{v}^*;\hat{h};\hat{k}) \mid \hat{k} = \beta_{Filter}(\mathsf{lk}^1) \\ \wedge \forall j : \hat{u}_j = \beta_{LocVal}(u_j, K_a, (\mathsf{lk}^n)_{\mathbf{n\leq 1}}) \\ \wedge\, \hat{\lambda}_t = \beta_{Val}(\ell) \wedge \forall k : \hat{v}_k = \beta_{LocVal}(R(r_k), K_a, (\mathsf{lk}^n)_{\mathbf{n<1}}) \\ \wedge\, \hat{h} = \beta_{LHeap}(K_a)\Big\}\end{aligned}$$

- If L is the top-most local state, and α is abnormal, then we use $\beta^{\ell}_{ALst}(\langle \mathsf{pp}\cdot u^*\cdot st^*\cdot R\rangle, K_a, (\mathsf{lk}^n)_n)$.

$$\begin{aligned}\beta^{\ell}_{ALst}(\langle \mathsf{pp}\cdot u^*\cdot st^*\cdot R\rangle, K_a, (\mathsf{lk}^n)_n) = \Big\{\mathsf{AState}_{\mathsf{pp}}((\hat{\lambda}_t,\hat{u}^*);\hat{v}^*;\hat{h};\hat{k}) \mid \hat{k} = \beta_{Filter}(\mathsf{lk}^1) \\ \wedge \forall j : \hat{u}_j = \beta_{LocVal}(u_j, K_a, (\mathsf{lk}^n)_{\mathbf{n\leq 1}}) \wedge \hat{\lambda}_t = \beta_{Val}(\ell) \\ \wedge \forall k : \hat{v}_k = \beta_{LocVal}(R(r_k), K_a, (\mathsf{lk}^n)_{\mathbf{n<1}}) \wedge \hat{h} = \beta_{LHeap}(K_a)\Big\}\end{aligned}$$

Using these, we can define how the call-stack α is abstracted. For all $i \leq n$, let $L_i = \langle c_i, m_i, pc_i \cdot \_ \cdot \_ \cdot \_\rangle$. If $\alpha = L_1 :: \cdots :: L_n$ and $n \geq 1$ then:

$$\begin{aligned}\beta^{\ell}_{Call}(\mathsf{waiting}(\_,\_) :: \alpha, K_a, (\mathsf{lk}^n)_n) &= \beta^{\ell}_{Call}(\alpha, K_a, (\mathsf{lk}^n)_n) \\ &= \beta^{\ell}_{Lst}(L_1, K_a, (\mathsf{lk}^n)_n) \cup \\ &\qquad \bigcup_{i\in[2,n]} \beta^{\ell}_{LstInv}(L_i, i, c_{i-1}, K_a, (\mathsf{lk}^n)_n) \\ \beta^{\ell}_{Call}(\mathtt{AbNormal}(\alpha), K_a, (\mathsf{lk}^n)_n) &= \beta^{\ell}_{ALst}(L_1, K_a, (\mathsf{lk}^n)_n) \cup \\ &\qquad \bigcup_{i\in[2,n]} \beta^{\ell}_{LstInv}(L_i, i, c_{i-1}, K_a, (\mathsf{lk}^n)_n) \\ \beta^{\ell}_{Call}(\varepsilon, K_a, (\mathsf{lk}^n)_n) &= \beta^{\ell}_{Call}(\mathtt{AbNormal}(\varepsilon), K_a, (\mathsf{lk}^n)_n) \quad = \quad \emptyset\end{aligned}$$

We can now define the following representation functions:

$$\begin{array}{lcl}
\beta^G_{Heap}(H) & = & \left\{\mathsf{H}(\lambda, \hat{b}) \mid H(\ell') = b \wedge \lambda = \beta_{Lab}(\ell') \wedge \hat{b} = \beta_{Blk}(b) \wedge \ell' \in dom(G)\right\} \\
\beta_{Stat}(S) & = & \{\mathsf{S}(c, f, \hat{v}) \mid S = S', c.f \mapsto v \wedge \hat{v} = \beta_{Val}(v)\} \\
\beta^\ell_{Pact}(\pi) & = & \left\{\mathsf{I_c}(\hat{b}) \mid c = \beta_{Lab}(\ell) \wedge \pi = \pi_0 :: i :: \pi_1 \wedge \hat{b} = \beta_{Blk}(i)\right\} \\
\beta^G_{Pthr}(\gamma) & = & \left\{\mathsf{T}(\lambda, \hat{b}) \mid \gamma = \gamma_0 :: \ell :: \gamma_1 \wedge \lambda = \beta_{Lab}(\ell) \wedge (\ell \mapsto b) \in G \wedge \hat{b} = \beta_{Blk}(b)\right\}
\end{array}$$

$$\begin{array}{lcl}
\beta^G_{Frm}(\langle \ell, s, \pi, \gamma, \alpha\rangle, K_a, (\mathsf{lk}^j)_j) & = & \beta_{Frm}(\underline{\langle \ell, s, \pi, \gamma, \alpha\rangle}, K_a, (\mathsf{lk}^j)_j) \\
& = & \beta_{Frm}(\langle\!\langle \ell, \ell', \pi, \gamma, \alpha\rangle\!\rangle, K_a, (\mathsf{lk}^j)_j) \\
& = & \beta^\ell_{Call}(\alpha, K_a, (\mathsf{lk}^j)_j) \cup \beta^\ell_{Pact}(\pi) \cup \beta^G_{Pthr}(\gamma)
\end{array}$$

Let $\Omega = \varphi_1 :: \ldots :: \varphi_n$ and $\Xi = \psi_1 :: \ldots :: \psi_m$. We then define the representation function β^G_{Stk} abstracting the activity stack and the thread pool as follows:

$$\beta^G_{Stk}(\Omega, \Xi, (K_i, (\mathsf{lk}^{i,j})_j)_i) = \left(\bigcup_{i \in [1,n]} \beta^G_{Frm}(\varphi_i, K_i, (\mathsf{lk}^{i,j})_j)\right) \cup \left(\bigcup_{l \in [1,m]} \beta^G_{Frm}(\psi_l, K_{n+l}, (\mathsf{lk}^{n+l,j})_j)\right)$$

The representation function β_{Lcnf} abstracts a local configuration Σ into a set of sets of abstract facts, one for each local configuration decomposition of Σ:

$$\beta_{Lcnf}(\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S) = \Big\{\beta^\ell_{Call}(\alpha, K_a, (\mathsf{lk}^j)_j) \cup \beta^\ell_{Pact}(\pi) \cup \beta^G_{Pthr}(\gamma) \cup \beta^G_{Heap}(H) \cup \beta_{Stat}(S) \\ \mid (G, (K_i)_i, K_a, (\mathsf{lk}^j)_j) \text{ is a local configuration decomposition of } \ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S\Big\}$$

The representation function β_{Cnf} abstracts a configuration Ψ into a set of sets of abstract facts, one for each configuration decomposition of Ψ:

$$\beta_{Cnf}(\Omega \cdot \Xi \cdot H \cdot S) = \Big\{\beta^G_{Stk}(\Omega, (K_i, (\mathsf{lk}^{i,j})_j)_i) \cup \beta^G_{Heap}(H) \cup \beta_{Stat}(S) \\ \mid (G, (K_i, (\mathsf{lk}^{i,j})_j)_i) \text{ is a configuration decomposition of } \Omega \cdot \Xi \cdot H \cdot S\Big\}$$

Remark 5. The predicates $\mathsf{Inv}^{c'}_{\mathsf{pp}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{k})$ are used to abstract local states of function which have invoked some other method and are waiting for it to return. There are two differences with $\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$: the first one is that we drop the local heap, which is no longer needed since it will be replaced by the callee's local heap when it will return. The second difference is that we have extra information about the class c' implementing the invoked method.

Also observe that this invoke predicate does not appear in any rules, and that it is *only* used in the proof. Therefore it can be ignored in an implementation.

5.3.6 Pre-Orders

We will now define several pre-orders and relations used to compare abstract elements. Some abstract syntactic domains, such as abstract values and abstract memory blocks, have two different pre-orders used to compare them, that we distinguish by decorating one with a nfs superscript. The pre-order with the nfs superscript is a *flow-insensitive* pre-order.

Abstract Values Pre-Orders

We define the pre-order $\sqsubseteq_{Loc}$ on abstract location by:

$$\hat{\lambda} \sqsubseteq_{Loc} \hat{\lambda}' \text{ iff } \begin{cases} \hat{\lambda} = \mathsf{NFS}(\mathsf{pp}) \wedge \hat{\lambda}' = \mathsf{FS}(\mathsf{pp}) \\ \hat{\lambda} = \mathsf{FS}(\mathsf{pp}) \wedge \hat{\lambda}' = \mathsf{NFS}(\mathsf{pp}) \\ \hat{\lambda} = \hat{\lambda}' \end{cases}$$

Based on this, we define the pre-order $\sqsubseteq^{\mathsf{nfs}}$ on abstract values to the reflexive and transitive closure of $\sqsubseteq \cup \sqsubseteq_{Loc}$. We then build the pre-orders $\sqsubseteq^{\mathsf{nfs}}_{Seq}$ and $\sqsubseteq_{Seq}$ on sequences of abstract values by having $\hat{u}^* \sqsubseteq^{\mathsf{nfs}}_{Seq} \hat{v}^*$ (resp. $\hat{u}^* \sqsubseteq_{Seq} \hat{v}^*$) iff $\hat{u}^*$ and $\hat{v}^*$ have the same length and $\forall i : \hat{u}_i \sqsubseteq^{\mathsf{nfs}} \hat{v}_i$ (resp. $\forall i : \hat{u}_i \sqsubseteq \hat{v}_i$). We then define a pre-order $\sqsubseteq^{\mathsf{nfs}}_{Blk}$ on abstract memory blocks as follows:

- if $\hat{b} = \{\!|c; (f \mapsto \hat{u})^*|\!\}$ and $\hat{b}' = \{\!|c; (f \mapsto \hat{v})^*|\!\}$ and $\hat{u}^* \sqsubseteq^{\mathsf{nfs}}_{Seq} \hat{v}^*$, then $\hat{b} \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}'$
- if $\hat{b} = \{\!|@c; \hat{u}|\!\}$ and $\hat{b}' = \{\!|@c; \hat{v}|\!\}$ and $\hat{u} \sqsubseteq^{\mathsf{nfs}} \hat{v}$, then $\hat{b} \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}'$
- if $\hat{b} = \tau[\hat{u}]$ and $\hat{b}' = \tau[\hat{v}]$ and $\hat{u} \sqsubseteq^{\mathsf{nfs}} \hat{v}$, then $\hat{b} \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}'$

We also define the pre-order $\sqsubseteq_{Blk}$ on abstract memory blocks, which is the the flow-sensitive counterpart of $\sqsubseteq^{\mathsf{nfs}}_{Blk}$.

- if $\hat{b} = \{\!|c; (f \mapsto \hat{u})^*|\!\}$ and $\hat{b}' = \{\!|c; (f \mapsto \hat{v})^*|\!\}$ and $\hat{u}^* \sqsubseteq_{Seq} \hat{v}^*$, then $\hat{b} \sqsubseteq_{Blk} \hat{b}'$
- if $\hat{b} = \{\!|@c; \hat{u}|\!\}$ and $\hat{b}' = \{\!|@c; \hat{v}|\!\}$ and $\hat{u} \sqsubseteq \hat{v}$, then $\hat{b} \sqsubseteq_{Blk} \hat{b}'$
- if $\hat{b} = \tau[\hat{u}]$ and $\hat{b}' = \tau[\hat{v}]$ and $\hat{u} \sqsubseteq \hat{v}$, then $\hat{b} \sqsubseteq_{Blk} \hat{b}'$

Finally we define the relation $\sqsubseteq_{Filter}$ on abstract filters to be the equality order. Next, we state some simple properties satisfied by these pre-orders.

Proposition 3. *$\sqsubseteq^{nfs}_{Blk}$ is coarser than $\sqsubseteq_{Blk}$, and $\sqsubseteq^{nfs}$ is coarser than $\sqsubseteq$.*

Proposition 4. *If $\hat{u} \neq \bot$ and $\hat{u} \sqsubseteq \hat{v}$ and $\hat{u} \sqsubseteq \hat{w}$ then $\hat{v} \sqcap \hat{w} \neq \bot$*

Proof. Since $(\hat{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ is a lattice we know that $\hat{u} \sqsubseteq \hat{v} \sqcap \hat{w}$. Moreover $\hat{u} \neq \bot$, therefore $\hat{v} \sqcap \hat{u} \neq \bot$. □

Proposition 5. *For any abstract memory blocks $\hat{b}, \hat{b}'$, for any abstract values $\hat{u}, \hat{v}$ and for any field f we have*

$$\hat{b} \sqsubseteq^{\mathit{nfs}}_{\mathit{Blk}} \hat{b}' \wedge \hat{u} \sqsubseteq^{\mathit{nfs}} \hat{v} \implies \hat{b}[f \mapsto \hat{u}] \sqsubseteq^{\mathit{nfs}}_{\mathit{Blk}} \hat{b}'[f \mapsto \hat{v}]$$
$$\hat{b} \sqsubseteq_{\mathit{Blk}} \hat{b}' \wedge \hat{u} \sqsubseteq \hat{v} \implies \hat{b}[f \mapsto \hat{u}] \sqsubseteq_{\mathit{Blk}} \hat{b}'[f \mapsto \hat{v}]$$

Facts Pre-Orders

For all register r_o, class c'', abstract heap $\hat{h}$ and sequence of abstract values $\hat{v}^*$ we define the formula:

$$\begin{aligned}\mathsf{Call}^{\Delta}_{r_o,c'',m'}(\hat{v}^*; \hat{h}) = \exists \mathsf{pp}', c', ((&\mathsf{NFS}(\mathsf{pp}') \sqsubseteq \hat{v}_o \wedge \mathsf{H}(\mathsf{pp}', \{\!|c'; \_|\!\}) \in \Delta) \\ &\vee \left(\mathsf{FS}(\mathsf{pp}') \sqsubseteq \hat{v}_o \wedge \hat{h}(\mathsf{pp}') = \{\!|c'; \_|\!\}\right)) \\ &\wedge c' \leq c'' \wedge c'' \in \widehat{lookup}(m')\end{aligned}$$

Intuitively this states that element o of the abstract registers $\hat{v}^*$ over-approximates an abstract location to an abstract object $\{\!|c'; \_|\!\}$ in $\hat{h}$ or Δ, such abstract virtual dispatch resolution on c', m' return c''. We are now ready to define more complex relation between abstract facts, using the pre-orders defined in the previous subsection. Let Δ, Δ' be two finite sets of facts. We define the relations $\sqsubseteq_R$, $\sqsubseteq_A$ and $\sqsubseteq^{\Delta'}_{\mathit{Inv}}$ as follows:

- $\mathsf{LState}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{\mathit{call}}); \hat{u}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{\mathit{call}}); \hat{v}^*; \hat{h}'; \hat{k}')$ iff
  - $\hat{\lambda}^1_t = \hat{\lambda}^2_t$ and $\hat{u}^*_{\mathit{call}} \sqsubseteq_{\mathit{Seq}} \hat{v}^*_{\mathit{call}}$
  - $\hat{u}^* \sqsubseteq_{\mathit{Seq}} \hat{v}^*$
  - $\hat{k} \sqsubseteq_{\mathit{Filter}} \hat{k}'$
  - $\forall \mathsf{pp}, \hat{h}(\mathsf{pp}) \neq \bot \implies \hat{h}(\mathsf{pp}) \sqsubseteq_{\mathit{Blk}} \hat{h}'(\mathsf{pp})$
- $\mathsf{AState}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{\mathit{call}}); \hat{u}^*; \hat{h}; \hat{k}) \sqsubseteq_A \mathsf{AState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{\mathit{call}}); \hat{v}^*; \hat{h}'; \hat{k}')$ iff :

  $$\mathsf{LState}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{\mathit{call}}); \hat{u}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{\mathit{call}}); \hat{v}^*; \hat{h}'; \hat{k}')$$

- $\mathsf{Inv}^{c''}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{\mathit{call}}); \hat{u}^*; \hat{k}) \sqsubseteq^{\Delta}_{\mathit{Inv}} \mathsf{LState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{\mathit{call}}); \hat{v}^*; \hat{h}'; \hat{k}')$ iff:
  - $\hat{\lambda}^1_t = \hat{\lambda}^2_t$ and $\hat{u}^*_{\mathit{call}} \sqsubseteq_{\mathit{Seq}} \hat{v}^*_{\mathit{call}}$
  - $\hat{u}^* \sqsubseteq_{\mathit{Seq}} \hat{v}^*$
  - $\hat{k} \sqsubseteq_{\mathit{Filter}} \hat{k}'$
  - $\mathit{lookup}(c, m) = (\_, st^*)$, $st_{pc} = \texttt{invoke}\ r_o\ m'\ \_$ and $\mathsf{Call}^{\Delta}_{r_o,c'',m'}(\hat{v}'^*; \hat{h}')$

Finally, we define the pre-order <: by having $\Delta <: \Delta'$ (where Δ, Δ' are two finite sets of facts) if and only if:

- $\forall \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t^1, \hat{u}^*_{call}); \hat{u}^*; \hat{h}; \hat{k}) \in \Delta$, $\exists \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t^2, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}') \in \Delta'$ s.t.
$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t^1, \hat{u}^*_{call}); \hat{u}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t^2, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}')$$
- $\forall \mathsf{AState}_{c,m,pc}((\hat{\lambda}_t^1, \hat{u}^*_{call}); \hat{u}^*; \hat{h}; \hat{k}) \in \Delta$, $\exists \mathsf{AState}_{c,m,pc}((\hat{\lambda}_t^2, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}') \in \Delta'$ s.t.
$$\mathsf{AState}_{c,m,pc}((\hat{\lambda}_t^1, \hat{u}^*_{call}); \hat{u}^*; \hat{h}; \hat{k}) \sqsubseteq_A \mathsf{AState}_{c,m,pc}((\hat{\lambda}_t^2, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}')$$
- $\forall \mathsf{Inv}^{c''}_{c,m,pc}((\hat{\lambda}_t^1, \hat{u}^*_{call}); \hat{u}^*; \hat{k}) \in \Delta$, $\exists \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t^2, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}') \in \Delta'$ s.t.
$$\mathsf{Inv}^{c''}_{c,m,pc}((\hat{\lambda}_t^1, \hat{u}^*_{call}); \hat{u}^*; \hat{k}) \sqsubseteq^{\Delta'}_{Inv} \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t^2, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}')$$
- $\forall \mathsf{H}(\lambda, \hat{b}) \in \Delta$, $\exists \mathsf{H}(\lambda, \hat{b}') \in \Delta'$ such that $\hat{b} \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}'$
- $\forall \mathsf{S}(c, f, \hat{u}) \in \Delta$, $\exists \mathsf{S}(c, f, \hat{v}) \in \Delta'$ such that $\hat{u} \sqsubseteq^{\mathsf{nfs}} \hat{v}$
- $\forall \mathsf{I}_\mathsf{c}(\hat{b}) \in \Delta$, $\exists \mathsf{I}_\mathsf{c}(\hat{b}') \in \Delta'$ such that $\hat{b} \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}'$
- $\forall \mathsf{T}(\lambda, \hat{b}) \in \Delta$, $\exists \mathsf{T}(\lambda, \hat{b}') \in \Delta'$ such that $\hat{b} \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}'$

5.3.7 Preliminary Lemmas

Pre-Orders

Lemma 14. *For all set of facts Δ and Δ', if $\Delta \subseteq \Delta'$ then*
$$\mathsf{Call}^{\Delta}_{r_o,c'',m'}(\hat{v}^*; \hat{h}) \implies \mathsf{Call}^{\Delta'}_{r_o,c'',m'}(\hat{v}^*; \hat{h})$$
As a direct corollary, $\sqsubseteq^{\Delta'}_{Inv}$ is coarser than $\sqsubseteq^{\Delta}_{Inv}$.

Lemma 15. *If $\Delta \subseteq \Delta'$, and $\Delta' <: \Delta''$ then $\Delta <: \Delta''$.*

Lemma 16. *If $\Delta_1 <: \Delta_2$ and $\Delta_3 <: \Delta_4$, then $\Delta_1 \cup \Delta_3 <: \Delta_2 \cup \Delta_4$.*

Lemma 17. *If $\Delta <: \Delta'$ and $\Delta' <: \Delta''$, then $\Delta <: \Delta''$.*

Proof. All cases are very easy, except for the following one:

Let $\mathsf{Inv}^{c''}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*_{call}); \hat{v}^*; \hat{k}) \in \Delta$, $\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*_{call}); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta'$, $\mathsf{LState}_{c,m,pc}((\hat{\lambda}''_t, \hat{u}''^*_{call}); \hat{v}''^*; \hat{h}''; \hat{k}'') \in \Delta''$. Assume that:
$$\mathsf{Inv}^{c''}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*_{call}); \hat{v}^*; \hat{k}) \sqsubseteq^{\Delta'}_{Inv} \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*_{call}); \hat{v}'^*; \hat{h}'; \hat{k}') \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}''_t, \hat{u}''^*_{call}); \hat{v}''^*; \hat{h}''; \hat{k}'')$$

We want to prove that:

$$\mathsf{Inv}^{c''}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*_{call}); \hat{v}^*; \hat{k}) \sqsubseteq^{\Delta''}_{Inv} \mathsf{LState}_{c,m,pc}((\hat{\lambda}''_t, \hat{u}''^*_{call}); \hat{v}''^*; \hat{h}''; \hat{k}'')$$

To this end we need to prove that the following four conditions holds:

- $\hat{\lambda}_t, \hat{u}^*_{call} \sqsubseteq_{Seq} \hat{\lambda}''_t, \hat{u}''^*_{call}$: follows directly from transitivity of $\sqsubseteq_{Seq}$
- $\hat{v}^* \sqsubseteq_{Seq} \hat{v}''^*$: follows directly from transitivity of $\sqsubseteq_{Seq}$
- $\hat{k} \sqsubseteq_{Filter} \hat{k}''$: follows directly from transitivity of $\sqsubseteq_{Filter}$
- $lookup(c, m) = (\_, st^*)$, $st_{pc} = \texttt{invoke}\ r_o\ m'\ \_$ and $\mathsf{Call}^{\Delta''}_{r_o,c'',m'}(\hat{v}''^*; \hat{h}'')$:

  The fact that $lookup(c, m) = (\_, st^*)$, $st_{pc} = \texttt{invoke}\ r_o\ m'\ \_$ is easy. It remains to check that $\mathsf{Call}^{\Delta''}_{r_o,c'',m'}(\hat{v}''^*; \hat{h}'')$. First we know that $\mathsf{Call}^{\Delta'}_{r_o,c'',m'}(\hat{v}'^*; \hat{h}')$ holds, therefore there exist pp' and c' such that:

$$\Big(\overbrace{(\mathsf{NFS}(\mathsf{pp}') \sqsubseteq \hat{v}'_{r_o} \wedge \mathsf{H}(\mathsf{pp}', \{\!|c'; \_|\!\}) \in \Delta')}^{A} \vee \overbrace{\big(\mathsf{FS}(\mathsf{pp}') \sqsubseteq \hat{v}'_{r_o} \wedge \hat{h}'(\mathsf{pp}') = \{\!|c'; \_|\!\}\big)}^{B}\Big) \wedge$$
$$c' \leq c'' \wedge c'' \in \widehat{lookup}(m')$$

  - Assume that A holds: we have $\mathsf{H}(\mathsf{pp}', \{\!|c'; \_|\!\}) \in \Delta'$ and $\mathsf{NFS}(\mathsf{pp}') \sqsubseteq \hat{v}'_{r_o}$. Then since $\Delta' <: \Delta''$ we know that there exists $\mathsf{H}(\mathsf{pp}', \{\!|c'; \_|\!\}) \in \Delta''$. Moreover since $\hat{v}'^* \sqsubseteq_{Seq} \hat{v}''^*$ and $\mathsf{NFS}(\mathsf{pp}') \sqsubseteq \hat{v}'_{r_o}$ we know that $\mathsf{NFS}(\mathsf{pp}') \sqsubseteq \hat{v}''_{r_o}$. Therefore $\mathsf{Call}^{\Delta''}_{r_o,c'',m'}(\hat{v}''^*; \hat{h}'')$ holds.
  - Assume that B holds: we have $\mathsf{FS}(\mathsf{pp}') \sqsubseteq \hat{v}'_{r_o}$ and $\hat{h}'(\mathsf{pp}') = \{\!|c'; \_|\!\}$. First, since $\hat{v}'^* \sqsubseteq_{Seq} \hat{v}''^*$ and $\mathsf{FS}(\mathsf{pp}') \sqsubseteq \hat{v}'_{r_o}$ we know that $\mathsf{FS}(\mathsf{pp}') \sqsubseteq \hat{v}''_{r_o}$. Moreover $\hat{h}'(\mathsf{pp}') = \{\!|c'; \_|\!\}$ and $\hat{h}'(\mathsf{pp}') \neq \bot \implies \hat{h}'(\mathsf{pp}') \sqsubseteq_{Blk} \hat{h}''(\mathsf{pp}')$, hence $\hat{h}''(\mathsf{pp}') = \{\!|c'; \_|\!\}$. Therefore $\mathsf{Call}^{\Delta''}_{r_o,c'',m'}(\hat{v}''^*; \hat{h}'')$ holds.

□

Representation Function

Proposition 6. *For all filter history $K, (\mathsf{lk}^j)_j$ we have:*

- *For any block b, $\beta_{LocBlk}(b, K) \sqsubseteq^{\mathsf{nfs}}_{Blk} \beta_{Blk}(b)$ and $\beta_{Blk}(b) \sqsubseteq^{\mathsf{nfs}}_{Blk} \beta_{LocBlk}(b, K)$.*
- *For any value v, $\beta_{LocVal}(v, K, (\mathsf{lk}^j)_j) \sqsubseteq^{\mathsf{nfs}} \beta_{Val}(v)$ and $\beta_{Val}(v) \sqsubseteq^{\mathsf{nfs}} \beta_{LocVal}(v, K, (\mathsf{lk}^j)_j)$.*

Proof. This is following from the fact that the pre-orders $\sqsubseteq^{\mathsf{nfs}}_{Blk}$ and $\sqsubseteq^{\mathsf{nfs}}$ ignore the flow-sensitive and flow-insensitive annotations of the abstract labels. □

Assumption 7 (Soundness of the Abstract Operations)**.** $\hat{\olessthan}$, $\hat{\odot}$ and $\hat{\oplus}$ are monotonous operators, and soundly over-approximate the concrete operators $\olessthan$, $\odot$ and $\oplus$: for all local heap K, we have:

- $u \olessthan v$ implies that $\beta_{\mathit{LocVal}}(u, K) \mathrel{\hat{\olessthan}} \beta_{\mathit{LocVal}}(v, K)$
- $\beta_{\mathit{LocVal}}(\odot v, K) \sqsubseteq \hat{\odot}\beta_{\mathit{LocVal}}(v, K)$
- $\beta_{\mathit{LocVal}}(u \oplus v, K) \sqsubseteq \beta_{\mathit{LocVal}}(u, K) \mathbin{\hat{\oplus}} \beta_{\mathit{LocVal}}(v, K)$

This carry over to all the representation functions $\beta_{\mathit{LocVal}}(\cdot, K, (\mathsf{lk}^i)_i)$ (with order $\sqsubseteq$) and $\beta_{\mathit{Val}}(\cdot)$ (with order $\sqsubseteq^{\mathsf{nfs}}$):

Proposition 7. *For all concrete values u and v, and for all filter history $K, (\mathsf{lk}^i)_i$ we have:*

- *$u \olessthan v$ implies that $\beta_{\mathit{LocVal}}(u, K, (\mathsf{lk}^i)_i) \mathrel{\hat{\olessthan}} \beta_{\mathit{LocVal}}(v, K, (\mathsf{lk}^i)_i)$ and that $\beta_{\mathit{Val}}(u) \mathrel{\hat{\olessthan}} \beta_{\mathit{Val}}(v)$*
- *$\beta_{\mathit{LocVal}}(\odot v, K, (\mathsf{lk}^i)_i) \sqsubseteq \hat{\odot}\beta_{\mathit{LocVal}}(v, K, (\mathsf{lk}^i)_i)$ and $\beta_{\mathit{Val}}(\odot v) \sqsubseteq^{\mathsf{nfs}} \hat{\odot}\beta_{\mathit{Val}}(v)$*
- *$\beta_{\mathit{LocVal}}(u \oplus v, K, (\mathsf{lk}^i)_i) \sqsubseteq \beta_{\mathit{LocVal}}(u, K, (\mathsf{lk}^i)_i) \mathbin{\hat{\oplus}} \beta_{\mathit{LocVal}}(v, K, (\mathsf{lk}^i)_i)$ and $\beta_{\mathit{Val}}(u \oplus v) \sqsubseteq^{\mathsf{nfs}} \beta_{\mathit{Val}}(u) \mathbin{\hat{\oplus}} \beta_{\mathit{Val}}(v)$*

Proof. Observe that for all filter history $K, (\mathsf{lk}^i)_i$, we have that for all concrete value u:

$$\beta_{\mathit{LocVal}}(u, K, (\mathsf{lk}^i)_i) = \beta_{\mathit{LocVal}}\left(u, \left(\mathsf{pp} \mapsto \Gamma^\infty(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})\right)^*\right)$$

This together with Assumption 7 shows the first point of each item bullet.

The second point of each item bullet follows from the fact that if $\sqsubseteq^{\mathsf{nfs}}$ is coarser than $\sqsubseteq$, and the monotonicity of the abstract operators. We are going to detail the proof of the second item bullet (the other cases work exactly in the same way). Let K be an arbitrary local heap:

$$\begin{array}{ll}
\beta_{\mathit{LocVal}}(\odot v, K) \sqsubseteq \hat{\odot}\beta_{\mathit{LocVal}}(v, K) & \text{by Assumption 7} \\
\beta_{\mathit{LocVal}}(\odot v, K) \sqsubseteq^{\mathsf{nfs}} \hat{\odot}\beta_{\mathit{LocVal}}(v, K) & \text{by Proposition 3} \\
\beta_{\mathit{Val}}(\odot v) \sqsubseteq^{\mathsf{nfs}} \beta_{\mathit{LocVal}}(\odot v, K) \sqsubseteq^{\mathsf{nfs}} \hat{\odot}\beta_{\mathit{LocVal}}(v, K) & \text{by Proposition 6}
\end{array}$$

By Proposition 6 we know that $\beta_{\mathit{LocVal}}(v, K) \sqsubseteq^{\mathsf{nfs}} \beta_{\mathit{Val}}(v)$, therefore by monotonicity of $\hat{\odot}$ we get that $\hat{\odot}\beta_{\mathit{LocVal}}(v, K) \sqsubseteq^{\mathsf{nfs}} \hat{\odot}\beta_{\mathit{Val}}(v)$. This concludes the $\hat{\odot}$ case by showing that:

$$\beta_{\mathit{Val}}(\odot v) \sqsubseteq^{\mathsf{nfs}} \beta_{\mathit{LocVal}}(\odot v, K) \sqsubseteq^{\mathsf{nfs}} \hat{\odot}\beta_{\mathit{LocVal}}(v, K) \sqsubseteq^{\mathsf{nfs}} \hat{\odot}\beta_{\mathit{Val}}(v)$$

□

Assumption 8 (Overriding)**.** If $\mathit{lookup}(c, m) = (c', st^*)$, then $c \leq c'$.

In the next results, let $\Delta \vdash \Delta'$ whenever $\Delta \vdash \mathsf{f}$ for each $\mathsf{f} \in \Delta'$.

Proposition 8. *$\hat{\sqcup}$ is an exact abstraction of $\sqcup^{\mathsf{loc}}$: for all filters lk^1 and lk^2 we have $\beta_{Filter}(\mathsf{lk}^1 \sqcup^{\mathsf{loc}} \mathsf{lk}^2) = \beta_{Filter}(\mathsf{lk}^1) \hat{\sqcup} \beta_{Filter}(\mathsf{lk}^2)$.*

Proposition 9. *For all abstract filter $\hat{k}$, for all abstract values $\hat{u}$ and $\hat{v}$ we have:*

- *if $\hat{u} \sqsubseteq \hat{v}$ then $\mathsf{lift}(\hat{u}; \hat{k}) \sqsubseteq \mathsf{lift}(\hat{v}; \hat{k})$.*
- *if $\hat{u} \sqsubseteq_{Loc} \hat{v}$ then $\mathsf{lift}(\hat{u}; \hat{k}) \sqsubseteq_{Loc} \mathsf{lift}(\hat{v}; \hat{k})$.*
- *if $\hat{u} \sqsubseteq^{\mathsf{nfs}} \hat{v}$ then $\mathsf{lift}(\hat{u}; \hat{k}) \sqsubseteq^{\mathsf{nfs}} \mathsf{lift}(\hat{v}; \hat{k})$.*
- *for all abstract heap $\hat{h}$ and $\hat{h}'$, if $\forall \mathsf{pp}, \hat{h}(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'(\mathsf{pp})$ then:*

$$\forall \mathsf{pp}, \mathsf{hlift}(\hat{h}; \hat{k})(\mathsf{pp}) \sqsubseteq_{Blk} \mathsf{hlift}(\hat{h}'; \hat{k})(\mathsf{pp})$$

Proof. The first point is an assumption made on the $\mathsf{lift}(\cdot;\cdot)$ function, and the second point is trivial. Observe that for all $\hat{u}, \hat{v}$, if $\hat{u} \sqsubseteq_{Loc} \hat{v}$ then $\mathsf{lift}(\hat{u}; \hat{k}) \sqsubseteq_{Loc} \mathsf{lift}(\hat{v}; \hat{k})$. Since $\sqsubseteq^{\mathsf{nfs}}$ is the transitive and reflexive closure of $\sqsubseteq$ and $\sqsubseteq_{Loc}$, this third point is a direct consequence of the first and second points. The fourth point is an easy consequence of $\mathsf{hlift}(\cdot;\cdot)$ definition and of the first point. □

Proposition 10. *$\hat{u} \sqsubseteq^{\mathsf{nfs}} \hat{v}$ implies that $\mathsf{lift}(\hat{u}; 1^*) \sqsubseteq \mathsf{lift}(\hat{v}; 1^*)$.*

Proof. By definition of $\sqsubseteq^{\mathsf{nfs}}$, we know that there exists $(\hat{v}_i)_{i \leq n}, (\hat{v}'_i)_{i \leq n}$ such that:

$$\hat{u} = \hat{v}_1 \sqsubseteq_{Loc} \hat{v}'_1 \sqsubseteq \hat{v}_2 \sqsubseteq_{Loc} \hat{v}'_2 \dots \hat{v}'_{n-1} \sqsubseteq \hat{v}_n \sqsubseteq_{Loc} \hat{v}'_n = \hat{v}$$

By Proposition 9.2, we know that for all $i \leq n$, $\hat{v}_i \sqsubseteq_{Loc} \hat{v}'_i$ implies that $\mathsf{lift}(\hat{v}_i; 1^*) \sqsubseteq_{Loc} \mathsf{lift}(\hat{v}'_i; 1^*)$. Moreover $\mathsf{lift}(\hat{v}_i; 1^*) \sqsubseteq_{Loc} \mathsf{lift}(\hat{v}'_i; 1^*)$ implies that there exists λ such that $\mathsf{lift}(\hat{v}_i; 1^*) = \mathsf{NFS}(\lambda)$ and $\mathsf{lift}(\hat{v}'_i; 1^*) = \mathsf{NFS}(\lambda)$. Therefore $\mathsf{lift}(\hat{v}_i; 1^*) \sqsubseteq \mathsf{lift}(\hat{v}'_i; 1^*)$. By Proposition 9.1, for all $i < n$, $\hat{v}'_i \sqsubseteq \hat{v}_{i+1}$ implies that $\mathsf{lift}(\hat{v}'_i; 1^*) \sqsubseteq \mathsf{lift}(\hat{v}_{i+1}; 1^*)$, hence we have:

$$\mathsf{lift}(\hat{u}; 1^*) = \mathsf{lift}(\hat{v}_1; 1^*) \sqsubseteq \mathsf{lift}(\hat{v}'_1; 1^*) \sqsubseteq \mathsf{lift}(\hat{v}_2; 1^*) \dots \mathsf{lift}(\hat{v}_n; 1^*) \sqsubseteq \mathsf{lift}(\hat{v}'_n; 1^*) = \mathsf{lift}(\hat{v}; 1^*)$$

Which concludes this proof. □

Proposition 11. *If for some i we have :*

$$\Gamma^i((\mathsf{lk}^j)_j, K_a) = \Gamma^{i+k}((\mathsf{lk}'^j)_j, K'_a) \text{ and } \Gamma^{i+1}((\mathsf{lk}^j)_j, K_a) = \Gamma^{i+k+1}((\mathsf{lk}'^j)_j, K'_a)$$

then for all local state L and class c' we have:

$$\beta^{\ell}_{LstInv}(L, i, c', K_a, (\mathsf{lk}^n)_n) = \beta^{\ell}_{LstInv}(L, i+k, c', K'_a, (\mathsf{lk}'^n)_n)$$

Proposition 12. *Let $\Sigma = \ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ and let $\Sigma[\![rhs]\!] = \ell$, then for any $X \in \beta_{Lcnf}(\Sigma)$ with local configuration decomposition $(G, (K_i)_i, K, (\mathsf{lk}^j)_j)$, $v \in dom(H)$ implies that $v \in dom(K)$.*

Proof. By a case analysis on the structure of *rhs*, and using the fact that we have a local configuration decomposition. □

Proposition 13. *Let $(G, (K_i)_i, K, (\mathsf{lk}^j)_j)$ and $(G', (K_i')_i, K', (\mathsf{lk}'^j)_j)$ be two local configuration decomposition of Ω_i such that $K = K'$ and $\forall j, \mathsf{lk}^j = \mathsf{lk}'^j$. Then we have:*

$$\beta_{Frm}(\Omega_i, K, (\mathsf{lk}'^j)_j) = \beta_{Frm}(\Omega_i, K, (\mathsf{lk}^j)_j)$$

Technical Lemmas

Lemma 18 (Right-hand Sides)**.** *Let $\Sigma = \ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha = \langle pp \cdot u^* \cdot st^* \cdot R\rangle :: \alpha_0$, let $\Sigma[\![rhs]\!] = v$, $X \in \beta_{Lcnf}(\Sigma)$ with local configuration decomposition $(G, (K_i)_i, K, (\mathsf{lk}^j)_j)$, let $\Delta :> X$.*

Let $\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t', \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta$ be such that :

$$\beta^{\ell}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^j)_j) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t', \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$$

Then there exists $\hat{v}$ such that $\beta_{LocVal}(v, K) \sqsubseteq \hat{v}$ and $\Delta \cup \langle\!\langle rhs\rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{\mathsf{pp}}(\hat{v})$.

Moreover if rhs is a register r_i then we can take $\hat{v} = \hat{v}_i'$.

Proof. By a case analysis on the structure of *rhs*. We are going to detail the object field look-up case, which is the more complicated one. Let $\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ be such that:

$$\beta^{\ell}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^j)_j) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \tag{5.8}$$

Let $\Sigma[\![r_i]\!] = \ell = p_\lambda$. Since $G, (K_i)_i$ is a heap decomposition of H we know that $\ell \in dom(G)$ or $\ell \in \bigcup_i dom(K_i)$. Moreover by Proposition 12, $\ell \in \bigcup_i dom(K_i)$ implies that $\ell \in dom(K)$. Therefore we are in one of the two following cases:

- $\ell \in dom(G)$: from Equation 5.8 we get that $\hat{v}_i = \beta_{LocVal}(\ell, K) = \mathsf{NFS}(\lambda)$. Moreover since:

$$\beta^{\ell}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^j)_j) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t', \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$$

  we know that $\mathsf{NFS}(\lambda) = \hat{v}_i \sqsubseteq \hat{v}_i'$. We know that there exists o such that $o = H(\ell) = \{\!|c; (f_j \mapsto u_j)^*, f \mapsto v|\!\}$. Since $\Delta :> X$, there exists $\mathsf{H}(\lambda, \{\!|c; (f_i \mapsto \hat{u}_i)^*, f \mapsto \hat{v}_f|\!\}) \in$

Δ such that $\beta_{Val}(v) \sqsubseteq^{\mathsf{nfs}} \hat{v}_f$. Let $\hat{v} = \mathsf{lift}(\hat{v}_f; 1^*)$, then we have $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{\mathsf{pp}}(\hat{v})$ by applying the rule:

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{NFS}(\lambda) \sqsubseteq \hat{v}'_i \wedge \mathsf{H}(\lambda, \{\!|c; (f_i \mapsto \hat{u}_i)^*, f \mapsto \hat{v}_f |\!\}) \implies \mathsf{RHS}_{\mathsf{pp}}(\mathsf{lift}(\hat{v}_f; 1^*))$$

which is in $\langle\!\langle r_i.f \rangle\!\rangle_{pp}$. It remains to check that $\beta_{LocVal}(v, K) \sqsubseteq \hat{v}$: if v is a primitive value then this is trivial. The value v is stored in a field of an object referenced to by ℓ, which is a flow-insensitive location and cannot contain flow-sensitive locations. Therefore v cannot be a flow-sensitive location. If v is a flow-insensitive location $p'_{\lambda'}$ then $\beta_{LocVal}(v, K) = \mathsf{NFS}(\lambda')$, and $\beta_{Val}(v) = \mathsf{NFS}(\lambda')$. Moreover by Proposition 10 we know that $\beta_{Val}(v) \sqsubseteq^{\mathsf{nfs}} \hat{v}_f$ implies that $\mathsf{lift}(\beta_{Val}(v); 1^*) \sqsubseteq^{\mathsf{nfs}} \mathsf{lift}(\hat{v}_f; 1^*)$. Since $\mathsf{lift}(\beta_{Val}(v); 1^*) = \mathsf{NFS}(\lambda') = \beta_{LocVal}(v, K)$, we proved that $\beta_{LocVal}(v, K) \sqsubseteq \hat{v}$.

- $\ell \in dom(K)$: from Equation 5.8 we get that $\hat{v}_i = \beta_{LocVal}(\ell, K) = \mathsf{FS}(\lambda)$. Moreover since:

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \tag{5.9}$$

we know that $\mathsf{FS}(\lambda) = \hat{v}_i \sqsubseteq \hat{v}'_i$. We know that there exists o such that $o = H(\ell) = \{\!|c; (f_j \mapsto u_j)^*, f \mapsto v|\!\}$, hence by definition of β_{LHeap} we get that $\hat{h}(\lambda) = \{\!|c; (f_i \mapsto \hat{u}_i)^*, f \mapsto \hat{v}_f |\!\}$ where $\beta_{LocVal}(v, K) \sqsubseteq \hat{v}_f$. Moreover from Equation 5.9 and the fact that $\hat{h}(\lambda) \neq \bot$ we get that $\hat{h}(\lambda) \sqsubseteq_{Blk} \hat{h}'(\lambda)$, which in turns implies that $\hat{h}'(\lambda) = \{\!|c; (f_i \mapsto \hat{u}''_i)^*, f \mapsto \hat{v}'_f |\!\}$ where $\hat{v}_f \sqsubseteq \hat{v}'_f$. By transitivity of $\sqsubseteq$ we have $\beta_{LocVal}(v, K) \sqsubseteq \hat{v}'_f$.

It just remains to show that $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{\mathsf{pp}}(\hat{v}'_f)$ by applying the following rule, which is in $\langle\!\langle r_i.f \rangle\!\rangle_{pp}$:

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{FS}(\lambda) \sqsubseteq \hat{v}'_i \wedge \hat{h}'(\lambda) = \{\!|c; (f_i \mapsto \hat{u}''_i)^*, f \mapsto \hat{v}'_f |\!\} \implies \mathsf{RHS}_{\mathsf{pp}}(\hat{v}'_f)$$

□

Lemma 19 (Reachability)**.** *For any abstract value $\hat{u}$ and abstract heap $\hat{h}$, there exists an abstract filter $\hat{k}_a$ such that $\vdash \mathsf{Reach}(\hat{u}; \hat{h}; \hat{k}_a)$ and $\hat{k}_a$ is the indicator function of the set of reachable elements starting from $\hat{u}$ in the points-to graph of $\hat{h}$.*

Proof. We define $Reach^n_\lambda$ and $Reach^n_{\hat{v}}$ as follows:

- $Reach^n_{\hat{v}} = \bigcup_{\mathsf{FS}(\lambda') \sqsubseteq \hat{v}} Reach^n_{\lambda'}$
- $Reach^0_\lambda = \{\lambda\}$
- $Reach^{n+1}_\lambda = Reach^n_\lambda \cup \bigcup_i Reach^n_{\hat{v}_i}$ if $\hat{h}(\lambda) = \{\!|c; (f_i \mapsto \hat{v}_i)_i |\!\}$

- $Reach_\lambda^{n+1} = Reach_\lambda^n \cup Reach_{\hat{v}}^n$ if $\hat{h}(\lambda) = \tau[\hat{v}]$
- $Reach_\lambda^{n+1} = Reach_\lambda^n \cup Reach_{\hat{v}}^n$ if $\hat{h}(\lambda) = \{\!|@\tau; \hat{v}|\!\}$

For all λ (resp. $\hat{v}$), $(Reach_\lambda^n)_{n\geq 0}$ (resp. $(Reach_{\hat{v}}^n)_{n\geq 0}$) is a non-decreasing sequence, and the set $Reach_\lambda$ (resp. $Reach_{\hat{v}}$) of reachable elements starting from λ (resp. $\hat{v}$) in the points-to graph of $\hat{h}$ is $Reach_\lambda = \bigcup_{n\geq 0} Reach_\lambda^n$ (resp. $Reach_{\hat{v}} = \bigcup_{n\geq 0} Reach_{\hat{v}}^n$). Moreover since $\hat{h}$ is finite, this limit is reached in a finite number of steps. Therefore there exists N such that $Reach_\lambda = \bigcup_{n\leq N} Reach_\lambda^n$ and $Reach_{\hat{v}} = \bigcup_{n\leq N} Reach_{\hat{v}}^n$.

We define I_n^λ to be the indicator function of $Reach_\lambda^n$, and $I_n^{\hat{v}}$ to be the indicator function of $Reach_{\hat{v}}^n$. We will see I_n^λ and $I_n^{\hat{v}}$ as abstract filters. It is easy to show by induction over n that for all $n \geq 0$, for all λ and for all $\hat{v}$ we have $\vdash \mathsf{Reach}(\mathsf{FS}(\lambda); \hat{h}; I_n^\lambda)$ and $\vdash \mathsf{Reach}(\hat{v}; \hat{h}; I_n^\lambda)$ (observe that the second point uses the fact that there is a finite number of λ). Therefore we have $\vdash \mathsf{Reach}(\hat{u}; \hat{h}; I_N^{\hat{u}})$, where $I_N^{\hat{u}}$ is the indicator function of $Reach_{\hat{u}}^N = Reach_{\hat{u}}$. □

Lemma 20 (Abstract Value Lifting)**.** *Let K and K' be two local heaps, u be a concrete value and S be a set of locations such that $dom(K')\backslash dom(K) = S$ and $u \notin S$.*

Let $\hat{v} = \beta_{LocVal}(u, K)$, $\mathsf{lk}_a = \{(p_\lambda \mapsto 1) \mid p_\lambda \in dom(K) \wedge \exists p'_\lambda \in S\}$ and $\hat{k}_a = \beta_{Filter}(\mathsf{lk}_a)$. Then we have:

$$\beta_{LocVal}(u, K') = \mathsf{lift}(\hat{v}; \hat{k}_a)$$

Proof. If u is a primitive value then this is trivial. Assume $u = \ell = p_\lambda$, then one of the following cases holds:

- $\ell \in dom(K') \cap dom(K)$. Then we have:

  $$\beta_{Loc}(p_\lambda, K') = \mathsf{FS}(\lambda) = \beta_{Loc}(p_\lambda, K)$$

  Moreover since $S \subseteq dom(K')$, we know that $\ell \notin S$. Assume that there exists a location $p'_\lambda \in S$, then since $dom(K')\backslash dom(K) = S$ we know that $p'_\lambda \in dom(K')$. Since $p'_\lambda \in dom(K')$ and $p \neq p'$, this implies that $dom(K')$ contains two locations with the same allocation point, which contradicts the fact that K' is a local heap. Therefore there exists no p' such that $p'_\lambda \in dom(K')$, which in turn implies that implies that $\hat{k}_a(\lambda) = 0$. Hence $\mathsf{lift}(\hat{v}; \hat{k}_a) = \mathsf{lift}(\mathsf{FS}(\lambda); \hat{k}_a) = \mathsf{FS}(\lambda)$, which concludes this case.

- $\ell \in dom(K')\backslash dom(K)$. Then since $dom(K')\backslash dom(K) = S$ we have $\ell \in S$. Besides by hypothesis $\ell \notin S$. Absurd.

- $\ell \in dom(K)\backslash dom(K')$. Therefore $p_\lambda \notin dom(K')$, and since K' is a local heap there exists $p' \neq p$ such that $p'_\lambda \in dom(K')$. Moreover since K is a local heap we have $p'_\lambda \notin dom(K)$. Therefore $p'_\lambda \in S$, which implies that $\hat{k}_a(\lambda) = 1$. By consequence we have:

  $$\beta_{Loc}(p_\lambda, K') = \mathsf{NFS}(\lambda) = \mathsf{lift}(\mathsf{FS}(\lambda); \hat{k}_a) = \mathsf{lift}(\beta_{Loc}(p_\lambda, K'); \hat{k}_a) = \mathsf{lift}(\hat{v}; \hat{k}_a)$$

- $\ell \notin dom(K') \cup dom(K)$. Then we trivially have:

$$\beta_{Loc}(p_\lambda, K') = \mathsf{NFS}(\lambda) = \mathsf{lift}(\mathsf{NFS}(\lambda); \hat{k}_a) = \mathsf{lift}(\beta_{Loc}(p_\lambda, K); \hat{k}_a) = \mathsf{lift}(\hat{v}; \hat{k}_a)$$

$\square$

Lemma 21 (Abstract Local State Lifting)**.** *Let $\Sigma = \ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha = \langle pp \cdot u^* \cdot st^* \cdot R\rangle :: \alpha_0$. Let $(G, (K_i)_i, K, (\mathsf{lk}^j)_j)$ be a local configuration decomposition of Σ, and assume that:*

$$\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$$

Let K' be a local heap, and S a set of locations such that:

- *$dom(K') \backslash dom(K) = S$*
- *$\forall p_\lambda \in S, K'(p_\lambda) = \bot$ and $\forall p_\lambda \notin S, K'(p_\lambda) = K(p_\lambda)$*
- *S is fresh in Σ*

Let $\mathsf{lk}_a = \{(p_\lambda \mapsto 1) \mid p_\lambda \in dom(K) \wedge \exists p'_\lambda \in S\}$ and $\hat{k}_a = \beta_{Filter}(\mathsf{lk}_a)$. Then we have:

1. $\beta^{\ell_r}_{Lst}(\langle c, m, pc+1 \cdot u^* \cdot st^* \cdot R\rangle, K', (\mathsf{lk}_a \sqcup_f \mathsf{lk}^1) :: (\mathsf{lk}^n)_{n>1})) = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a); \mathsf{hlift}(\hat{h}; \hat{k}_a); \hat{k}_a \,\hat{\sqcup}\, \hat{k})$
2. *for all register r_d, concrete value w, locations $p_{\lambda'}$ and memory block b we have:*

$$\begin{aligned} & \beta^{\ell_r}_{Lst}(\langle c, m, pc+1 \cdot u^* \cdot st^* \cdot R[r_d \mapsto w]\rangle, K'[p_{\lambda'} \mapsto b], (\mathsf{lk}_a \sqcup_f \mathsf{lk}^1) :: (\mathsf{lk}^n)_{n>1})) \\ = \quad & \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a)[d \mapsto \beta_{LocVal}(w, K')]; \\ & \mathsf{hlift}(\hat{h}; \hat{k}_a)[\lambda' \mapsto \beta_{LocBlk}(b, K')]; \hat{k}_a \,\hat{\sqcup}\, \hat{k}) \end{aligned}$$

Proof. We are only going to prove 1), as 2) is a rather simple extension of 1). We want to show the four following points:

- We know that $dom(K') \backslash S \subseteq dom(K)$. Moreover by definition of lk_a we know that $S = \{p_\lambda \mid \exists p'_\lambda, \mathsf{lk}_a(p'_\lambda) = 1\}$. Moreover for all ℓ, $\mathsf{lk}(\ell) = 1$ implies that $\ell \in dom(K)$. Hence by Proposition 2.5 we have:

$$\Gamma^2(K, (\mathsf{lk}_j)_{j\geq 1}) = \Gamma^2(K', (\mathsf{lk}_a \sqcup^{\mathsf{loc}} \mathsf{lk}^1) :: (\mathsf{lk}^j)_{j\geq 2})$$

  It is then easy to check that for all $l \leq |u^*|$, we have $\beta_{LocVal}(u_l, K', (\mathsf{lk}_a \sqcup_f \mathsf{lk}^1)) = \beta_{LocVal}(u_l, K, \mathsf{lk}^1) = \hat{u}_l$.

- Let r_k be a register of R. Since S is fresh in Σ, we know that $R(r_k) \notin S$, therefore by Lemma 20 we get that $\beta_{LocVal}(R(r_k), K') = \mathsf{lift}(\hat{v}_k; \hat{k}_a)$.

- Let pp be an allocation point. We want to show that there exists $p_{\mathsf{pp}} \in dom(K')$ such that $\mathsf{hlift}(\hat{h}; \hat{k}_a)(\mathsf{pp}) = \beta_{LocBlk}(K'(p_{\mathsf{pp}}), K'))$. Since K' is a local heap, we know that there exists $\ell = p_{\mathsf{pp}} \in dom(K')$. One of the two following cases holds:
  - $\ell \in S$. By hypothesis, we know that $K'(\ell) = \bot$. Moreover by definition of $\hat{k}_a$ we know that $\hat{k}_a(\mathsf{pp}) = 1$, therefore we have:

    $$\beta_{LocBlk}(K'(\ell), K') = \beta_{LocBlk}(\bot, K') = \bot = \mathsf{hlift}(\hat{h}; \hat{k}_a)(\mathsf{pp})$$

  - $\ell \notin S$. Then by hypothesis we know that $K'(\ell) = K(\ell)$. Assume that $K(\ell) = \{\!|c; (f_i \mapsto u_i)_{i \le n}|\!\}$ (the array and intent cases are similar). Then we have:

    $$\beta_{LocBlk}(K'(\ell), K') = \{\!|c; (f_i \mapsto \beta_{LocVal}(u_i, K'))_{i \le n}|\!\}$$

    Since S is fresh in Σ we know that for all $i \le n$, $u_i \notin S$. Therefore by Lemma 20, for all $i \le n$, we have $\beta_{LocVal}(u_i, K'))_{i \le n} = \mathsf{lift}(\beta_{LocVal}(u_i, K); \hat{k}_a)$. Moreover since $\ell \in dom(K') \backslash S$, we know that $\hat{k}_a(\lambda) = 0$. Therefore:

    $$\begin{aligned}\{\!|c; (f_i \mapsto \beta_{LocVal}(u_i, K'))_{i \le n}|\!\} &= \{\!|c; (f_i \mapsto \mathsf{lift}(\beta_{LocVal}(u_i, K); \hat{k}_a))_{i \le n}|\!\} \\ &= \mathsf{hlift}(\hat{h}; \hat{k}_a)(\lambda)\end{aligned}$$

- $\hat{k}_a \,\hat{\sqcup}\, \hat{k} = \beta_{Filter}(\mathsf{lk}_a \sqcup_f \mathsf{lk}^1)$: this is trivial.

$\square$

We can now state the local preservation lemma, which shows that our abstraction soundly over-approximates the concrete reduction $\rightsquigarrow^*$ between local reduction.

Lemma 22 (Local Preservation). *If $\Sigma \rightsquigarrow^* \Sigma'$ under a given program P, then for any $X \in \beta_{Lcnf}(\Sigma)$ with local configuration decomposition $(G, (K_i)_{i \le n}, K, (\mathsf{lk}^j)_j)$, for any $\Delta :> X$ there exists Δ' and $X' \in \beta_{Lcnf}(\Sigma')$ with local configuration decomposition $(G', (K'_i)_{i \le n}, K', (\mathsf{lk}'^j)_j)$ such that $\forall i, K_i \neq K \implies K_i = K'_i$, $\Delta' :> X'$ and $(\!|P|\!) \cup \Delta \vdash \Delta'$.*

The proof is postponed in Section 5.3.11.

5.3.8 Serialization

To state and prove the global soundness theorem, we are going to need some lemmas to handle heap serialization. Basically these lemmas state that if one serializes only memory blocks that are abstracted in a flow-insensitive fashion, then the serialized versions are still properly over-approximated. The serialization lemmas will be applicable in the global soundness theorem proof because the concrete semantics use serialization for inter-components communications and because our analysis always abstract shared memory blocks in a flow-insensitive fashion.

Lemma 23. *The following statements hold:*

- *if* $\Gamma \vdash ser^H_{Val}(v) = (v', H', \Gamma')$ *then* $\beta_{Val}(v) = \beta_{Val}(v')$
- *if* $\Gamma \vdash ser^H_{Blk}(b) = (b', H', \Gamma')$ *then* $\beta_{Blk}(b) = \beta_{Blk}(b')$

Proof. If $v = prim$, then $v' = prim$ and $\beta_{Val}(v) = \beta_{Val}(v') = \beta_{Prim}(prim)$. If $v = p_\lambda$ then $v' = p'_\lambda$ for some pointer p' and $\beta_{Val}(v) = \mathsf{NFS}(\lambda) = \beta_{Val}(v')$. The second point is a direct consequence of the first one. □

Let $image(\Gamma) = \{\ell' \mid \exists \ell.(\ell \mapsto \ell') \in \Gamma\}$.

Lemma 24. *If* $image(\Gamma) \cap dom(H) = \emptyset$ *then :*

- *if* $\Gamma \vdash ser^H_{Val}(v) = (v', H', \Gamma')$ *then* $image(\Gamma') \cap dom(H) = \emptyset$.
- *if* $\Gamma \vdash ser^H_{Blk}(b) = (b', H', \Gamma')$ *then* $image(\Gamma') \cap dom(H) = \emptyset$.

Proof. We prove the first two points by mutual induction on the proof derivation:

- $\dfrac{}{\Gamma \vdash ser^H_{Val}(prim) = (prim, \cdot, \Gamma)}$: by lemma's hypothesis.

- $\dfrac{(p_\lambda \mapsto p'_\lambda) \in \Gamma}{\Gamma, \vdash ser^H_{Val}(p_\lambda) = (p'_\lambda, \cdot, \Gamma)}$: idem.

- $\dfrac{p'_\lambda \text{ fresh pointer} \qquad \begin{matrix} p_\lambda \notin dom(\Gamma) \\ \Gamma, p_\lambda \mapsto p'_\lambda \vdash ser^H_{Blk}(H(p_\lambda)) = (b, H'', \Gamma') \end{matrix} \qquad H' = H'', p'_\lambda \mapsto b}{\Gamma \vdash ser^H_{Val}(p_\lambda) = (p'_\lambda, H', \Gamma')}$:

  p'_λ is fresh and $image(\Gamma) \cap dom(H) = \emptyset$, therefore $image(\Gamma, p_\lambda \mapsto p'_\lambda) \cap dom(H) = \emptyset$. Hence by induction we know that $image(\Gamma') \cap dom(H) = \emptyset$.

- $\dfrac{\Gamma_0 = \Gamma \qquad \forall i \in [1, n] : \Gamma_{i-1} \vdash ser^H_{Val}(v_i) = (u_i, H_i, \Gamma_i) \qquad H' = H_1, \ldots, H_n}{\Gamma \vdash ser^H_{Blk}(\{\!|c'; (f_i \mapsto v_i)^{i \leq n}|\!\}) = (\{\!|c'; (f_i \mapsto u_i)^{i \leq n}|\!\}, H', \Gamma_n)}$:

  We do an induction over $i \in [0, n]$ to prove that $image(\Gamma_i) \cap dom(H) = \emptyset$: $\Gamma_0 = \Gamma$ hence by lemma's hypothesis $image(\Gamma_0) \cap dom(H) = \emptyset$. Now assume that $image(\Gamma_{i-1}) \cap dom(H) = \emptyset$, then by outer induction hypothesis we have $image(\Gamma_i) \cap dom(H) = \emptyset$.

- Block serialization of arrays and intents works exactly like the object case.

□

Lemma 25. *If* $image(\Gamma) \cap dom(H) = \emptyset$ *then*

- *if* $\Gamma \vdash ser^{H}_{Val}(u) = (u', H', \Gamma')$ *then* $u \notin dom(H)$.
- *if* $\Gamma \vdash ser^{H}_{Blk}(b) = (b', H', \Gamma')$ *then* $(\_ \mapsto b') \not\to_{\mathsf{ref}} H$.

Proof. Simple proof by case analysis on the last (or two last) derivation rule(s) applied. □

Lemma 26. *Let* $G, (K_i)_i$ *be a heap decomposition of* H. *If* $\Delta :> \beta^{G}_{Heap}(H)$ *and* $image(\Gamma) \cap dom(H) = \emptyset$ *then:*

- *if* $\Gamma \vdash ser^{H}_{Val}(v) = (v', H', \Gamma')$ *and* $v \in dom(G)$ *or* v *is a primitive value then* $\Delta :> \beta^{G\cup H'}_{Heap}(H')$
- *if* $\Gamma \vdash ser^{H}_{Blk}(b) = (b', H', \Gamma')$ *and there exists* ℓ *such that* $(\ell \mapsto b) \in G$ *then* $\Delta :> \beta^{G\cup H'}_{Heap}(H')$

Moreover $G \cup H' \cdot (K_i)_i$ *is a heap decomposition of* $H \cup H'$.

Proof. We prove this by mutual induction on the serialization proof derivation.

- $\dfrac{}{\Gamma \vdash ser^{H}_{Val}(prim) = (prim, \cdot, \Gamma)}$: in that case $\beta^{G\cup H'}_{Heap}(H') = \emptyset$
- $\dfrac{(p_\lambda \mapsto p'_\lambda) \in \Gamma}{\Gamma, \vdash ser^{H}_{Val}(p_\lambda) = (p'_\lambda, \cdot, \Gamma)}$: idem here we have $\beta^{G\cup H'}_{Heap}(H') = \emptyset$
- $\dfrac{p'_\lambda \text{ fresh pointer} \qquad \begin{array}{c} p_\lambda \notin dom(\Gamma) \\ \Gamma, p_\lambda \mapsto p'_\lambda \vdash ser^{H}_{Blk}(H(p_\lambda)) = (b, H'', \Gamma') \end{array} \qquad H' = H'', p'_\lambda \mapsto b}{\Gamma \vdash ser^{H}_{Val}(p_\lambda) = (p'_\lambda, H', \Gamma')}$:

  Since $p_\lambda \in dom(G)$ we know that $(p_\lambda \mapsto H(p_\lambda)) \in G$. Therefore by induction we know that $\Delta >: \beta^{G\cup H''}_{Heap}(H'')$. Observe the following:

  $$\beta^{G\cup H'}_{Heap}(H') = \beta^{G\cup H''}_{Heap}(H'') \cup \beta^{G\cup H'}_{Heap}(\nu(p_\lambda) \mapsto b)$$

  Therefore to show that $\Delta :> \beta^{G\cup H'}_{Heap}(H')$ we just need to show that:

  $$\begin{array}{rcll} \Delta & :> & \beta^{G\cup H'}_{Heap}(p'_\lambda \mapsto b) & \\ & = & \{\mathsf{H}(\lambda, \beta_{Blk}(b))\} & \\ & = & \{\mathsf{H}(\lambda, \beta_{Blk}(H(p_\lambda)))\} & \text{by Lemma 23} \\ & = & \beta^{G}_{Heap}(p_\lambda \mapsto H(p_\lambda)) & \text{since } p_\lambda \in dom(G) \end{array}$$

  The last point is implied by the fact that $\Delta :> \beta^{G}_{Heap}(H)$.

  Moreover by induction we know that $G \cup H'' \cdot (K_i)_i$ is a heap decomposition of $H \cup H''$. By Lemma 25 we know that $(\_ \mapsto b) \not\to_{\mathsf{ref}} H$. Moreover p'_λ is a fresh location, therefore it is easy to check that $G \cup H' \cdot (K_i)_i$ is a heap decomposition of $H \cup H'$.

- $$\frac{\Gamma_0 = \Gamma \qquad \forall i \in [1,n] : \Gamma_{i-1} \vdash ser^{H}_{Val}(v_i) = (u_i, H_i, \Gamma_i) \qquad H' = H_1, \ldots, H_n}{\Gamma \vdash ser^{H}_{Blk}(\{\!| c'; (f_i \mapsto v_i)^{i \leq n} |\!\}) = (\{\!| c'; (f_i \mapsto u_i)^{i \leq n} |\!\}, H', \Gamma_n)} :$$

  By applying repeatedly Lemma 24 we get that for all $i \in [1,n]$, $image(\Gamma_i) \cap dom(H) = \emptyset$.

  We know that there exists p_λ such that $(p_\lambda \mapsto \{\!| c'; (f_i \mapsto v_i)^{i \leq n} |\!\}) \in G$. Since $G, (K_i)_i$ is a heap decomposition, we know that for all $i \in [1,n]$, $u_i \in dom(G)$ or u_i is a primitive value. Therefore by induction we know that for all $i \in [1,n]$ $\Delta :> \beta^{G \cup H_i}_{Heap}(H_i)$, which implies that :

  $$\Delta :> \bigcup_{1 \leq i \leq n} \beta^{G \cup H_i}_{Heap}(H_i) = \beta^{G \cup (\bigcup_{1 \leq i \leq n} H_i)}_{Heap}\left(\bigcup_{1 \leq i \leq n} H_i\right)$$

  Moreover the induction hypothesis gives us the fact that for all $i \in [1,n], G \cup H_i \cdot (K_i)_i$ is a heap decomposition of $H \cup H_i$. It is rather simple to check that this implies that $G \cup \left(\bigcup_{1 \leq i \leq n} H_i\right) \cdot (K_i)_i$ is a heap decomposition of $H \left(\bigcup_{1 \leq i \leq n} H_i\right)$.

- Block serialization of arrays and intents works exactly like the object case.

□

5.3.9 Proof of Theorem 3

The global preservation theorem states that our analysis is soundly over-approximating the configuration reduction relation. To prove it, we need an extra assumption on the values that can be given by the Android system to a callback:

Assumption 9. For all configuration decomposition $(G, (K_i, (\mathsf{lk}^{i,j})_j)_i)$, for all location ℓ pointing to an activity object, for all lifecycle state s, for any arbitrary callback state $\alpha_{\ell.s} = \langle \_ \cdot \_ \cdot \_ \cdot R \rangle :: \varepsilon$, the callback register R contains only locations in G.

This is because callback arguments are supplied by the system, and are either primitive values, locations pointing to running Activity objects (which are always global), or locations to Bundle. Bundle are special objects (that we did not model), which are used to save an activity state in order to be able to restore it after it has been destroyed (for example by a screen orientation change). To properly handle callbacks, we would need to model these Bundle objects, and to always abstract them in a flow-insensitive fashion.

Theorem (Global Preservation). If $\Psi \Rightarrow^* \Psi'$ under a given program P, then for any $X \in \beta_{Cnf}(\Psi)$, for any $\Delta :> X$ there exists Δ' and $X' \in \beta_{Cnf}(\Psi')$ such that $\Delta' :> X'$ and $(\!| P |\!) \cup \Delta \vdash \Delta'$.

The proof can be found in Section 5.3.12.

5.3.10 Application to Taint Tracking

Lemma 27 (Taint Abstraction Soundness)**.** *For all configuration $\Psi = \Omega \cdot \Xi \cdot H \cdot S$, for all $\phi = \langle \ell, s, \pi, \gamma, \alpha \rangle \in \Omega$ or $\phi = \langle\!\langle \ell, \ell', \pi, \gamma, \alpha \rangle\!\rangle \in \Xi$, if $\alpha = \langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle :: \_$ then for all register r_k we have that all $\Delta \in \beta_{Cnf}(\Psi)$ with configuration decomposition $(G, (K_i, (\mathsf{lk}^{i,j})_j)_i)$ such that K_n is ϕ's local heap, for all $\Delta' :> \Delta$, there exist two abstract local state facts $\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ and $\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$ such that:*

$$\begin{aligned} & \beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, K_n, (\mathsf{lk}^{n,j})_j) \\ = \;& \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) && \in \Delta \\ \sqsubseteq_R \;& \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') && \in \Delta' \end{aligned}$$

and there exists $\hat{t}$ such that $\mathsf{taint}_\Psi(R(r_k)) \sqsubseteq^{\mathsf{t}} \hat{t}$ and :

$$(\!|P|\!) \cup \Delta' \vdash \mathsf{Taint}(\hat{v}'_i, \hat{h}', \hat{t})$$

Proof. The first part is easy, the only difficulty lies in proving that there exists $\hat{t}$ such that $\mathsf{taint}_\Psi(R(r_k)) \sqsubseteq^{\mathsf{t}} \hat{t}$ and :

$$(\!|P|\!) \cup \Delta' \vdash \mathsf{Taint}(\hat{v}'_i, \hat{h}', \hat{t})$$

We let:

$$\mathsf{taint}^0_\Psi(u) = \begin{cases} t & \text{if } u = prim^t \\ \mathsf{public} & \text{otherwise} \end{cases}$$

For all n we define the following functions:

$$\mathsf{taint}^{n+1}_\Psi(u) = \begin{cases} \sqcup^{\mathsf{t}}{}_i \; \mathsf{taint}^n_\Psi(v_i) & \text{if } u = \ell \wedge H(\ell) = \{\!| c; (f_i \mapsto v_i)^* |\!\} \\ \sqcup^{\mathsf{t}}{}_i \; \mathsf{taint}^n_\Psi(v_i) & \text{if } u = \ell \wedge H(\ell) = \tau[v^*] \\ \sqcup^{\mathsf{t}}{}_i \; \mathsf{taint}^n_\Psi(v_i) & \text{if } u = \ell \wedge H(\ell) = \{\!| @c; (k_i \mapsto v_i)^* |\!\} \\ t & \text{if } u = prim^t \end{cases}$$

We know that $\mathsf{taint}_\Psi(v) = \lim_{n \in \mathbb{N}} \mathsf{taint}^n_\Psi(v)$ and that this limit is reached in a finite number of steps (since the lattice and the heap are finite). We then show by induction on n that for all u, for all $u \sqsubseteq \hat{u}$, there exists $\hat{t}$ such that $\mathsf{taint}^n_\Psi(u) \sqsubseteq^{\mathsf{t}} \hat{t}$ and:

$$(\!|P|\!) \cup \Delta' \vdash \mathsf{Taint}(\hat{u}, \hat{h}', \hat{t})$$

Applying the previous result to $\mathsf{taint}_\Psi(R(r_k))$ conclude this proof. ☐

Lemma 28. *If for all sinks $(c, m) \in Sinks$, $\Delta \in \beta_{Cnf}(\Psi)$:*

$$(\!|P|\!) \cup \Delta \vdash \mathsf{LState}_{c,m,pc}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{Taint}(\hat{v}_i, \hat{h}, \mathsf{secret})$$

is unsatisfiable for each i, then P does not leak from Ψ.

Proof. We prove the contraposition. Assume that a program P satisfies Definition 12, then there exists a configuration Ψ' starting from Ψ where one of the registers r_k in a sink (c, m) contains a secret value. By Theorem 3, for all $\Delta \in \beta_{Cnf}(\Psi)$ there exists $\Delta' \in \beta_{Cnf}(\Psi')$ and $\Delta'' :> \Delta'$ such that $(\!|P|\!) \cup \Delta \vdash \Delta''$.

Let $(G, (K_i, (\mathsf{lk}^{i,j})_j)_i)$ be the configuration decomposition of Δ' and K_n be the local heap of ϕ. By Lemma 27 there exist two abstract local state facts $\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ and $\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$ such that:

$$\begin{aligned} & \beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K_n, (\mathsf{lk}^{n,j})_j) & \\ = \ & \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) & \in \Delta' \\ \sqsubseteq_R \ & \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') & \in \Delta'' \end{aligned}$$

and there exists $\hat{t}$ such that $\mathsf{taint}_{\Psi'}(R(r_k)) \sqsubseteq^{\mathsf{t}} \hat{t}$ and :

$$(\!|P|\!) \cup \Delta'' \vdash \mathsf{Taint}(\hat{v}'_i, \hat{h}', \hat{t})$$

Since $\mathsf{taint}_{\Psi'}(R(r_k)) = \mathsf{secret}$ we know that $\hat{t} = \mathsf{secret}$. This implies that the following formula is derivable:

$$(\!|P|\!) \cup \Delta \vdash \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{Taint}(\hat{v}'_i, \hat{h}, \mathsf{secret})$$

□

5.3.11 Proof of Lemma 22

Proof. If $\Sigma = \Sigma'$ then it suffices to take $\Delta' = \Delta$.

We are just going to prove that this is true if Σ reduces to Σ' in one step. The lemma proof is then obtained by a straightforward induction on the reduction length.

Let $X \in \beta_{Lcnf}(\Sigma)$ with local configuration decomposition $(G, (K_i)_{i\leq n}, K, (\mathsf{lk}^j)_j)$. Let Δ be such that $\Delta :> X$.

Notation Conventions: When not explicitly mentioned otherwise, we let $\Sigma = \ell_r \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$ with $\alpha = L_1 :: \alpha_0$, and let $\Sigma' = \ell_r \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S'$ with $\alpha' = L'_1 :: \alpha'_0$. We also let $L_1 = \langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle$, and $L'_1 = \langle c', m', pc' \cdot u'^* \cdot st'^* \cdot R'\rangle$.

Proof Structure First we are going to describe each case structure:

1. Define $(G', (K'_i)_{i\leq n}, K', (\mathsf{lk}'^j)_j)$ and show that it is a local configuration decomposition of Σ', and that $\forall i, K_i \neq K \implies K_i = K'_i$
2. Define $D_{Call}, D_{Heap}, D_{Stat}, D_{Pact}$ and D_{Pthr} such that:
   - $\beta^{\ell_r}_{Call}(\alpha', K', (\mathsf{lk}'^j)_j) \backslash \beta^{\ell_r}_{Call}(\alpha, K, (\mathsf{lk}^j)_j) \subseteq D_{Call}$

- $\beta^{G'}_{Heap}(H') \backslash \beta^{G}_{Heap}(H) \subseteq D_{Heap}$
- $\beta_{Stat}(S') \backslash \beta_{Stat}(S) \subseteq D_{Stat}$
- $\beta^{\ell_r}_{Pact}(\pi') \backslash \beta^{\ell_r}_{Pact}(\pi) \subseteq D_{Pact}$
- $\beta^{G}_{Pthr}(\gamma') \backslash \beta^{G}_{Pthr}(\gamma) \subseteq D_{Pthr}$

3. Define Δ_{Call}, Δ_{Heap}, Δ_{Stat}, Δ_{Pact} and Δ_{Pthr}.

4. Show that:
   - $D_{Call} <: \Delta \cup \Delta_{Call}$
   - $D_{Heap} <: \Delta_{Heap}$
   - $D_{Stat} <: \Delta_{Stat}$
   - $D_{Pact} <: \Delta_{Pact}$
   - $D_{Pthr} <: \Delta_{Pthr}$

5. Show that:
   - $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$
   - $(\!|P|\!) \cup \Delta \vdash \Delta_{Heap}$
   - $(\!|P|\!) \cup \Delta \vdash \Delta_{Stat}$
   - $(\!|P|\!) \cup \Delta \vdash \Delta_{Pact}$
   - $(\!|P|\!) \cup \Delta \vdash \Delta_{Pthr}$

This is enough to prove the lemma. Indeed by point 1) we know that $X' = \beta^{\ell_r}_{Call}(\alpha', K', (\mathsf{lk}'^j)_j) \cup \beta^{G'}_{Heap}(H') \cup \beta_{Stat}(S') \cup \beta^{\ell_r}_{Pact}(\pi') \cup \beta^{G'}_{Pact}(\gamma')$ is in $\beta_{Lcnf}(\Sigma')$. Let $\Delta' = \Delta \cup \Delta_{Call} \cup \Delta_{Heap} \cup \Delta_{Stat} \cup \Delta_{Pact} \cup \Delta_{Pthr}$.

Using the fact that $\Delta :> X$ and point 4) we get by applying Lemma 16 that $X \cup D_{Call} \cup D_{Heap} \cup D_{Stat} \cup D_{Pact} <: \Delta'$. We know that $X' \subseteq X \cup D_{Call} \cup D_{Heap} \cup D_{Stat} \cup D_{Pact} \cup D_{Pthr}$ by the definitions in point 2). Then by applying Lemma 15 we have $X' <: X \cup D_{Call} \cup D_{Heap} \cup D_{Stat} \cup D_{Pact} \cup D_{Pthr}$, and by applying Lemma 17 we have $X' <: \Delta'$.

The fact that $(\!|P|\!) \cup \Delta \vdash \Delta$ and point 5) implies that $(\!|P|\!) \cup \Delta \vdash \Delta'$, which concludes the proof.

We apply this method to each case, and detail the most important cases in the next following items.

- **(R-Goto):** The rule applied is `goto` pc'.
  1. Let $G', (K'_i)_i = G, (K_i)_i$ and $(\mathsf{lk}'^j)_j = (\mathsf{lk}^j)_j$. It is trivial to check that $(G', (K'_i)_i, K', (\mathsf{lk}'^j)_j)$ is a local configuration decomposition of Σ'.

2. Since $G', (K'_i)_i = G, (K_i)_i$ and $(\mathsf{lk}'^j)_j = (\mathsf{lk}^j)_j$ we know that for all $i \geq 2$ we have $\Gamma^i(K, (\mathsf{lk}^j)_j) = \Gamma^i(K', (\mathsf{lk}'^j)_j)$. Therefore using Proposition 11 we know that for all $i \geq 2$ we have:
$$\beta^{\ell_r}_{LstInv}(\alpha_i, i, \_, K, (\mathsf{lk}^n)_n) = \beta^{\ell_r}_{LstInv}(\alpha_i, i, \_, K', (\mathsf{lk}'^n)_n)$$
Hence $D_{Call} = \beta^{\ell_r}_{Lst}(\langle c, m, pc' \cdot u^* \cdot st^* \cdot R\rangle, K', (\mathsf{lk}'^n)_n)$ satisfies the wanted properties.
3. We know that $\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) =$ $\mathsf{LState}_{c,m,pc}((\hat\lambda_t, \hat u^*); \hat v^*; \hat h; \hat k)$ is in X and $X <: \Delta$. Therefore there exists $\mathsf{LState}_{c,m,pc}((\hat\lambda'_t, \hat u'^*); \hat v'^*; \hat h'; \hat k')$ in Δ such that :
$$\mathsf{LState}_{c,m,pc}((\hat\lambda_t, \hat u^*); \hat v^*; \hat h; \hat k) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat\lambda'_t, \hat u'^*); \hat v'^*; \hat h'; \hat k')$$
Then we define $\Delta_{Call} = \mathsf{LState}_{c,m,pc'}((\hat\lambda'_t, \hat u'^*); \hat v'^*; \hat h'; \hat k')$.
4. We are going to show that $D_{Call} <: \Delta \cup \Delta_{Call}$. First one can check that:
$$\beta^{\ell_r}_{Lst}(\langle c, m, pc' \cdot u^* \cdot st^* \cdot R\rangle, K', (\mathsf{lk}'^n)_n) = \mathsf{LState}_{c,m,pc'}((\hat\lambda_t, \hat u^*); \hat v^*; \hat h; \hat k)$$
The fact that $\mathsf{LState}_{c,m,pc'}((\hat\lambda_t, \hat u^*); \hat v^*; \hat h; \hat k) \sqsubseteq_R \mathsf{LState}_{c,m,pc'}((\hat\lambda'_t, \hat u'^*); \hat v'^*; \hat h'; \hat k')$ is then trivial.
5. We are going to show that $(\!| P |\!) \cup \Delta \vdash \Delta_{Call}$. We know that $(\!| \texttt{goto}\ pc' |\!)_{\mathsf{pp}}$ is included in $(\!| P |\!)$, therefore we have the following rule:
$$\mathsf{LState}_{c,m,pc}((\hat\lambda'_t, \hat u'^*); \hat v'^*; \hat h'; \hat k') \implies \mathsf{LState}_{c,m,pc'}((\hat\lambda'_t, \hat u'^*); \hat v'^*; \hat h'; \hat k')\} \in (\!| P |\!)$$
Moreover $\mathsf{LState}_{c,m,pc}((\hat\lambda'_t, \hat u'^*); \hat v'^*; \hat h'; \hat k')$ is in Δ, therefore by resolution we get:
$$(\!| P |\!) \cup \Delta \vdash \mathsf{LState}_{c,m,pc'}((\hat\lambda'_t, \hat u'^*); \hat v'^*; \hat h'; \hat k')$$
This concludes this proof.

- (R-MoveFld) The rule applied is $\texttt{move}\ r_o.f\ rhs$. We know that there exist two abstract local state facts $\mathsf{LState}_{c,m,pc}((\hat\lambda_t, \hat u^*); \hat v^*; \hat h; \hat k)$ and $\mathsf{LState}_{c,m,pc}((\hat\lambda'_t, \hat u'^*); \hat v'^*; \hat h'; \hat k')$ such that:
$$\begin{aligned}&\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \\ &\quad \mathsf{LState}_{c,m,pc}((\hat\lambda_t, \hat u^*); \hat v^*; \hat h; \hat k) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat\lambda'_t, \hat u'^*); \hat v'^*; \hat h'; \hat k') \in \Delta\end{aligned} \tag{5.11}$$
Let $\Sigma[\![r_o]\!] = \ell''$, we know by Proposition 12 we know that either $\ell'' \in G$ or $\ell'' \in K$.

Case 1: $\ell'' \in G$

By Lemma 18 we know that $\beta_{LocVal}(\Sigma[\![r_o]\!], K) \sqsubseteq \hat v'_{r_o}$. Moreover by applying Lemma 18 to rhs we know that there exists $\hat v''$ such that $\beta_{LocVal}(\Sigma[\![rhs]\!], K) \sqsubseteq \hat v''$ and that $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{\mathsf{pp}}(\hat v'')$. By Lemma 19 there exists $\hat k_a$ such that $\vdash \mathsf{Reach}(\hat v''; \hat h'; \hat k_a)$ and $\hat k_a$ is the indicator function of the set of reachable elements starting from $\hat v''$ in the points-to graph of $\hat h'$.

1. For all $j \neq a$, let $K'_j = K_j$. Let $Reach_a$ be the subset of K defined as follows:

$$Reach_a = \{(p_\lambda \mapsto b) \in K \mid \hat{k}_a(\lambda) = 1\}$$

Let M be the partial mapping containing, for all λ, exactly one entry $(p_\lambda \mapsto \bot)$ if there exists a pointer p'_λ in the domain of $Reach_a$. Moreover we assume that the location p_λ is a fresh location. Let $K' = (K)_{|dom(K)\setminus dom(Reach_a)} \cup M$, and $G' = (G[\ell'' \mapsto G(\ell'')[f \mapsto \Sigma[\![rhs]\!]]]) \cup Reach_a$.
We define lk_a to be the indicator function of $Reach_a$, $\mathsf{lk}'^1 = \mathsf{lk}_a \sqcup^{\mathsf{loc}} \mathsf{lk}^1$ and $(\mathsf{lk}'^j)_{j>1} = (\mathsf{lk}^j)_{j>1}$. One can check that $G', (K'_i)_i$ is a heap decomposition of $H' \cdot S'$. We know that:

$$\begin{aligned} & dom(K')\setminus \left\{p_{\mathsf{pp}} \in dom(K') \mid \exists p', \mathsf{lk}_a(p'_{\mathsf{pp}}) = 1\right\} \\ = \;& dom(K')\setminus \left\{p_{\mathsf{pp}} \in dom(K') \mid \exists p', p'_{\mathsf{pp}} \in dom(Reach_a)\right\} \\ = \;& dom(K')\setminus dom(M) \\ \subseteq \;& dom(K) \end{aligned}$$

Therefore by Proposition 2.5 we get that for all $i \geq 2$, $\Gamma^i(K, (\mathsf{lk}^j)_j) = \Gamma^i(K', (\mathsf{lk}'^j)_j)$. Moreover $dom(K')\setminus dom(K) = dom(M)$, hence by Lemma 11 we know that $(K', (\mathsf{lk}'^j)_j)$ is a filter history of α'.
The fact that $(G', (K'_i)_i, K', (\mathsf{lk}'^j)_j)$ is a local configuration decomposition of Σ' follows easily.

2. Let $L_2, \ldots, L_n$ be such that $\alpha = \langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle :: L_2 :: \cdots :: L_n$. By Proposition 11 we know that for all $j \geq 2$:

$$\beta^{\ell_r}_{LstInv}(L_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta^{\ell_r}_{LstInv}(L_j, j, \_, K', (\mathsf{lk}'^i)_i)$$

One can then show that the following definitions of D_{Call} and D_{Heap} satisfy the wanted properties:

   * $D_{Call} = \beta^{\ell_r}_{Lst}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R\rangle, K', (\mathsf{lk}'^i)_i)$
   * $D_{Heap} = \{\mathsf{H}(\lambda, \hat{b}) \mid H(\ell') = b \wedge \lambda = \beta_{Lab}(\ell') \wedge \hat{b} = \beta_{Blk}(b) \wedge \ell' \in dom(Reach_a)\}$
   $\cup \{\mathsf{H}(\lambda, \hat{b}) \mid \lambda = \beta_{Lab}(\ell'') \wedge \hat{b} = \beta_{Blk}(H(\ell'')[f \mapsto \beta_{Val}(\Sigma[\![rhs]\!])])\}$

3. * $\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}_a \hat{\cup} \hat{k}')$.
   * We define Δ_{Heap} as follows: for all pp, if $\hat{k}_a(\mathsf{pp}) = 1$ and $\hat{h}'(\mathsf{pp}) \neq \bot$ then $\mathsf{H}(\mathsf{pp}, \hat{h}'(\mathsf{pp})) \in \Delta_{Heap}$.
   Moreover we add to Δ_{Heap} the following formula: since $\beta^G_{Heap}(H) <: \Delta$ and $H(\ell'') \neq \bot$ we know that there exists $\mathsf{H}(\lambda_o, \hat{b}_o) \in \Delta$ such that $\beta_{Blk}(H(\ell''))\sqsubseteq^{\mathsf{nfs}}_{Blk}\hat{b}_o$ and $\lambda_o = \beta_{Lab}(\ell'')$. Then we add $H(\lambda_o, \hat{b}_o[f \mapsto \hat{v}''])$ to Δ_{Heap}.

4. We are going to show that:
   * $D_{Call} <: \Delta \cup \Delta_{Call}$: by applying Lemma 21.1 we know that:

   $$\beta^{\ell_r}_{Lst}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R\rangle, K', (\mathsf{lk}'^n)_n)) = \\ \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a); \mathsf{hlift}(\hat{h}; \hat{k}_a); \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k})$$

   Therefore we just have to prove that:

   $$\mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a); \mathsf{hlift}(\hat{h}; \hat{k}_a); \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}) \\ \sqsubseteq_R \ \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}') \quad (5.12)$$

   From Equation (5.11) we know that $\hat{\lambda}_t = \hat{\lambda}'_t$, $\hat{u}^* \sqsubseteq_{Seq} \hat{u}'^*$, $\hat{v}^* \sqsubseteq_{Seq} \hat{v}'^*$, $\hat{k} \sqsubseteq_{Filter} \hat{k}'$ and that $\forall \mathsf{pp}, \hat{h}(\mathsf{pp}) \neq \bot \implies \hat{h}(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'(\mathsf{pp})$.
   To show that Equation (5.12) holds we have four conditions to check:
   - We already know that $\hat{\lambda}_t = \hat{\lambda}'_t$ and $\hat{u}^* \sqsubseteq_{Seq} \hat{u}'^*$.
   - Since $\hat{v}^* \sqsubseteq_{Seq} \hat{v}'^*$, we know by applying Proposition 9 that $\mathsf{lift}(\hat{v}^*; \hat{k}_a) \sqsubseteq_{Seq} \mathsf{lift}(\hat{v}'^*; \hat{k}_a)$.
   - Since $\hat{k} \sqsubseteq_{Filter} \hat{k}'$, it is straightforward to check that $\hat{k}_a \mathbin{\hat{\sqcup}} \hat{k} \sqsubseteq_{Filter} \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}'$.
   - By applying Proposition 9 we know that $\forall \mathsf{pp}, \mathsf{hlift}(\hat{h}; \hat{k}_a)(\mathsf{pp}) \sqsubseteq_{Blk} \mathsf{hlift}(\hat{h}'; \hat{k}_a)(\mathsf{pp})$.
   * $\Delta_{Heap} :> D_{Heap}$:
   - In a first time we are going to show that:

     $$\Delta_{Heap} >: \{\mathsf{H}(\lambda, \hat{b}) \mid H = H', \ell' \mapsto b \\ \wedge \lambda = \beta_{Lab}(\ell') \wedge \hat{b} = \beta_{Blk}(b) \wedge \ell' \in dom(Reach_a)\}$$

     Let $\mathsf{H}(\lambda, \hat{b})$ be an element of the right set of the above relation. We know that there exists b, ℓ' such that $H(\ell') = b$, $\lambda = \beta_{Lab}(\ell')$, $\hat{b} = \beta_{Blk}(b)$ and $\ell' \in dom(Reach_a)$. Besides $\ell' \in Reach_a$ implies that $\hat{k}_a(\lambda) = 1$. We have:

     $$\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$$

     Therefore by definitions of $\beta^{\ell_r}_{Lst}$ and of β_{LHeap} we know that :

     $$\hat{h} = \{(\mathsf{pp} \mapsto \beta_{LocBlk}(K(p_{\mathsf{pp}}), K)) \mid p_{\mathsf{pp}} \in dom(K)\}$$

     Since $(\ell' \mapsto b) \in K$ we have $\hat{h}(\lambda) = \beta_{LocBlk}(b, K)$. Besides by applying Proposition 6 we know that $\beta_{Blk}(b) \sqsubseteq^{\mathsf{nfs}}_{Blk} \beta_{LocBlk}(b, K)$. In summary:

     $$\hat{b} = \beta_{Blk}(b) \sqsubseteq^{\mathsf{nfs}}_{Blk} \beta_{LocBlk}(b, K) = \hat{h}(\lambda) \quad (5.13)$$

     By Equation (5.11) we know that $\forall \mathsf{pp}, \hat{h}(\mathsf{pp}) \neq \bot \implies \hat{h}(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'(\mathsf{pp})$. Since $(\ell' \mapsto b) \in H$, we know that $\hat{h}(\lambda) \neq \bot$, which implies that $\hat{h}(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'(\mathsf{pp})$, and by Proposition 3 we get that$\hat{h}(\mathsf{pp}) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}'(\mathsf{pp})$. Putting Equation (5.13) together with this we get that:

$$\hat{b} \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}(\lambda) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}'(\lambda)$$

We know that $\hat{k}_a(\lambda) = 1$. Besides $\hat{h}(\lambda) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}'(\lambda)$ and $\hat{h}(\lambda) \neq \bot$ implies that $\hat{h}'(\lambda) \neq \bot$. Therefore $\mathsf{H}(\lambda, \hat{h}'(\lambda)) \in \Delta_{Heap}$, which concludes this case by showing that $\mathsf{H}(\lambda, \hat{b}) <: \mathsf{H}(\hat{h}'(\lambda)) \in \Delta_{Heap}$.

· It remains to show that:

$$\{\mathsf{H}(\lambda, \hat{b}) \mid \lambda = \beta_{Lab}(\ell'') \land \hat{b} = \beta_{Blk}(H(\ell'')[f \mapsto \Sigma[\![rhs]\!]])\} <: \Delta_{Heap}$$

Recall that $\beta_{LocVal}(\Sigma[\![rhs]\!], K) \sqsubseteq \hat{v}''$, $\mathsf{H}(\lambda_o, \hat{b}_o) \in \Delta$, $\beta_{Blk}(H(\ell'')) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}_o$, $\lambda_o = \beta_{Lab}(\ell'')$ and $H(\lambda_o, \hat{b}_o[f \mapsto \hat{v}'']) \in \Delta_{Heap}$.
By Proposition 3 we have $\beta_{LocVal}(\Sigma[\![rhs]\!], K) \sqsubseteq^{\mathsf{nfs}} \hat{v}''$, and by Proposition 6 we have $\beta_{Val}(\Sigma[\![rhs]\!]) \sqsubseteq^{\mathsf{nfs}} \beta_{LocVal}(\Sigma[\![rhs]\!], K)$. Therefore by transitivity of $\sqsubseteq^{\mathsf{nfs}}$ we have $\beta_{Val}(\Sigma[\![rhs]\!]) \sqsubseteq^{\mathsf{nfs}} \hat{v}''$. Finally by definition of β_{Blk} we have that:

$$\beta_{Blk}(H(\ell'')[f \mapsto \Sigma[\![rhs]\!]]) = \beta_{Blk}(H(\ell''))[f \mapsto \beta_{Val}(\Sigma[\![rhs]\!])]$$

Applying Proposition 5 to $\beta_{Blk}(H(\ell'')) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}_o$ and $\beta_{Val}(\Sigma[\![rhs]\!]) \sqsubseteq^{\mathsf{nfs}} \hat{v}''$ we get that :

$$\beta_{Blk}(H(\ell''))[f \mapsto \beta_{Val}(\Sigma[\![rhs]\!])] \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}_o[f \mapsto \hat{v}'']$$

Which proves that :

$$\mathsf{H}(\lambda_o, \beta_{Blk}(H(\ell'')[f \mapsto \Sigma[\![rhs]\!]])) <: \mathsf{H}(\lambda_o, \hat{b}_o[f \mapsto \hat{v}'']) <: \Delta_{Heap}$$

This concludes the proof of $D_{Heap} <: \Delta_{Heap}$.

5. ∗ $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$: Recall that $\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta$ and that:

$$\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}_a \hat{\sqcup} \hat{k}')$$

We proved at the beginning of this case that $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{\mathsf{pp}}(\hat{v}'')$ and $\vdash \mathsf{Reach}(\hat{v}''; \hat{h}'; \hat{k}_a)$.
Recall that $\lambda_o = \beta_{Lab}(\ell'')$ and that $\ell'' \in dom(G)$. Lemma 18 applied to ℓ'' and $\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$ gives us that $\mathsf{NFS}(\lambda_o) = \beta_{LocVal}(\ell'', K) \sqsubseteq \hat{v}'_o$. Moreover we know that $\mathsf{H}(\lambda_o, \hat{b}_o) \in \Delta$, hence we can apply the following rule:

$$\mathsf{NFS}(\lambda_o) \sqsubseteq \hat{v}'_o \land \mathsf{H}(\lambda_o, \hat{b}_o) \implies \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda_o); \hat{b}_o)$$

Finally we apply the following rule:

$$\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda_o); \hat{b}_o)$$
$$\wedge \mathsf{Reach}(\hat{v}''; \hat{h}'; \hat{k}_a)$$
$$\implies \mathsf{LState}_{\mathsf{c,m,pc+1}}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}'); \mathsf{hlift}(\hat{h}; \hat{k}'); \hat{k}_a \,\hat{\sqcup}\, \hat{k}')$$

This concludes this case.

* $(\!|P|\!) \cup \Delta \vdash \Delta_{Heap}$: $(\!|P|\!)$ contains the two following rules:

$$\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$$
$$\wedge \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda_o); \hat{b}_o) \wedge \mathsf{Reach}(\hat{v}''; \hat{h}'; \hat{k}_a)$$
$$\wedge \mathsf{H}(\lambda_o, \{\!|c'; (f' \mapsto \hat{u}'')^*, f \mapsto \_|\!\})$$
$$\implies \mathsf{H}(\lambda_o, \{\!|c'; (f' \mapsto \hat{u}'')^*, f \mapsto \hat{v}'')|\!\}) \tag{5.14}$$
$$\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda_o); \hat{b}_o)$$
$$\wedge \mathsf{Reach}(\hat{v}''; \hat{h}'; \hat{k}_a) \wedge \mathsf{Reach}(\hat{v}''; \hat{h}'; \hat{k}_a) \implies \mathsf{LiftHeap}(\hat{h}'; \hat{k}_a) \tag{5.15}$$

Δ_{Heap} is the set defined by:

· for all pp, if $\hat{k}_a(\mathsf{pp}) = 1 \wedge \hat{h}'(\mathsf{pp}) \neq \bot$ then $\mathsf{H}(\mathsf{pp}, \hat{h}'(\mathsf{pp})) \in \Delta_{Heap}$: Let pp satisfying the above conditions. The following rules is in $(\!|P|\!)$:

$$\mathsf{LiftHeap}(\hat{h}'; \hat{k}_a) \wedge \hat{h}'(\mathsf{pp}) = \hat{b} \wedge \hat{k}_a(\mathsf{pp}) = 1 \implies \mathsf{H}(\mathsf{pp}, \hat{b})$$

Rule (5.15) plus the above rule yield $(\!|P|\!) \cup \Delta \vdash \mathsf{H}(\mathsf{pp}, \hat{h}'(\mathsf{pp}))$.

· $\mathsf{H}(\lambda_o, \hat{b}_o[f \mapsto \hat{v}''])$ is in Δ_{Heap}: directly entailed by the rule (5.14).

Case 2: $\ell'' \in K$.

Let $\lambda_o = \beta_{Lab}(\ell'')$, since $\ell'' \in dom(K)$ we have that $\hat{v}_o = \mathsf{FS}(\lambda_o)$. We know from Equation (5.11) that $\hat{v}_o \sqsubseteq \hat{v}'_o$, therefore $\mathsf{FS}(\lambda_o) \sqsubseteq \hat{u}'_o$.

Let b be such that $(\ell'' \mapsto b) \in H$. This implies that $\hat{h}(\lambda_o) \neq \bot$, hence from Equation (5.11) we get that $\hat{h}(\lambda_o) \sqsubseteq_{Blk} \hat{h}'(\lambda_o)$, which in turn implies that there exists $\hat{b}_o = \{\!|c'; (f \mapsto \hat{u}'')|\!\}$ such that $\hat{b}_o = \hat{h}'(\lambda_o)$.

1. Let $K' = K[\ell'' \mapsto K(\ell'')[f \mapsto \Sigma[\![rhs]\!]]]$, $G' = G$ and for all $i \neq a$, $K'_i = K_i$. Let $(\mathsf{lk}'^j)_j = (\mathsf{lk}^j)_j$. Observe that $dom(K) = dom(K')$, and that $(\mathsf{lk}^j)_j = (\mathsf{lk}'^j)_j$, therefore by Proposition 2.4 we know that for all $j \geq 2$, $\Gamma^j(K, (\mathsf{lk}^j)_j) = \Gamma^j(K', (\mathsf{lk}'^j)_j)$. By applying Lemma 11 we get that $(K'_a, (\mathsf{lk}'^j)_j)$ is a filter history of α'. It is then rather easy to check that $(G', (K'_i)_i, K', (\mathsf{lk}'^j)_j)$ is a local configuration decomposition of Σ'.
2. By Proposition 11 we get that for all $j \geq 2$:

$$\beta^{\ell_r}_{LstInv}(\alpha_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta^{\ell_r}_{LstInv}(\alpha_j, j, \_, K', (\mathsf{lk}'^i)_i)$$

It is then easy to check that $D_{Call} = \beta^{\ell_r}_{Lst}(\langle c, m, pc{+}1{\cdot}u^*{\cdot}st^*{\cdot}R\rangle, K', (\mathsf{lk}'^n)_n)$ satisfies the wanted property.

3. By Lemma 18 we know that there exists $\hat{v}''$ such that $\beta_{LocVal}(\Sigma[\![rhs]\!], K) \sqsubseteq \hat{v}''$ and $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{\mathsf{pp}}(\hat{v}'')$. Then we define Δ_{Call} to be the set containing the predicate:
$$\mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \underbrace{\hat{h}'[\lambda_o \mapsto \hat{b}_o[f \mapsto \hat{v}'']]}_{\hat{h}'_1}; \hat{k}')$$

4. We are going to show that $D_{Call} <: \Delta_{Call} \cup \Delta$: first one can check that:
$$\begin{aligned}&\beta^{\ell_r}_{Lst}(\langle c, m, pc+1 \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n)\\ =\ & \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \underbrace{\hat{h}[\lambda_o \mapsto \hat{h}(\lambda_o)[f \mapsto \beta_{LocVal}(\Sigma[\![rhs]\!], K)]]}_{\hat{h}_1}; \hat{k})\end{aligned}$$
We are trying to prove that:
$$\mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}_1; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'_1; \hat{k}')$$
Since we already know that:
$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \quad (5.16)$$
We just need to prove that $\forall \mathsf{pp}, \hat{h}_1(\mathsf{pp}) \neq \bot \implies \hat{h}_1(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'_1(\mathsf{pp})$:
   * Equation 5.16 gives us that $\forall \mathsf{pp}, \hat{h}(\mathsf{pp}) \neq \bot \implies \hat{h}(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'(\mathsf{pp})$, and we know that for all $\mathsf{pp} \neq \lambda_o$ we have $\hat{h}(\mathsf{pp}) = \hat{h}_1(\mathsf{pp})$ and $\hat{h}'(\mathsf{pp}) = \hat{h}'_1(\mathsf{pp})$. Hence $\forall \mathsf{pp} \neq \lambda_o, (\hat{h}_1)(\mathsf{pp}) \neq \bot \implies \hat{h}_1(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'_1(\mathsf{pp})$.
   * $\hat{h}_1(\lambda_o) = \hat{h}(\lambda_o)[f \mapsto \beta_{LocVal}(\Sigma[\![rhs]\!], K)]$ and $\hat{h}'_1(\lambda_o) = \hat{b}_o[f \mapsto \hat{v}'']$. Moreover $\hat{h}(\lambda_o) \neq \bot$, so $\hat{h}(\lambda_o) \sqsubseteq_{Blk} \hat{h}'(\lambda_o) = \hat{b}_o$. Therefore by Proposition 5 we have $\hat{h}_1(\lambda_o) \sqsubseteq_{Blk} \hat{h}'_1(\lambda_o)$.

5. We are going to show that $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$: Recall that $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{\mathsf{pp}}(\hat{v}'')$.
We know that $\vdash \mathsf{FS}(\lambda_o) \sqsubseteq \hat{v}'_o$. Moreover recall that $\hat{b}_o = \{\!|c'; (f \mapsto \hat{u}'')|\!\} = \hat{h}'(\lambda_o)$. Therefore we can apply the following two rules:
$$\mathsf{FS}(\lambda_o) \sqsubseteq \hat{v}'_o \wedge \hat{b}_o = \hat{h}'(\lambda_o) \implies \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}; \mathsf{FS}(\lambda_o); \hat{b}_o)$$
$$\begin{aligned}&\mathsf{RHS}_{\mathsf{pp}}(\hat{v}'') \wedge \mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}; \mathsf{FS}(\lambda_o); \hat{b}_o)\\ &\quad\implies \mathsf{LState}_{\mathsf{c,m,pc+1}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}[\lambda \mapsto \hat{b}_o[f \mapsto \hat{v}'']; \hat{k}')\end{aligned}$$
Which conclude this case.

- (R-Call)
Since Σ reduces to Σ' by applying the rule `invoke` $r_o\ m'\ (r_{i_k})^{k \leq n}$ we know that $\Sigma[\![r_o]\!] = \ell$ and that
$$lookup(type_H(\ell), m') = (c', st'^*) \qquad sign(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$$
$$R' = ((r_j \mapsto \mathbf{0})^{j \leq loc}, r_{loc+1} \mapsto \ell, (r_{loc+1+k} \mapsto \Sigma[\![r_{i_k}]\!])^{k \leq n})$$
$$\alpha' = \langle c', m', 0 \cdot (\Sigma[\![r_{i_k}]\!])^{k \leq n} \cdot st'^* \cdot R'\rangle :: \alpha$$

1. Let $G', (K'_i)_i = G, (K_i)_i$ and $(\mathsf{lk}'^j)_j = (\mathsf{pp} \mapsto 0)^* :: (\mathsf{lk}^l)_l$ (we have one more filter in the list).
   It is easy to check that $G', (K'_i)_i$ is a heap decomposition of $H' \cdot S'$. By Proposition 2.3 we know that for all $j \geq 1$, $\Gamma^j(K, (\mathsf{lk}^j)_j) = \Gamma^{j+1}(K', (\mathsf{lk}'^j)_j)$. Moreover $\Gamma^1(K, (\mathsf{lk}^j)_j) = \Gamma^1(K', (\mathsf{lk}'^j)_j)$.
   Let us show that $(K'_a, (\mathsf{lk}'^j)_j)$ is a filter history α'. The fact that:
   $$\forall i, \forall p_{\mathsf{pp}}, \Big((i = 0 \land p_{\mathsf{pp}} \in dom(K')) \lor \mathsf{lk}'^i(p_{\mathsf{pp}}) = 1\Big) \implies \forall j \neq i, \mathsf{lk}'^j(p_{\mathsf{pp}}) = 0$$
   is rather obvious here, so we are going to focus on showing that:
   $$\Gamma^i(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \implies \Gamma^i(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \notin dom(\alpha'_{|\geq l})$$
   - If $1 < i < l \leq n$. For all pp we have:
     $$\Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \text{ iff}$$
     $$\Gamma^{i-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^{l-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})$$
     Moreover since $(K_a, (\mathsf{lk}^j)_j)$ is a filter history of α we know that:
     $$\Gamma^{i-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^{l-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})$$
     implies
     $$\Gamma^{i-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \notin dom(\alpha_{|\geq l-1})$$
     Since $l > 2$, $\alpha_{|\geq l-1} = \alpha'_{|\geq l}$. Moreover $\Gamma^{i-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) =$ $\Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp})$, so:
     $$\Gamma^{i-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \notin dom(\alpha_{|\geq l-1})$$
     implies
     $$\Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \notin dom(\alpha'_{|\geq l})$$
     Hence we have:
     $$\Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp})$$
     implies
     $$\Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \notin dom(\alpha'_{|\geq l})$$
   - If $i = 1$ and $1 < l \leq n$. For all pp we have:
     $$\Gamma^1(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \text{ iff}$$
     $$\Gamma^1(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^{l-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})$$
     If $l = 2$ then $\Gamma^1(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^{l-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})$ is never true, so the result holds. If $l > 2$ then the same reasoning that we did in the previous case works.

The fact that $(G', (K'_i)_i, K', (\mathsf{lk}'^j)_j)$ is a local configuration decomposition of Σ' follows easily.

2. By Proposition 11 we get that for all $j > 2$:

$$\beta^{\ell_r}_{LstInv}(\alpha_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta^{\ell_r}_{LstInv}(\alpha_j, j+1, \_, K', (\mathsf{lk}'^i)_i)$$

One can then show that the following set D_{Call} satisfies the wanted property:

$$\begin{aligned} D_{Call} = \{&\beta^{\ell_r}_{Lst}(\langle c', m', 0 \cdot (\Sigma[\![r_{i_k}]\!])^{k \leq n} \cdot st'^* \cdot R'\rangle, K', (\mathsf{lk}'^j)_j)\} \\ &\cup \{\beta^{\ell_r}_{LstInv}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, 2, c', K', (\mathsf{lk}'^j)_j)\} \end{aligned}$$

3. We know that there exist $\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta$ and $\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ such that

$$\begin{aligned} &\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \\ \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) &\sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \end{aligned} \tag{5.17}$$

Let $\lambda_o = \beta_{Lab}(\ell)$. Let $\hat{u}^*_{call} = (\hat{u}_{i_k})^{k \leq n}$ and $\hat{u}'^*_{call} = (\hat{u}'_{i_k})^{k \leq n}$. One can check that:

$$\begin{aligned} &\beta^{\ell_r}_{Lst}(\langle c', m', 0 \cdot (\mathbf{0}_k)^{k \leq loc}, (\Sigma[\![r_{i_k}]\!])^{k \leq n} \cdot st'^* \cdot R'\rangle, K', (\mathsf{lk}'^j)_j) = \\ &\qquad \mathsf{LState}_{c',m',0}((\hat{\lambda}_t, \hat{u}^*_{call}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \hat{u}^*_{call}; \hat{h}; 0^*) \end{aligned} \tag{5.18}$$

$$\begin{aligned} &\beta^{\ell_r}_{LstInv}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, 2, c', K', (\mathsf{lk}'^j)_j) = \\ &\qquad \mathsf{Inv}^{c'}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{k}) \end{aligned} \tag{5.19}$$

We define $\Delta_{Call} = \{\mathsf{LState}_{c',m',0}((\hat{\lambda}'_t, \hat{u}'^*_{call}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \hat{u}'^*_{call}; \hat{h}'; 0^*)\}$ $\cup \{\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')\}$

4. We are going to show that $D_{Call} <: \Delta \cup \Delta_{Call}$, or more specifically that:

$$\mathsf{Inv}^{c'}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{k}) \sqsubseteq^{\Delta}_{Inv} \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \tag{5.20}$$

$$\begin{aligned} &\mathsf{LState}_{c',m',0}((\hat{\lambda}_t, \hat{u}^*_{call}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \hat{u}^*_{call}; \hat{h}; 0^*) \sqsubseteq_R \\ &\mathsf{LState}_{c',m',0}((\hat{\lambda}'_t, \hat{u}'^*_{call}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \hat{u}'^*_{call}; \hat{h}'; 0^*) \end{aligned} \tag{5.21}$$

Eq. (5.20)**:** All conditions are trivial consequences of Equation (5.17), except for $\mathsf{Call}^{\Delta \cup \Delta_{Call}}_{r_o, c', m'}(\hat{v}'^*; \hat{h}')$, that we are going to show.
We know by Lemma 18 that $\beta_{LocVal}(\Sigma[\![r_o]\!], K) \sqsubseteq \hat{v}'_o$. The fact that $lookup(type_H(\ell), m') = (c', st'^*)$ implies that $H(\ell) = \{\!|c''; \_|\!\}$ for some class c'' such that $c'' \leq c'$, and that $c' \in \widehat{lookup}(m')$. By definition of $\beta_{Lcnf}(\Sigma)$ we know that if $\ell \in dom(G)$ then there exists $\mathsf{H}(\lambda_o, \{\!|c''; \_|\!\}) \in X$, and if $\ell \in dom(K)$ then $\hat{h}(\lambda_o) = \{\!|c''; \_|\!\}$.

  * If $\ell \in dom(K)$ and $\hat{h}(\lambda_o) = \{\!|c'; \_|\!\}$: then by definition of β_{LocVal} we have $\beta_{LocVal}(\Sigma[\![r_o]\!], K) = \mathsf{FS}(\lambda_o)$, hence $\mathsf{FS}(\lambda_o) \sqsubseteq \hat{v}'_o$. Besides since $\hat{h}(\lambda_o) = \{\!|c''; \_|\!\} \sqsubseteq_{Blk} \hat{h}'(\lambda_o)$ we know that there exists some $\hat{b}$ such that $\hat{h}'(\lambda_o) = \{\!|c''; \hat{b}|\!\}$.
  * If $\ell \in dom(G)$ and $\mathsf{H}(\lambda_o, \{\!|c''; \_|\!\}) \in X$, then there exists $\hat{b}$ such that $\mathsf{H}(\lambda_o, \{\!|c''; \hat{b}|\!\}) \in \Delta$. Besides by definition of β_{LocVal} we have $\beta_{LocVal}(\Sigma[\![r_o]\!], K) = \mathsf{NFS}(\lambda_o)$, which implies that $\hat{v}'_o \sqsubseteq \mathsf{NFS}(\lambda_o)$.

This concludes the proof that $\mathsf{Call}^{\Delta \cup \Delta_{Call}}_{r_o, c', m'}(\hat{v}'^*; \hat{h}')$ holds.

Eq. (5.21)**:** The fact that $0^* \sqsubseteq_{Filter} 0^*$ is trivial. From Equation (5.17) we know that $\forall \mathsf{pp}, \hat{h}(\mathsf{pp}) \neq \bot \implies \hat{h}(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'(\mathsf{pp})$ and that $\hat{u}^* \sqsubseteq_{Seq} \hat{v}^*$. The latter implies that $\hat{u}^*_{call} = (\hat{u}_{i_k})^{k \leq n} \sqsubseteq_{Seq} (\hat{u}'_{i_k})^{k \leq n} = \hat{v}^*_{call}$. This concludes this case.

5. We are going to show that $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$. Since $\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta$ we just need to check that $(\!|P|\!) \cup \Delta \vdash \mathsf{LState}_{c',m',0}((\hat{\lambda}'_t, \hat{u}'^*_{call}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \hat{u}'^*_{call}; \hat{h}'; 0^*)$
As in case 4. we know that one of the following holds:
   – if $\vdash \mathsf{FS}(\lambda_o) \sqsubseteq \hat{v}'_o$ and $\hat{h}'(\lambda_o) = \{\!|c''; \hat{b}|\!\}$ then we can apply the following rule:
$$\mathsf{FS}(\lambda_o) \sqsubseteq \hat{v}'_o \wedge \hat{h}'(\lambda_o) = \{\!|c''; \hat{b}|\!\} \implies \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \mathsf{FS}(\lambda_o); \{\!|c''; \hat{b}|\!\})$$
   – if $\vdash \mathsf{NFS}(\lambda_o) \sqsubseteq \hat{v}'_o$ and $\mathsf{H}(\lambda_o, \{\!|c''; \hat{b}|\!\}) \in \Delta$ then we can apply the rule:
$$\mathsf{NFS}(\lambda_o) \sqsubseteq \hat{v}'_o \wedge \mathsf{H}(\lambda_o, \{\!|c''; \hat{b}|\!\}) \implies \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda_o); \{\!|c''; \hat{b}|\!\})$$
Hence $\Delta \vdash \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \_; \{\!|c''; \hat{b}|\!\})$. Moreover we already knew that $c'' \leq c'$ and that $c' \in \widehat{lookup}(m')$, therefore we can apply the following rule, which is included in $(\!|P|\!)$:
$$\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \_; \{\!|c''; \hat{b}|\!\}) \wedge c'' \leq c' \implies \mathsf{LState}_{c',\mathsf{m}',0}((\hat{\lambda}'_t, \hat{u}'_{call}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \hat{u}'_{call}; \hat{h}'; 0^*)$$
This concludes the proof that $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$.

- (R-Return)

1. Let $G', (K'_i)_i = G, (K_i)_i$ and $(\mathsf{lk}'^j)_j = (\mathsf{lk}_1 \sqcup^{\mathsf{loc}} \mathsf{lk}_2) :: (\mathsf{lk}_i)_{i>2}$.
The fact that $G', (K'_i)_i$ is a heap decomposition of Σ' is easy to prove.
Since $\Sigma \rightsquigarrow \Sigma'$ we know that $\alpha = \langle c, m, pc \cdot v^* \cdot st^* \cdot R \rangle :: \langle c', m', pc' \cdot u'^* \cdot st'^* \cdot R' \rangle :: \alpha_1$ and that $\alpha' = \langle c', m', pc' + 1 \cdot u'^* \cdot st'^* \cdot R'[r_{\mathsf{res}} \mapsto \Sigma[\![r_{\mathsf{res}}]\!]] \rangle :: \alpha_1$. By Proposition 2.2 we know that for all $j > 1$, $\Gamma^{j+1}(K, (\mathsf{lk}^j)_j) = \Gamma^j(K', (\mathsf{lk}'^j)_j)$. Moreover $\Gamma^1(K, (\mathsf{lk}^j)_j) = \Gamma^1(K', (\mathsf{lk}'^j)_j)$.
Let us show that $(K'_a, (\mathsf{lk}'^j)_j)$ is a filter history α'. Let us show that $(K'_a, (\mathsf{lk}'^j)_j)$ is a filter history α'. The fact that:
$$\forall i, \forall p_{\mathsf{pp}}, \Big((i = 0 \wedge p_{\mathsf{pp}} \in dom(K')) \vee \mathsf{lk}'^i(p_{\mathsf{pp}}) = 1\Big) \implies \forall j \neq i, \mathsf{lk}'^j(p_{\mathsf{pp}}) = 0$$

is easy to prove, so we are going to focus on showing that:

$$\Gamma^i(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \implies \Gamma^i(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \not\in dom(\alpha'_{|\geq l})$$

– If $1 < i < l \leq n$, then for all pp we have:

$$\Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp})$$

iff

$$\Gamma^{i+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^{l+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})$$

Moreover since $(K_a, (\mathsf{lk}^j)_j)$ is a filter history of α we know that:

$$\Gamma^{i+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^{l+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})$$

implies

$$]\Gamma^{i+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \not\in dom(\alpha_{|\geq l+1})$$

$\alpha_{|\geq l+1} = \alpha'_{|\geq l}$, and $\Gamma^{i+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) = \Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp})$, hence:

$$\Gamma^{i+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \not\in dom(\alpha_{|>l+1}) \implies \Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \not\in dom(\alpha'_{|\geq l})$$

Therefore we have:

$$\Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \implies \Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \not\in dom(\alpha'_{|\geq l})$$

– If $i = 1$ and $1 < l \leq n$. For all pp we have:

$$\Gamma^1(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp})$$

iff

$$\Gamma^1(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^{l+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})$$

The same reasoning that we did in the previous case works.

The fact that $(G', (K'_i)_i, K', (\mathsf{lk}'^j)_j)$ is a local configuration decomposition of Σ' follows easily.

2. By Proposition 11 we get for all $j \geq 1$:

$$\beta^{\ell_r}_{LstInv}(\alpha_j, j+1, \_, K, (\mathsf{lk}^i)_i) = \beta^{\ell_r}_{LstInv}(\alpha_j, j, \_, K', (\mathsf{lk}'^i)_i)$$

One can then check that the following definition of D_{Call} satisfies the wanted property:

$$D_{Call} = \{\beta^{\ell_r}_{Lst}(\langle c', m', pc' + 1 \cdot u'^* \cdot st'^* \cdot R'[r_{\mathsf{res}} \mapsto \Sigma[\![r_{\mathsf{res}}]\!]]\rangle, K', (\mathsf{lk}'^j)_j)\}$$

3. We know that:

$$\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^j)_j) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*_1); \hat{v}^*_1; \hat{h}_1; \hat{k}_1) \tag{5.22}$$
$$\sqsubseteq_R \quad \mathsf{LState}_{c,m,pc}((\hat{w}'_1, \hat{u}'^*_1); \hat{v}'^*_1; \hat{h}'_1; \hat{k}'_1) \in \Delta$$

$$\beta^{\ell_r}_{LstInv}(\langle c', m', pc' \cdot u'^* \cdot st'^* \cdot R'\rangle, 2, c, K, (\mathsf{lk}^j)_j) =$$
$$\mathsf{Inv}^c_{c',m',pc'}((\hat{\lambda}_t, \hat{u}^*_2); \hat{v}^*_2; \hat{k}_2) \tag{5.23}$$
$$\sqsubseteq^\Delta_{Inv} \quad \mathsf{LState}_{c',m',pc'}((\hat{w}'_2, \hat{u}'^*_2); \hat{v}'^*_2; \hat{h}'_2; \hat{k}'_2) \in \Delta$$

Let $\Delta_{Call} = \{\mathsf{LState}_{c',m',pc'+1}((\hat{w}'_2, \hat{u}'^*_2); \mathsf{lift}(\hat{v}'^*_2; \hat{k}'_1)[\mathsf{res} \mapsto (\hat{v}'^*_1)_{\mathsf{res}}]; \hat{h}'_1; \hat{k}'_1 \hat{\sqcup} \hat{k}'_2)\}$.

4. By Proposition 2.1 and Proposition 2.2 we have $\Gamma^3(K, \mathsf{lk}^1 :: \mathsf{lk}^2) = \Gamma^2(K, \mathsf{lk}^1 \sqcup^{\mathsf{loc}} \mathsf{lk}^2)$, therefore for all $k \leq |u^*_2|$ we have

$$\beta_{LocVal}((u^*_2)_k, K, \mathsf{lk}^1 :: \mathsf{lk}^2) = \beta_{LocVal}((u^*_2)_k, K, \mathsf{lk}^1 \sqcup^{\mathsf{loc}} \mathsf{lk}^2) \tag{5.24}$$

Let r_d be a register different from r_{res}, we want to show that:

$$\beta_{LocVal}(R'(r_d), K) = \mathsf{lift}(\beta_{LocVal}(R'(r_d), K, \mathsf{lk}^1); \hat{k}_1) \tag{5.25}$$

If $R'(r_d)$ is a primitive value then this is trivial, so assume $R'(r_d) = \ell = p_\lambda$. Let $\ell' = p'_\lambda \in dom(K)$ (it exists because K is a local heap). Then we have several cases:

- Case 1: for all p''_λ, we have, $\mathsf{lk}^1(p''_\lambda) = 0$. Then $\Gamma^\infty(K, \mathsf{lk}^1)(\lambda) = \Gamma^\infty(K, \varepsilon)(\lambda) = \ell'$, therefore :

  $$\beta_{LocVal}(\ell, K, \mathsf{lk}^1) = \beta_{Loc}(\ell, K, \mathsf{lk}^1) = \beta_{Loc}(\ell, K) = \beta_{LocVal}(\ell, K)$$

  Moreover $\forall p''_\lambda, \mathsf{lk}^1(p''_\lambda) = 0$ also implies that $\hat{k}_1(\lambda) = 0$, hence :

  $$\mathsf{lift}(\beta_{LocVal}(\ell, K, \mathsf{lk}^1); \hat{k}_1) = \beta_{LocVal}(\ell, K, \mathsf{lk}^1)$$

  This concludes this case.

- Case 2: there exists $\ell'' = p''_\lambda$ such that $\mathsf{lk}^1(p''_\lambda) = 1$. Then $\Gamma^\infty(K, \mathsf{lk}^1)(\lambda) = \ell''$ and $\Gamma^\infty(K, \varepsilon)(\lambda) = \ell'$. We know that $\mathsf{lk}^1(\ell'') = 1$ and that $\ell' \in dom(K)$, therefore since $(K, (\mathsf{lk}^j)_j)$ is a filter history we have $\ell' \neq \ell''$.
  This implies that $\Gamma^2(K, \mathsf{lk}^1)(\lambda) \neq \Gamma^1(K, \varepsilon)(\lambda)$, therefore since $(\mathsf{lk}^i)_i$ is a filter history of Σ we know that $\ell' = \Gamma^\infty(K, \varepsilon)(\lambda) \neq R'(r_d) = \ell$. Hence one of the two following cases holds:

  * $\ell \neq \ell''$. Then $\beta_{LocVal}(\ell, K) = \beta_{LocVal}(\ell, K, \mathsf{lk}^1) = \mathsf{NFS}(\lambda) = \mathsf{lift}(\beta_{LocVal}(\ell, K, \mathsf{lk}^1); \hat{k}_1)$.

* $\ell = \ell''$. Then we have:

$$\beta_{LocVal}(\ell, K, \mathsf{lk}^1) = \mathsf{FS}(\lambda) \text{ and } \beta_{Loc}(\ell, K) = \mathsf{NFS}(\lambda)$$

Moreover $\mathsf{lk}^1(\ell'') = 1$ implies that $\hat{k}_1(\lambda) = 1$, therefore :

$$\mathsf{lift}(\beta_{LocVal}(\ell, K, \mathsf{lk}^1); \hat{k}_1) = \mathsf{lift}(\mathsf{FS}(\lambda); \hat{k}_1) = \mathsf{NFS}(\lambda) = \beta_{Loc}(\ell, K)$$

Using Equation 5.24 and Equation 5.25 one can easily show that:

$$D_{Call} = \mathsf{LState}_{c',m',pc'+1}((\hat{\lambda}_t, \hat{u}_2^*); \mathsf{lift}(\hat{v}_2^*; \hat{k}_1)[\mathsf{res} \mapsto (\hat{v}_1^*)_{\mathsf{res}}]; \hat{h}_1; \\ \beta_{Filter}(\mathsf{lk}^1 \sqcup^{\mathsf{loc}} \mathsf{lk}^2))$$

We want to show that $D_{Call} <: \Delta \cup \Delta_{Call}$: by definition of $\sqsubseteq_R$ we need to check the four following conditions:

- $\hat{\lambda}_t = \hat{w}_2'$ and $\hat{u}_2^* \sqsubseteq_{Seq} \hat{u}_2'^*$: this is trivially implied by Equation (5.23).
- $\forall i, \mathsf{lift}(\hat{v}_2^*; \hat{k}_1)[\mathsf{res} \mapsto (\hat{v}_1^*)_{\mathsf{res}}] \sqsubseteq \mathsf{lift}(\hat{v}_2'^*; \hat{k}_1')[\mathsf{res} \mapsto (\hat{v}_1'^*)_{\mathsf{res}}]$: the case where $i = r_{\mathsf{res}}$ is a trivial consequence of Equation (5.23).
  Assume $i \neq r_{\mathsf{res}}$: from Equation (5.22) we get that $\hat{k}_1 \sqsubseteq_{Filter} \hat{k}_1'$, which implies that $\hat{k}_1 = \hat{k}_1'$. Let $\hat{w} = \mathsf{lift}((\hat{v}_2^*)_i; \hat{k}_1))$ and $\hat{w}' = \mathsf{lift}((\hat{v}_2'^*)_i; \hat{k}_1') = \mathsf{lift}((\hat{v}_2'^*)_i; \hat{k}_1)$. We also know from Equation (5.23) that $\hat{v}_2 \sqsubseteq_{Seq} \hat{v}_2'^*$, therefore by applying Proposition 9 we get that $\hat{w} \sqsubseteq \hat{w}'$.
- $\beta_{Filter}(\mathsf{lk}^1 \sqcup^{\mathsf{loc}} \mathsf{lk}^2) \sqsubseteq_{Filter} \hat{k}_1' \mathbin{\hat{\sqcup}} \hat{k}_2'$: from Equation (5.22), Equation (5.23) and $\beta_{Lst}^{\ell_r}$ definition we know that $\hat{k}_1 = \beta_{Filter}(\mathsf{lk}^1) \sqsubseteq_{Filter} \hat{k}_1'$ and that $\hat{k}_2 = \beta_{Filter}(\mathsf{lk}^2) \sqsubseteq_{Filter} \hat{k}_2'$. By Proposition 8 we know that $\beta_{Filter}(\mathsf{lk}^1 \sqcup^{\mathsf{loc}} \mathsf{lk}^2) = \beta_{Filter}(\mathsf{lk}^1) \mathbin{\hat{\sqcup}} \beta_{Filter}(\mathsf{lk}^2)$. Therefore $\beta_{Filter}(\mathsf{lk}^1 \sqcup^{\mathsf{loc}} \mathsf{lk}^2) = \hat{k}_1 \mathbin{\hat{\sqcup}} \hat{k}_2$. It directly follows that $\hat{k}_1 \mathbin{\hat{\sqcup}} \hat{k}_2 \sqsubseteq_{Filter} \hat{k}_1' \mathbin{\hat{\sqcup}} \hat{k}_2'$.
- $\forall \mathsf{pp}, \hat{h}_1(\mathsf{pp}) \neq \bot \implies \hat{h}_1(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}_1'(\mathsf{pp})$: this is trivially implied by Equation (5.23).

5. We are going to show that $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$. First observe that the following rule is included in $(\!|P|\!)$:

$$\mathsf{LState}_{c,m,pc}((\hat{w}_1', \hat{u}_1'^*); \hat{v}_1'^*; \hat{h}_1'; \hat{k}_1') \implies \mathsf{Res}_{c,m}((\hat{w}_1', \hat{u}_1'^*); (\hat{v}_1'^*)_{\mathsf{res}}; \hat{h}_1'; \hat{k}_1')$$

Therefore $\Delta \vdash \mathsf{Res}_{c,m}((\hat{w}_1', \hat{u}_1'^*); (\hat{v}_1'^*)_{\mathsf{res}}; \hat{h}_1'; \hat{k}_1')$.

By well-formedness of Σ we know that $sign(c', m') = (\tau_i)_{i \leq n} \xrightarrow{loc} \tau$, $st_{pc'}' =$ `invoke` $r_o\ m\ (r_{j_i})_{i \leq n}$ and $u^* = (R'(r_{j_i})))_{i \leq n}$. Moreover from Equation (5.22) we get that $\forall i \leq n, (\hat{u}_1^*)_i = \beta_{LocVal}((u^*)_i, K, \mathsf{lk}^1) \sqsubseteq (\hat{u}_1'^*)_i$, and from Equation (5.23) we get that $\forall k, (\hat{v}_1^*)_k = \beta_{LocVal}((R'(r_k)), K, \mathsf{lk}^1) \sqsubseteq (\hat{v}_2'^*)_k$. Therefore for all $i \leq n$ we have $(\hat{u}_1^*)_i = \beta_{LocVal}((u^*)_i, K, \mathsf{lk}^1) = \beta_{LocVal}((R'(r_{j_i})), K, \mathsf{lk}^1) = (\hat{v}_1^*)_{j_i}$, which implies that $(\hat{u}_1^*)_i \sqsubseteq (\hat{u}_1'^*)_i$ and $(\hat{u}_1^*)_i \sqsubseteq (\hat{v}_2'^*)_{j_i}$. By Proposition 4 we get that $(\hat{v}_2'^*)_{j_i} \sqcap (\hat{u}_1'^*)_i \neq \bot$.

Similarly from Equation (5.22) we get that $\hat{\lambda}_t = \beta_{Val}(\ell_r) = \hat{w}'_1$, and from Equation (5.23) we get that $\hat{\lambda}_t = \beta_{Val}(\ell_r) = \hat{w}'_2$, hence we have $\hat{w}'_1 = \hat{w}'_2$. From Equation (5.23) we get that $\mathsf{Call}^{\Delta}_{r_o,c',m'}(\hat{v}'^*_2; \hat{h}'_2)$ holds. Therefore there exist λ_o and c'' such that:

$$\Big(\overbrace{\big(\mathsf{NFS}(\lambda_o) \sqsubseteq (\hat{v}'^*_2)_o \wedge \mathsf{H}(\lambda_o, \{\!|c''; \_|\!\}) \in \Delta\big)}^{A} \vee$$
$$\overbrace{\Big(\mathsf{FS}(\lambda_o) \sqsubseteq (\hat{v}'^*_2)_o \wedge \hat{h}'_2(\lambda_o) = \{\!|c''; \_|\!\}\Big)}^{B}\Big) \wedge c'' \leq c' \wedge c' \in \widehat{lookup}(m')$$

Hence one of the following cases holds:

– If $\mathsf{FS}(\lambda_o) \sqsubseteq (\hat{v}'^*_2)_o \wedge \hat{h}'_2(\lambda_o) = \{\!|c''; \_|\!\}$ then we can apply the following rule:

$$\mathsf{FS}(\lambda_o) \sqsubseteq (\hat{v}'^*_2)_o \wedge \hat{h}'_2(\lambda_o) = \{\!|c''; \_|\!\} \implies \mathsf{GetBlk}_o(\hat{v}'^*_2; \hat{h}'_2; \mathsf{FS}(\lambda_o); \{\!|c''; \_|\!\})$$

– If $\mathsf{NFS}(\lambda_o) \in (\hat{v}'^*_2)_o \wedge \mathsf{H}(\lambda_o, \{\!|c''; \_|\!\}) \in \Delta$ then we can apply the rule:

$$\mathsf{NFS}(\lambda_o) \sqsubseteq (\hat{v}'^*_2)_o \wedge \mathsf{H}(\lambda_o, \{\!|c''; \_|\!\}) \implies \mathsf{GetBlk}_o(\hat{v}'^*_2; \hat{h}'_2; \mathsf{NFS}(\lambda_o); \{\!|c''; \_|\!\})$$

Therefore we can apply the following rule, which is included in $(\!|P|\!)$:

$$\begin{aligned}
&\mathsf{LState}_{c',m',pc'}((\hat{w}'_2, \hat{u}'^*_2); \hat{v}'^*_2; \hat{h}'_2; \hat{k}'_2) \\
&\wedge \mathsf{GetBlk}_o(\hat{v}'^*_2; \hat{h}'_2; \_; \{\!|c''; \_|\!\}) \wedge c'' \leq c' \\
&\wedge \mathsf{Res}_{c,m}((\hat{w}'_1, \hat{u}'^*_1); (\hat{v}'^*_1)_{\mathsf{res}}; \hat{h}'_1; \hat{k}'_1) \wedge \hat{w}'_1 = \hat{w}'_2 \\
&\wedge \Big(\bigwedge\nolimits_{j \leq n} (\hat{v}'^*_2)_{i_j} \sqcap (\hat{u}'^*_1)_j \neq \bot\Big) \\
\implies &\mathsf{LState}_{c',m',pc'+1}((\hat{w}'_2, \hat{u}'_2); \mathsf{lift}(\hat{v}'^*_2; \hat{k}'_1)[\mathsf{res} \mapsto (\hat{v}'^*_1)_{\mathsf{res}}]; \hat{h}'_1; \hat{k}'_1 \,\hat{\sqcup}\, \hat{k}'_2)
\end{aligned}$$

This shows that $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$.

- (R-NewObj)

$$\text{(R-NewObj)}\quad \frac{\begin{array}{c} o = \{\!|c'; (f_\tau \mapsto \mathbf{0}_\tau)^*|\!\} \\ \ell = p_{c,m,pc} \notin dom(H) \\ H' = H[\ell \mapsto o] \qquad R' = R[r_d \mapsto \ell] \end{array}}{\Sigma, \mathsf{new}\ r_d\ c' \Downarrow \Sigma^+[H \mapsto H', R \mapsto R']}$$

We know that there exist $\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ and $\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$ such that:

$$\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$$
$$\sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta \tag{5.26}$$

By Lemma 19 there exists $\hat{k}_a$ such that $\vdash \mathsf{Reach}(\mathsf{FS}(\mathsf{pp}); \hat{h}'; \hat{k}_a)$ and $\hat{k}_a$ is the indicator function of the set of reachable elements starting from $\mathsf{FS}(\mathsf{pp})$ in the points-to graph of $\hat{h}'$.

1. For all $j \neq a$, let $K'_j = K_j$. Let $Reach_a$ the subset of K defined as follows:

$$Reach_a = \{(p_\lambda \mapsto b) \in K \mid \hat{k}_a(\lambda) = 1\}$$

Let M be the partial mapping containing, for all λ, exactly one entry $(p_\lambda \mapsto \bot)$ if there exists a location p'_λ in the domain of $Reach_a$. Besides we assume that the location p_λ is a fresh location.Let $G' = G \cup Reach_a$, and K' be the local heap defined by:

$$K' = \Big((K)_{|dom(K) \setminus dom(Reach_a)} \cup M \Big) [\ell \mapsto o]$$

Let lk_a be the indicator function of $Reach_a$, $\mathsf{lk}'^1 = \mathsf{lk}_a \sqcup^{\mathsf{loc}} \mathsf{lk}^1$ and $(\mathsf{lk}'^j)_{j>1} = (\mathsf{lk}^j)_{j>1}$.
One can check that $G', (K'_i)_i$ is a heap decomposition of $H' \cdot S'$. Besides we have:

$$\begin{aligned} & dom(K') \setminus \Big\{ p_{\mathsf{pp}} \in dom(K') \mid \exists p', \mathsf{lk}_a(p'_{\mathsf{pp}}) = 1 \Big\} \\ = \quad & dom(K') \setminus \Big\{ p_{\mathsf{pp}} \in dom(K') \mid \exists p', p'_{\mathsf{pp}} \in dom(Reach_a) \Big\} \\ = \quad & dom(K') \setminus (dom(M) \cup \{\ell\}) \\ \subseteq \quad & dom(K) \end{aligned}$$

Hence by Proposition 2.5 we know that for all $i \geq 2$, $\Gamma^i(K, (\mathsf{lk}^j)_j) = \Gamma^i(K', (\mathsf{lk}'^j)_j)$. For all $\ell_x \in dom(\alpha)$, we have by well-formedness of Σ that $\ell_x \in dom(H)$. Therefore since $\ell \notin dom(H)$ we know that $\ell \notin dom(\alpha)$. Moreover $dom(M)$ is a set of fresh locations, therefore $(dom(K') \setminus dom(K)) \cap dom(\alpha_{|>1}) = \emptyset$.
We know that $dom(K') \setminus dom(K) \subseteq dom(M) \cup \{\ell\}$, and $dom(M)$ is a set of fresh locations so it is easy to check that $dom(M) \cap \{\ell' \mid \exists j, \mathsf{lk}^j(\ell') = 1\} = \emptyset$. Besides we are going to assume that ℓ is not only not appearing in Σ, but that it is also not appearing in any of the filters, i.e. $\ell \notin \{\ell' \mid \exists j, \mathsf{lk}^j(\ell') = 1\}$. Basically this means that ℓ is not only a location that was never used yet in the heap H, but also a location that was never introduced as a "dummy" location for proof purposes. We could modify the (R-NewObj) rule, and the configuration decomposition definition, so as to avoid this, but that would make the definitions even lengthier than they are.
Hence we can apply Lemma 11, which shows us that $(K'_a, (\mathsf{lk}'^j)_j)$ is a filter history of α'. The fact that $(G', (K'_i)_i, K', (\mathsf{lk}'^j)_j)$ is a local configuration decomposition of Σ' follows easily.

2. Let $L_2, \dots, L_n$ be such that $\alpha = \langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle :: L_2 :: \cdots :: L_n$. By Proposition 11 we know that for all $j \geq 2$,

$$\beta^{\ell_r}_{LstInv}(L_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta^{\ell_r}_{LstInv}(L_j, j, \_, K', (\mathsf{lk}'^i)_i)$$

One can then show that the following definitions satisfy the wanted property:

– $D_{Call} = \beta^{\ell_r}_{Lst}(\langle c, m, pc+1 \cdot u^* \cdot st^* \cdot R[r_d \mapsto \ell]\rangle, K', (\mathsf{lk}'^i)_i)$

– $D_{Heap} = \{\mathsf{H}(\lambda, \hat{b}) \mid H(\ell') = b \wedge \lambda = \beta_{Lab}(\ell') \wedge \hat{b} = \beta_{Blk}(b) \wedge \ell' \in dom(Reach_a)\}$

3. – $\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})]; \mathsf{hlift}(\hat{h}'; \hat{k}_a)[\mathsf{pp} \mapsto \{\!| c'; (f \mapsto \hat{\mathbf{0}}_\tau)^* |\!\}]; \hat{k}_a \hat{\sqcup} \hat{k}')$

– We define Δ_{Heap} as follows: for all pp, if $\hat{k}_a(\mathsf{pp}) = 1 \wedge \hat{h}'(\mathsf{pp}) \neq \bot$ then $\mathsf{H}(\mathsf{pp}, \hat{h}'(\mathsf{pp})) \in \Delta_{Heap}$.

4. We are going to show that:

– $D_{Call} <: \Delta_{Call}$: by applying Lemma 21.2 we get that:

$$\begin{aligned}&\beta^{\ell_r}_{Lst}(\langle c, m, pc+1 \cdot u^* \cdot st^* \cdot R[r_d \mapsto \ell]\rangle, K', (\mathsf{lk}'^n)_n))\\ &= \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})];\\ &\quad \mathsf{hlift}(\hat{h}; \hat{k}_a)[\mathsf{pp} \mapsto \{\!| c'; (f \mapsto \hat{\mathbf{0}}_\tau)^* |\!\}]; \hat{k}_a \hat{\sqcup} \hat{k})\end{aligned}$$

Therefore we just have to prove that:

$$\begin{aligned}&\mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})];\\ &\quad \overbrace{\mathsf{hlift}(\hat{h}; \hat{k}_a)[\mathsf{pp} \mapsto \{\!| c'; (f \mapsto \hat{\mathbf{0}}_\tau)^* |\!\}]}^{\hat{h}_1}; \hat{k}_a \hat{\sqcup} \hat{k})\\ \sqsubseteq_R\ &\mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})];\\ &\quad \underbrace{\mathsf{hlift}(\hat{h}'; \hat{k}_a)[\mathsf{pp} \mapsto \{\!| c'; (f \mapsto \hat{\mathbf{0}}_\tau)^* |\!\}]}_{\hat{h}'_1}; \hat{k}_a \hat{\sqcup} \hat{k}')\end{aligned} \tag{5.27}$$

From Equation (5.26) we know that $\hat{\lambda}_t = \hat{\lambda}'_t$, $\hat{u}^* \sqsubseteq_{Seq} \hat{u}'^*$, $\hat{v}^* \sqsubseteq_{Seq} \hat{v}'^*$, $\hat{k} \sqsubseteq_{Filter} \hat{k}'$ and that $\forall \mathsf{pp}, \hat{h}(\mathsf{pp}) \neq \bot \implies \hat{h}(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'(\mathsf{pp})$. To show that Equation (5.27) holds we have four conditions to check:

∗ We already know that $\hat{\lambda}_t = \hat{\lambda}'_t$ and $\hat{u}^* \sqsubseteq_{Seq} \hat{u}'^*$.

∗ Since $\hat{v}^* \sqsubseteq_{Seq} \hat{v}'^*$, we know by applying Proposition 9 that $\mathsf{lift}(\hat{v}^*; \hat{k}_a) \sqsubseteq_{Seq} \mathsf{lift}(\hat{v}'^*; \hat{k}_a)$.

∗ Since $\hat{k} \sqsubseteq_{Filter} \hat{k}'$, it is straightforward to check that $\hat{k}_a \hat{\sqcup} \hat{k} \sqsubseteq_{Filter} \hat{k}_a \hat{\sqcup} \hat{k}'$.

∗ For all $\mathsf{pp}' \neq \mathsf{pp}$, $\hat{h}_1(\mathsf{pp}') = \mathsf{hlift}(\hat{h}; \hat{k}_a)(\mathsf{pp}')$ and $\hat{h}'_1(\mathsf{pp}') = \mathsf{hlift}(\hat{h}'; \hat{k}_a)(\mathsf{pp}')$. Therefore by applying Proposition 9 we know that $\hat{h}_1(\mathsf{pp}') \sqsubseteq_{Blk} \hat{h}'_1(\mathsf{pp}')$. Moreover $\hat{h}_1(\mathsf{pp}) = \hat{h}'_1(\mathsf{pp}) = \{\!| c'; (f \mapsto \hat{\mathbf{0}}_\tau)^* |\!\}$, hence we have $\hat{h}_1(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'_1(\mathsf{pp})$.

- $\Delta_{Heap} :> D_{Heap}$: we want to show that:

  $$\Delta_{Heap} >: \{\mathsf{H}(\lambda, \hat{b}) \mid H(\ell') = b \wedge \lambda = \beta_{Lab}(\ell') \wedge \hat{b} = \beta_{Blk}(b) \wedge \ell' \in dom(Reach_a)\}$$

  Let $\mathsf{H}(\lambda, \hat{b})$ be an element of the right set of the above relation. We know that there exists b, ℓ' such that $H(\ell') = b$,$\lambda = \beta_{Lab}(\ell')$,$\hat{b} = \beta_{Blk}(b)$ and $\ell' \in dom(Reach_a)$. Observe that $\ell' \in Reach_a$ implies that $\hat{k}_a(\lambda) = 1$. We have:

  $$\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$$

  Therefore by definitions of $\beta^{\ell_r}_{Lst}$ and of β_{LHeap} we know that :

  $$\hat{h} = \{(\mathsf{pp} \mapsto \beta_{LocBlk}(K(p_{\mathsf{pp}}), K)) \mid p_{\mathsf{pp}} \in dom(K)\}$$

  Since $(\ell' \mapsto b) \in K$ we have $\hat{h}(\lambda) = \beta_{LocBlk}(b, K)$. Besides by applying Proposition 6 we know that $\beta_{Blk}(b) \sqsubseteq^{\mathsf{nfs}}_{Blk} \beta_{LocBlk}(b, K)$. In summary:

  $$\hat{b} = \beta_{Blk}(b) \sqsubseteq^{\mathsf{nfs}}_{Blk} \beta_{LocBlk}(b, K) = \hat{h}(\lambda) \tag{5.28}$$

  By Equation (5.26) we know that $\forall \mathsf{pp}, \hat{h}(\mathsf{pp}) \neq \bot \implies \hat{h}(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'(\mathsf{pp})$. Since $(\ell' \mapsto b) \in dom(H)$, we know that $\hat{h}(\lambda) \neq \bot$, which implies that $\hat{h}(\lambda) \sqsubseteq_{Blk} \hat{h}'(\lambda)$. Putting Equation (5.28) together with this we get that $\hat{b} \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}(\lambda) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}'(\lambda)$.
  We know that $\hat{k}_a(\lambda) = 1$. Besides $\hat{h}(\lambda) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}'(\lambda)$ and $\hat{h}(\lambda) \neq \bot$ implies that $\hat{h}'(\lambda) \neq \bot$. Therefore $\mathsf{H}(\lambda, \hat{h}'(\lambda)) \in \Delta_{Heap}$, which concludes this case.

5. - $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$: recall that $\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta$ and that

   $$\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})]; \mathsf{hlift}(\hat{h}'; \hat{k}_a)$$
   $$[\mathsf{pp} \mapsto \{\!|c'; (f \mapsto \hat{\mathbf{0}}_\tau)^*|\!\}]; \hat{k}_a \hat{\sqcup} \hat{k}')$$

   We already know that $\vdash \mathsf{Reach}(\mathsf{FS}(\mathsf{pp}); \hat{h}'; \hat{k}_a)$, hence we can apply the following rule which is included in $(\!|P|\!)$:

   $$\begin{aligned}&\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{Reach}(\mathsf{FS}(\mathsf{pp}); \hat{h}'; \hat{k}_a)\\ &\implies \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})];\\ &\quad \mathsf{hlift}(\hat{h}'; \hat{k}_a)[\mathsf{pp} \mapsto \{\!|c'; (f \mapsto \hat{\mathbf{0}}_\tau)^*|\!\}]; \hat{k}_a \hat{\sqcup} \hat{k}')\end{aligned}$$

   This concludes this case.
   - $(\!|P|\!) \cup \Delta \vdash \Delta_{Heap}$: we can apply the following rule, which is included in $(\!|P|\!)$:

   $$\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{Reach}(\mathsf{FS}(\mathsf{pp}); \hat{h}'; \hat{k}_a) \implies \mathsf{LiftHeap}(\hat{h}'; \hat{k}_a) \tag{5.29}$$

Δ_{Heap} is the set defined by: for all pp, if $\hat{k}_a(\mathsf{pp}) = 1 \wedge \hat{h}'(\mathsf{pp}) \neq \bot$ then $\mathsf{H}(\mathsf{pp}, \hat{h}'(\mathsf{pp})) \in \Delta_{Heap}$. Let pp be a program point satisfying those conditions. The following rules is in included in $(\!| P |\!)$:

$$\mathsf{LiftHeap}(\hat{h}'; \hat{k}_a'^*) \wedge \hat{h}'(\mathsf{pp}) = \hat{b} \wedge \hat{k}_a(\mathsf{pp}) = 1 \implies \mathsf{H}(\mathsf{pp}, \hat{b})$$

Equation (5.29) plus the above rule yield $(\!| P |\!) \cup \Delta \vdash \mathsf{H}(\mathsf{pp}, \hat{h}'(\mathsf{pp}))$.

- (R-StartThread)

$$\text{(R-StartThread)}\quad \frac{\ell = \Sigma[\![r_i]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\} \qquad \gamma' = \ell :: \gamma}{\Sigma, \texttt{start-thread}\ r_i \Downarrow \Sigma^+[\gamma \mapsto \gamma']}$$

We know that there exist $\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ and $\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t', \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$ such that:

$$\beta_{Lst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t', \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta \quad (5.30)$$

Let $\ell = \Sigma[\![r_i]\!]$, $H(\ell) = b = \{\!|c'; (f \mapsto w)^*|\!\}$. By Assumption 5 we know that with $c' \leq \mathsf{Thread}$. Let K be the local heap of Σ. Also let $\lambda = \beta_{Lab}(\ell)$ and $\hat{b} = \beta_{Blk}(b)$.

Case 1: $(\ell \mapsto b) \in G$.

1. Let $(G', (K_i')_i, K', (\mathsf{lk}'^j)_j) = (G, (K_i)_i, K, (\mathsf{lk}^j)_j)$. This is trivially a local configuration decomposition of Σ'.
2. We take:
   * $D_{Call} = \beta_{Lst}^{\ell_r}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n)$
   * $D_{Pthr} = \mathsf{T}(\lambda, \hat{b})$
3. We define:
   * $\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t', \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$
   * $(\ell \mapsto b) \in G$, therefore $\mathsf{H}(\lambda, \hat{b}) \in X$. Since $X <: \Delta$ we have $\hat{b}'$ such that $\mathsf{H}(\lambda, \hat{b}') \in \Delta$ and $\hat{b} \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}'$. We then define $\Delta_{Pthr} = \mathsf{T}(\lambda, \hat{b}')$.
4. We are going to show that:
   * $D_{Call} <: \Delta_{Call}$. We first check that $D_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$. This case then follows directly from Equation (5.30).
   * $D_{Pthr} <: \Delta_{Pthr}$: this case is trivial since $\hat{b} \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}'$.
5. We know by Lemma 18 that $\beta_{LocVal}(\Sigma[\![r_i]\!], K) \sqsubseteq \hat{v}_i'$. Moreover since $\Sigma[\![r_i]\!] = \ell \in dom(G)$ we have $\beta_{LocVal}(\Sigma[\![r_i]\!], K) = \mathsf{NFS}(\lambda)$. We already knew that $\mathsf{H}(\lambda, \hat{b}') \in \Delta$, therefore we have $\Delta \vdash \mathsf{NFS}(\lambda) \sqsubseteq \hat{v}_i' \wedge \mathsf{H}(\lambda, \hat{b}')$, which implies that $\Delta \vdash \mathsf{GetBlk}_i(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda); \hat{b}')$. Since $\beta_{Blk}(b) =$

$\beta_{Blk}(\{\!|c'; (f \mapsto w)^*|\!\}) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}'$ we know that $\hat{b}' = \{\!|c'; (f \mapsto \hat{w})|\!\}$. Moreover we know that $(\!|P|\!)$ contains the two following rules:

$$\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_i(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda); \{\!|c'; (f \mapsto \hat{w})|\!\})$$
$$\wedge\, c' \leq \mathsf{Thread} \implies \mathsf{T}(\lambda, \{\!|c'; (f \mapsto \hat{w})^*|\!\})$$
$$\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_i(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda); \{\!|c'; (f \mapsto \hat{w})|\!\})$$
$$\wedge\, c' \leq \mathsf{Thread} \implies \mathsf{LState}_{\mathsf{c,m,pc+1}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$$

By applying them we get that $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$ and $(\!|P|\!) \cup \Delta \vdash \Delta_{Pthr}$, which concludes this case.

Case 2: $\ell \in dom(K)$

1. By Lemma 19 there exists $\hat{k}_a$ such that $\vdash \mathsf{Reach}(\mathsf{FS}(\lambda); \hat{h}'; \hat{k}_a)$ and $\hat{k}_a$ is the indicator function of the set of reachable elements starting from $\mathsf{FS}(\lambda)$ in the points-to graph of $\hat{h}'$. For all $j \neq a$, let $K'_j = K_j$, and let $Reach_a$ be the subset of K defined as follows:

$$Reach_a = \{(p_\lambda \mapsto b) \in K \mid \hat{k}_a(\lambda) = 1\}$$

   Let M be the partial mapping containing, for all λ', exactly one entry $(p_{\lambda'} \mapsto \bot)$ if there exists a location $p'_{\lambda'}$ in the domain of $Reach_a$. Besides we assume that the location $p_{\lambda'}$ is a fresh location.
   Let $K' = \Big((K)_{|dom(K)\setminus dom(Reach_a)} \cup M\Big)$ and $G' = G \cup Reach_a$, and we define lk_a to be the indicator function of $Reach_a$, $\mathsf{lk}'^1 = \mathsf{lk}_a \sqcup^{\mathsf{loc}} \mathsf{lk}^1$ and $(\mathsf{lk}'^j)_{j>1} = (\mathsf{lk}^j)_{j>1}$.
   One can check that $G', (K'_i)_i$ is a heap decomposition of $H \cdot S$. As we did in (R-MoveFld), we can apply By Proposition 2.5 to get that for all $i \geq 2$, $\Gamma^i(K, (\mathsf{lk}^j)_j) = \Gamma^i(K', (\mathsf{lk}'^j)_j)$. $dom(M)$ is a set of fresh locations, therefore we can apply Lemma 11, which shows us that $(K'_a, (\mathsf{lk}'^j)_j)$ is a filter history of α'. The fact that $(G', (K'_i)_i, K', (\mathsf{lk}'^j)_j)$ is a local configuration decomposition of Σ' follows easily.
2. Let $L_2, \ldots, L_n$ be such that $\alpha = \langle c, m, pc \cdot u^* \cdot st^* \cdot R]\rangle :: L_2 :: \cdots :: L_n$. By Proposition 11 we know that for all $j \geq 2$:

$$\beta^{\ell_r}_{LstInv}(L_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta^{\ell_r}_{LstInv}(L_j, j, \_, K', (\mathsf{lk}'^i)_i)$$

   One can then show that the following sets satisfy the wanted property:
   * $D_{Call} = \beta^{\ell_r}_{Lst}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R\rangle, K', (\mathsf{lk}'^n)_n))$
   * $D_{Heap} = \{\mathsf{H}(\lambda'', \hat{b}'') \mid H(\ell'') = b'' \wedge \lambda'' = \beta_{Lab}(\ell'') \wedge \hat{b}'' = \beta_{Blk}(b'') \wedge \ell'' \in dom(Reach_a)\}$
   * $D_{Pthr} = \mathsf{T}(\lambda, \hat{b})$
3. We define:
   * $\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}_a \,\hat{\cup}\, \hat{k}')$

  - We define Δ_{Heap} as follows: for all pp, if $\hat{k}_a(\mathsf{pp}) = 1 \wedge \hat{h}'(\mathsf{pp}) \neq \bot$ then $\mathsf{H}(\mathsf{pp}, \hat{h}'(\mathsf{pp})) \in \Delta_{Heap}$.
  - $\ell \in dom(K)$, therefore we know that $\hat{h}(\lambda) = \beta_{LocBlk}(b, K) \neq \bot$. From (5.30) and the definition of $\sqsubseteq_R$ we get that $\hat{h}(\lambda) \sqsubseteq_{Blk} \hat{h}'(\lambda)$. We define $\Delta_{Pthr} = \mathsf{T}(\lambda, \hat{h}'(\lambda))$.

4. We are going to show that:
  - $D_{Call} <: \Delta_{Call}$. By applying Lemma 21.1 we get that:

$$\beta^{\ell_r}_{Lst}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R\rangle, K', (\mathsf{lk}'^n)_n)) = \\ \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a); \mathsf{hlift}(\hat{h}; \hat{k}_a); \hat{k}_a \,\hat{\sqcup}\, \hat{k})$$

Therefore we just have to prove that:

$$\mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a); \mathsf{hlift}(\hat{h}; \hat{k}_a); \hat{k}_a \,\hat{\sqcup}\, \hat{k}) \\ \sqsubseteq_R \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}_a \,\hat{\sqcup}\, \hat{k}') \tag{5.31}$$

From Equation (5.30) we know that $\hat{\lambda}_t = \hat{\lambda}'_t$, $\hat{u}^* \sqsubseteq_{Seq} \hat{u}'^*$, $\hat{v}^* \sqsubseteq_{Seq} \hat{v}'^*$, $\hat{k} \sqsubseteq_{Filter} \hat{k}'$ and that $\forall \mathsf{pp}, \hat{h}(\mathsf{pp}) \neq \bot \implies \hat{h}(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'(\mathsf{pp})$. To show that Equation (5.31) holds we have four conditions to check:

    · We already know that $\hat{\lambda}_t = \hat{\lambda}'_t$ and $\hat{u}^* \sqsubseteq_{Seq} \hat{u}'^*$.
    · Since $\hat{v}^* \sqsubseteq_{Seq} \hat{v}'^*$, we know by applying Proposition 9 that $\mathsf{lift}(\hat{v}^*; \hat{k}_a) \sqsubseteq_{Seq} \mathsf{lift}(\hat{v}'^*; \hat{k}_a)$.
    · Since $\hat{k} \sqsubseteq_{Filter} \hat{k}'$, it is straightforward to check that $\hat{k}_a \hat{\sqcup} \hat{k} \sqsubseteq_{Filter} \hat{k}_a \hat{\sqcup} \hat{k}'$.
    · For all pp, by applying Proposition 9 we know that $\mathsf{hlift}(\hat{h}; \hat{k}_a)(\mathsf{pp}) \sqsubseteq_{Blk} \mathsf{hlift}(\hat{h}'; \hat{k}_a)(\mathsf{pp})$.

  - $D_{Heap} <: \Delta_{Heap}$: we want to show that

$$\Delta_{Heap} >: \{\mathsf{H}(\lambda'', \hat{b}'') \mid H(\ell'') = b'' \wedge \lambda'' = \beta_{Lab}(\ell'') \wedge \hat{b}'' = \beta_{Blk}(b'') \\ \wedge \ell'' \in dom(Reach_a)\}$$

Let $\mathsf{H}(\lambda, \hat{b})$ be an element of the right set of the above relation. We know that there exists b'', ℓ'' such that $H(\ell'') = b''$,$\lambda'' = \beta_{Lab}(\ell'')$,$\hat{b}'' = \beta_{Blk}(b'')$ and $\ell'' \in dom(Reach_a)$. Besides $\ell'' \in Reach_a$ implies that $\hat{k}_a(\lambda'') = 1$. We have:

$$\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$$

Therefore by definitions of $\beta^{\ell_r}_{Lst}$ and of β_{LHeap} we know that :

$$\hat{h} = \{(\mathsf{pp} \mapsto \beta_{LocBlk}(K(p_{\mathsf{pp}}), K)) \mid p_{\mathsf{pp}} \in dom(K)\}$$

Since $(\ell'' \mapsto b'') \in K$ we have $\hat{h}_{\lambda''} = \beta_{LocBlk}(b'', K)$. Besides by applying Proposition 6 we know that $\beta_{Blk}(b'') \sqsubseteq^{\mathsf{nfs}}_{Blk} \beta_{LocBlk}(b'', K)$. In summary:

$$\hat{b}'' = \beta_{Blk}(b'') \sqsubseteq^{\mathsf{nfs}}_{Blk} \beta_{LocBlk}(b'', K) = \hat{h}(\lambda'') \tag{5.32}$$

From Equation (5.30) we get that $\forall \mathsf{pp}, \hat{h}(\mathsf{pp}) \neq \bot \implies \hat{h}(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'(\mathsf{pp})$. Since $(\ell'' \mapsto b'') \in H$, we know that $\hat{h}(\lambda'') \neq \bot$, which implies that $\hat{h}(\lambda'') \sqsubseteq_{Blk} \hat{h}'(\lambda'')$. Putting Equation (5.32) together with this we get that $\hat{b}'' \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}(\lambda'') \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}'(\lambda'')$.
We know that $\hat{k}_a(\lambda'') = 1$. Besides $\hat{h}(\lambda'') \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}'(\lambda'')$ and $\hat{h}(\lambda'') \neq \bot$ implies that $\hat{h}'(\lambda'') \neq \bot$. Therefore $\mathsf{H}(\lambda'', \hat{h}'(\lambda'')) \in \Delta_{Heap}$, which concludes this case.

* $\ell \in dom(K)$, therefore $\hat{h}(\lambda) = \beta_{LocBlk}(b, K) \neq \bot$. Hence by Equation (5.30) we know that $\hat{h}(\lambda) \sqsubseteq_{Blk} \hat{h}'(\lambda)$. By Proposition 6 we know that $\hat{b} = \beta_{Blk}(b) \sqsubseteq^{\mathsf{nfs}}_{Blk} \beta_{LocBlk}(b, K) = \hat{h}(\lambda)$, and by Proposition 3 we get that $\hat{h}(\lambda) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}'(\lambda)$. Therefore $\hat{b} \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}'(\lambda)$, which shows that $D_{Pthr} <: \Delta_{Pthr}$.

5. We are going to show that:
   * $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$: recall that $\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta$ and that:

   $$\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}_a \,\hat{\sqcup}\, \hat{k}')$$

   We know by Lemma 18 that $\beta_{LocVal}(\Sigma[\![r_i]\!], K) \sqsubseteq \hat{v}'_i$. Moreover since $\Sigma[\![r_i]\!] = \ell \in dom(K)$ we have $\mathsf{FS}(\lambda) = \beta_{LocVal}(\Sigma[\![r_i]\!], K)$. We saw previously that $\beta_{Blk}(b) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{h}'(\lambda)$, and since $b = \{\!|c'; (f \mapsto w)^*|\!\}$, we have $\hat{h}'(\lambda) = \{\!|c'; (f \mapsto \hat{w})^*|\!\}$. Hence we have the following abstract heap look-up fact:

   $$\vdash \mathsf{GetBlk}_i(\hat{v}'^*; \hat{h}'; \mathsf{FS}(\lambda); \{\!|c'; (f \mapsto \hat{w})^*|\!\})$$

   Finally $c' \leq \mathsf{Thread}$ and $\vdash \mathsf{Reach}(\mathsf{FS}(\lambda); \hat{h}'; \hat{k}_a)$, which allows us to apply the following rule, which is included in $(\!|P|\!)$:

   $$\begin{gathered}\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \\ \wedge \mathsf{GetBlk}_i(\hat{v}'^*; \hat{h}'; \mathsf{FS}(\lambda); \{\!|c'; (f \mapsto \hat{w})^*|\!\}) \wedge \mathsf{Reach}(\mathsf{FS}(\lambda); \hat{h}'; \hat{k}_a) \\ \wedge c' \leq \mathsf{Thread} \\ \implies \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}' \,\hat{\sqcup}\, \hat{k}_a)\end{gathered}$$

   This concludes this case.
   * $(\!|P|\!) \cup \Delta \vdash \Delta_{Heap}$: We can apply the following rule, which is in $(\!|P|\!)$:

   $$\begin{gathered}\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \\ \wedge \mathsf{GetBlk}_i(\hat{v}'^*; \hat{h}'; \mathsf{FS}(\lambda); \{\!|c'; (f \mapsto \hat{w})^*|\!\}) \wedge \mathsf{Reach}(\mathsf{FS}(\lambda); \hat{h}'; \hat{k}_a) \\ \wedge c' \leq \mathsf{Thread} \implies \mathsf{LiftHeap}(\hat{h}'; \hat{k}_a)\end{gathered} \tag{5.33}$$

Δ_{Heap} is the set defined by: for all pp, if $\hat{k}_a(\mathsf{pp}) = 1 \wedge \hat{h}'(\mathsf{pp}) \neq \bot$ then $\mathsf{H}(\mathsf{pp}, \hat{h}'(\mathsf{pp})) \in \Delta_{Heap}$. Let pp satisfying those conditions. $(\!|P|\!)$ contains the following rule:

$$\mathsf{LiftHeap}(\hat{h}'; \hat{k}_a) \wedge \hat{h}'(\mathsf{pp}) = \hat{b}'' \wedge \hat{k}_a(\mathsf{pp}) = 1 \implies \mathsf{H}(\mathsf{pp}, \hat{b}'')$$

Rule Equation (5.33) plus the above rule yield $(\!|P|\!) \cup \Delta \vdash \mathsf{H}(\mathsf{pp}, \hat{h}'(\mathsf{pp}))$.

* $(\!|P|\!) \cup \Delta \vdash \Delta_{Pthr}$: directly obtained by applying:

$$\begin{aligned}&\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')\\ &\qquad \wedge \mathsf{GetBlk}_i(\hat{v}'^*; \hat{h}'; \mathsf{FS}(\lambda); \{\!|c'; (f \mapsto \hat{w})^*|\!\}) \wedge c' \leq \mathsf{Thread}\\ &\qquad\qquad \implies \mathsf{T}(\lambda, \{\!|c'; (f \mapsto \hat{w})^*|\!\})\end{aligned}$$

- (R-InterruptWait)

$$\text{(R-InterruptWait)}\quad \frac{\begin{array}{c} H(\ell_r) = \{\!|\lambda_r; (f_r \mapsto u_r)^*, \mathsf{inte} \mapsto \mathtt{true}|\!\} \\ p_{c,m,pc} \notin dom(H) \qquad o = \{\!|c_r; (f_r \mapsto u_r)^*, \mathsf{inte} \mapsto \mathtt{false}|\!\} \\ \alpha = \mathsf{waiting}(\_, \_) :: \alpha_0 \qquad o_e = \{\!|\mathsf{IntExcpt}; |\!\} \end{array}}{\Sigma \Downarrow \Sigma[\alpha \mapsto \mathtt{AbNormal}(\alpha_0[r_{\mathsf{excpt}} \mapsto \ell_e]), H \mapsto H[p_{c,m,pc} \mapsto o_e, \ell_r \mapsto o]]}$$

1. Let $\mathsf{pp} = c, m, pc$. Let $G' = G[\ell_r \mapsto o] \cup \{(p_{c,m,pc} \mapsto o_e)\}$ and $((K'_i)_{i \leq n}, K', (\mathsf{lk}'^j)_j) = ((K_i)_{i \leq n}, K, (\mathsf{lk}^j)_j)$. Since $(G, (K_i)_i, K, (\mathsf{lk}^j)_j)$ is a local configuration decomposition of Σ, we know that $\ell_r \in dom(G)$. Besides $p_{c,m,pc}$ is a fresh location, hence it is quite easy to check that $(G', (K'_i)_i, K', (\mathsf{lk}'^j)_j)$ is a local configuration decomposition of Σ', and that $\forall i, K_i \neq K \implies K_i = K'_i$.
2. Let $\alpha = L_1 :: \ldots :: L_n$. By Proposition 2.4 we know that for all $i \geq 2$, $\Gamma^i(K, (\mathsf{lk}^j)_j) = \Gamma^i(K', (\mathsf{lk}'^j)_j)$. Therefore by Proposition 11 we know that for all $j \geq 2$:

   $$\beta^{\ell_r}_{LstInv}(L_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta^{\ell_r}_{LstInv}(L_j, j, \_, K', (\mathsf{lk}'^i)_i)$$

   One can then show that the following definitions satisfy the wanted property:
   - $D_{Call} = \beta^{\ell_r}_{ALst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R[r_{\mathsf{excpt}} \mapsto p_{c,m,pc}]\rangle, K', (\mathsf{lk}'^n)_n))$
   - $D_{Heap} = \{H(\beta_{Lab}(\ell_r), \beta_{Blk}(o))\} \cup \{H(\beta_{Lab}(p_{c,m,pc}), \beta_{Blk}(o_e))\}$
3. We know that there exist $\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ and $\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$ such that:

   $$\begin{aligned}\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R&\\ \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta& \quad (5.34)\end{aligned}$$

   We define:
   - $\Delta_{Call} = \mathsf{AState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*[\mathsf{excpt} \mapsto \mathsf{pp}]; \hat{h}'; \hat{k}')$

– Since $X <: \Delta$ and $\ell_r \in dom(G)$ we know that there exists $\mathsf{H}(\lambda_r, \hat{b}) \in \Delta$ such that $H(\ell_r) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}$. This implies that $\hat{b} = \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}$ and that $(\beta_{Val}(u_r))^* \sqsubseteq^{\mathsf{nfs}}_{Seq} \hat{v}^*_r$ and $\beta_{Val}(\mathtt{true}) \sqsubseteq^{\mathsf{nfs}} \hat{v}_i$. We define :

$$\Delta_{Heap} = \{\mathsf{H}(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \widehat{\mathtt{false}}|\!\})\} \cup \{\mathsf{H}(\mathsf{pp}; \{\!|\mathsf{IntExcpt}; |\!\})\}$$

4. Show that:

   – $D_{Call} <: \Delta_{Call}$: one can check that:

$$\beta^{\ell_r}_{ALst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R[r_{\mathsf{excpt}} \mapsto p_{c,m,pc}]\rangle, K', (\mathsf{lk}'^n)_n)) = \mathsf{AState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*[\mathsf{excpt} \mapsto \mathsf{pp}]; \hat{h}; \hat{k}) \quad (5.35)$$

From Equation (5.34) we know that:

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$$

This implies that:

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*[\mathsf{excpt} \mapsto \mathsf{pp}]; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*[\mathsf{excpt} \mapsto \mathsf{pp}]; \hat{h}'; \hat{k}')$$

Hence by definition of $\sqsubseteq_A$ we have:

$$\mathsf{AState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*[\mathsf{excpt} \mapsto \mathsf{pp}]; \hat{h}; \hat{k}) \sqsubseteq_A \mathsf{AState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*[\mathsf{excpt} \mapsto \mathsf{pp}]; \hat{h}'; \hat{k}')$$

Equation (5.35) and the above relation shows that $D_{Call} <: \Delta_{Call}$.

   – $D_{Heap} <: \Delta_{Heap}$: we know that $(\beta_{Val}(u_r))^* \sqsubseteq^{\mathsf{nfs}}_{Seq} \hat{u}^*_r$. Besides $\beta_{Val}(\mathtt{false}) \sqsubseteq^{\mathsf{nfs}} \widehat{\mathtt{false}}$, therefore we have $\beta_{Blk}(o) \sqsubseteq^{\mathsf{nfs}}_{Blk} \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \widehat{\mathtt{false}}|\!\})$, which in turn implies that :

$$\{H(\beta_{Lab}(\ell_r), \beta_{Blk}(o))\} <: \{\mathsf{H}(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*|\!\}\} \subseteq \Delta_{Heap}$$

The fact that $\{H(\beta_{Lab}(\ell_r), \beta_{Blk}(o_e))\} <: \{\mathsf{H}(\mathsf{pp}; \{\!|\mathsf{IntExcpt}; |\!\})\} \subseteq \Delta_{Heap}$ is trivial.

5. By definition of β_{Lst}, we get from Equation (5.34) that $\hat{\lambda}_t = \beta_{Val}(\ell_r) = \mathsf{NFS}(\lambda_r)$, and that $\hat{\lambda}_t = \hat{\lambda}'_t$. Besides we know that $\mathsf{H}(\lambda_r, \hat{b}) \in \Delta$, where $\hat{b} = \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}$ and $\beta_{Val}(\mathtt{true}) = \widehat{\mathtt{true}} \sqsubseteq^{\mathsf{nfs}} \hat{v}_i$, which implies that $\widehat{\mathtt{true}} \sqsubseteq \hat{v}_i$. Moreover Equation (5.34) gives us that $\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta$, therefore we have :

$$\Delta \vdash \mathsf{LState}_{c,m,pc}((\mathsf{NFS}(\lambda_r), \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{H}(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}) \wedge \widehat{\mathtt{true}} \sqsubseteq \hat{v}_i \quad (5.36)$$

Since Σ is well-formed, and since $L_1 = \mathsf{waiting}(\_,\_)$ we know that $st_{pc} = \texttt{wait}\ \_$. Therefore $(\!|P|\!)$ contains the following rules:

$$\mathsf{LState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_r), \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{H}(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}) \wedge \widehat{\texttt{true}} \sqsubseteq \hat{v}_i \implies \mathsf{AState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_r), \hat{u}'^*); \hat{v}'^*[\mathsf{excpt} \mapsto \mathsf{pp}]; \hat{h}'; \hat{k}') \tag{5.37}$$

$$\mathsf{LState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_r), \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{H}(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}) \wedge \widehat{\texttt{true}} \sqsubseteq \hat{v}_i \implies \mathsf{H}(\lambda_r, \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \widehat{\texttt{false}}|\!\}) \tag{5.38}$$

$$\mathsf{LState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_r), \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{H}(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}) \wedge \widehat{\texttt{true}} \sqsubseteq \hat{v}_i \implies \mathsf{H}(\mathsf{pp}; \{\!|\mathsf{IntExcpt}; |\!\}) \tag{5.39}$$

- $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$: this is trivially implied by Equation (5.36) and Equation (5.37).
- $(\!|P|\!) \cup \Delta \vdash \Delta_{Heap}$: Equation (5.36) and Equation (5.38) gives us that $(\!|P|\!) \cup \Delta \vdash \mathsf{H}(\lambda_r, \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \widehat{\texttt{false}}|\!\})$, and abstract fact $\mathsf{H}(\mathsf{pp}; \{\!|\mathsf{IntExcpt}; |\!\})$ is obtained by Equation (5.39).

• (R-Caught)

$$\text{(R-CAUGHT)}\quad \frac{\ell = \Sigma[\![r_{\mathsf{excpt}}]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*|\!\} \qquad \mathsf{ExcptTable}(c, m, pc, c') = pc' \qquad \alpha' = \langle c, m, pc' \cdot \_ \cdot \_ \cdot R\rangle :: \alpha_0}{\Sigma \Downarrow \Sigma[\alpha \mapsto \alpha']}$$

Here call-stack is abnormal and of the form $\alpha = \texttt{AbNormal}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle :: \alpha_0)$.

1. We take $(G', (K'_i)_i, K', (\mathsf{lk}'^j)_j) = (G, (K_i)_i, K, (\mathsf{lk}^j)_j)$. It is trivially a local configuration decomposition of Σ', and $\forall i, K_i \neq K \implies K_i = K'_i$
2. Let $L_1 :: \ldots :: L_n = \texttt{AbNormal}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle :: \alpha_0)$. By Proposition 2.4 we know that for all $i \geq 2$, $\Gamma^i(K, (\mathsf{lk}^j)_j) = \Gamma^i(K', (\mathsf{lk}'^j)_j)$. Therefore by Proposition 11 we know that for all $j \geq 2$:
$$\beta^{\ell_r}_{LstInv}(L_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta^{\ell_r}_{LstInv}(L_j, j, \_, K', (\mathsf{lk}'^i)_i)$$
One can then show that $D_{Call} = \beta^{\ell_r}_{ALst}(\langle c, m, pc' \cdot u^* \cdot st^* \cdot R\rangle, K', (\mathsf{lk}'^n)_n))$ satisfies the wanted property.
3. We know that there exist $\mathsf{AState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ and $\mathsf{AState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$ such that:
$$\beta^{\ell_r}_{ALst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \mathsf{AState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_A \mathsf{AState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta \tag{5.40}$$

We take $\Delta_{Call} = \mathsf{LState}_{c,m,pc'}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$.

4. $D_{Call} <: \Delta_{Call}$: this is a trivial consequence of Equation (5.40).

5. We want to show that $(\!| P |\!) \cup \Delta \vdash \Delta_{Call}$. First recall that $\mathsf{ExcptTable}(c, m, pc, c') = pc'$, hence $c' \leq \mathsf{Throwable}$ by Assumption 4. We know by Lemma 18 that $\beta_{LocVal}(\ell, K) \sqsubseteq \hat{v}'_{\mathsf{excpt}}$. Let $\lambda = \beta_{Lab}(\ell)$.

   – If $\ell \in dom(G)$ then we have $\beta_{LocVal}(\ell, K) = \mathsf{NFS}(\lambda)$. Moreover since $X <: \Delta$ we know that there exists $\mathsf{H}(\lambda, \{\!| c'; (f \mapsto \hat{w})^* |\!\}) \in \Delta$. Therefore we have:

$$\Delta \vdash \mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda); \{\!| c'; (f \mapsto \hat{w})^* |\!\}) \wedge c' \leq \mathsf{Throwable}$$

   – If $\ell \in dom(K)$ then we have $\beta_{LocVal}(\Sigma[\![r_{\mathsf{excpt}}]\!], K) = \mathsf{FS}(\lambda)$. Since $\ell \in dom(K)$, we know that $\hat{h}(\lambda) = \beta_{LocBlk}(H(\ell), K) \neq \bot$. Therefore from Equation (5.40) we get that $\hat{h}(\lambda) \sqsubseteq_{Blk} \hat{h}'(\lambda)$, which in turns implies that $\hat{h}'(\lambda) = \{\!| c'; (f \mapsto \hat{w})^* |\!\}$. Hence we have:

$$\Delta \vdash \mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}'^*; \hat{h}'; \mathsf{FS}(\lambda); \{\!| c'; (f \mapsto \hat{w})^* |\!\}) \wedge c' \leq \mathsf{Throwable}$$

   In both case we can apply the rule below, which is included in $(\!| P |\!)$:

$$\begin{aligned} &\mathsf{AState}_{c,m,pc}(\hat{u}'^*; \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}'^*; \hat{h}'; \_; \{\!| c'; (f \mapsto \hat{w})^* |\!\}) \\ &\qquad \wedge c' \leq \mathsf{Throwable} \implies \mathsf{LState}_{c,m,pc'}(\hat{u}'^*; \hat{v}'^*; \hat{h}'; \hat{k}') \end{aligned}$$

   This concludes this case.

- (R-UnCaught)

$$\text{(R-UnCaught)}\quad \frac{\ell = \Sigma[\![r_{\mathsf{excpt}}]\!] \qquad H(\ell) = \{\!| c_e; (f \mapsto v)^* |\!\} \qquad \mathsf{ExcptTable}(c, m, pc, c_e) = \bot}{\Sigma \Downarrow \Sigma[\alpha \mapsto \mathtt{AbNormal}(\alpha_0[r_{\mathsf{excpt}} \mapsto \ell])]}$$

Here the call-stack is abnormal $\alpha = \mathtt{AbNormal}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle :: \alpha_0)$. If α_0 is the empty list, then this case is easy. Hence we assume that :

$$\begin{aligned} \alpha &= \mathtt{AbNormal}(\langle c, m, pc \cdot v^* \cdot st^* \cdot R \rangle :: \langle c', m', pc' \cdot u'^* \cdot st'^* \cdot R' \rangle :: \alpha_1) \\ \alpha' &= \mathtt{AbNormal}(\langle c', m', pc' \cdot u'^* \cdot st'^* \cdot R'[r_{\mathsf{excpt}} \mapsto \ell] \rangle :: \alpha_1) \end{aligned}$$

1. Let $G', (K'_i)_i = G, (K_i)_i$ and $(\mathsf{lk}'^j)_j = (\mathsf{lk}_1 \sqcup^{\mathsf{loc}} \mathsf{lk}_2) :: (\mathsf{lk}_i)_{i>2}$.
   The proof that $(G', (K'_i)_i, K', (\mathsf{lk}'^j)_j)$ is a local configuration decomposition of Σ' is the same than in the (R-Return) case.

2. By Proposition 11 we get for all $j \geq 1$:

$$\beta^{\ell_r}_{LstInv}(\alpha_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta^{\ell_r}_{LstInv}(\alpha_j, j, \_, K', (\mathsf{lk}'^i)_i)$$

One can then check that:

$$D_{Call} = \beta^{\ell_r}_{ALst}(\langle c', m', pc' \cdot u'^* \cdot st'^* \cdot R'[r_{\mathsf{excpt}} \mapsto \ell]\rangle, K', (\mathsf{lk}'^j)_j)$$

3. We know that:

$$\beta^{\ell_r}_{ALst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^j)_j) = \mathsf{AState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}_1^*); \hat{v}_1^*; \hat{h}_1; \hat{k}_1) \tag{5.41}$$

$$\sqsubseteq_R \mathsf{AState}_{c,m,pc}((\hat{w}_1', \hat{u}_1'^*); \hat{v}_1'^*; \hat{h}_1'; \hat{k}_1') \in \Delta$$

$$\beta^{\ell_r}_{LstInv}(\langle c', m', pc' \cdot u'^* \cdot st'^* \cdot R'\rangle, 2, c, K, (\mathsf{lk}^j)_j) = \mathsf{Inv}^c_{c',m',pc'}((\hat{\lambda}_t, \hat{u}_2^*); \hat{v}_2^*; \hat{k}_2) \tag{5.42}$$

$$\sqsubseteq^{\Delta}_{Inv} \mathsf{LState}_{c',m',pc'}((\hat{w}_2', \hat{u}_2'^*); \hat{v}_2'^*; \hat{h}_2'; \hat{k}_2') \in \Delta$$

Let $\Delta_{Call} = \mathsf{AState}_{c',m',pc'}((\hat{w}_2', \hat{u}_2'^*); \mathsf{lift}(\hat{v}_2'^*; \hat{k}_1')[\mathsf{excpt} \mapsto (\hat{v}_1'^*)_{\mathsf{excpt}}]; \hat{h}_1'; \hat{k}_1' \hat{\cup} \hat{k}_2')$.

4. The proof that $D_{Call} <: \Delta \cup \Delta_{Call}$ is exactly the same than in the (R-Return) case.

5. We are going to show that $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$. Since $\mathsf{ExcptTable}(c, m, pc, c_e) = \bot$ we know that $c_e \leq \mathsf{Throwable}$ by Assumption 4. Therefore we have the following rule in $(\!|P|\!)$:

$$\mathsf{AState}_{c,m,pc}((\hat{w}_1', \hat{u}_1'^*); \hat{v}_1'^*; \hat{h}_1'; \hat{k}_1') \wedge \mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}_1'^*; \hat{h}_1'; \_; \{\!|c_e; \_|\!\})$$
$$\wedge\, c_e \leq \mathsf{Throwable} \implies \mathsf{Uncaught}_{c,m}((\hat{w}_1', \hat{u}_1'^*); (\hat{v}_1'^*)_{\mathsf{excpt}}; \hat{h}_1'; \hat{k}_1')$$

As it was done in (R-Caught), one can show that:

$$\Delta \vdash \mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}_1'^*; \hat{h}_1'; \_; \{\!|c_e; \_|\!\}) \wedge c_e \leq \mathsf{Throwable}$$

Therefore $\Delta \vdash \mathsf{Uncaught}_{c,m}((\hat{w}_1', \hat{u}_1'^*); \hat{\lambda}; \hat{h}_1'; \hat{k}_1')$.

By well-formedness of Σ we know that $sign(c', m') = (\tau_i)_{i \leq n} \xrightarrow{loc} \tau$, $st'_{pc'} =$ `invoke` $r_o\ m\ (r_{j_i})_{i \leq n}$ and $u^* = (R'(r_{j_i})))_{i \leq n}$. By using the same reasoning that we did in (R-Return) we can show that:

$$\Delta \vdash \mathsf{GetBlk}_o(\hat{v}_2'^*; \hat{h}_2'; \_; \{\!|c''; \_|\!\}) \wedge c'' \leq c' \wedge \hat{w}_1' = \hat{w}_2'$$
$$\wedge \left(\bigwedge\nolimits_{j \leq n} (\hat{v}_2'^*)_{i_j} \sqcap (\hat{u}_1'^*)_j \neq \bot\right)$$

Hence we can apply the following rule, which is included in $(\!|P|\!)$:

$$\begin{aligned}
&\mathsf{LState}_{c',m',pc'}((\hat{w}_2', \hat{u}_2'^{*}); \hat{v}_2'^{*}; \hat{h}_2'; \hat{k}_2') \land \mathsf{GetBlk}_o(\hat{v}_2'^{*}; \hat{h}_2'; \_; \{\!|c''; \_|\!\}) \land c'' \leq c' \\
&\land \mathsf{Uncaught}_{c,m}((\hat{w}_1', \hat{u}_1'^{*}); (\hat{v}_1'^{*})_{\mathsf{excpt}}; \hat{h}_1'; \hat{k}_1') \land \hat{w}_1' = \hat{w}_2' \land \\
&\left(\bigwedge\nolimits_{j \leq n} (\hat{v}_2'^{*})_{i_j} \sqcap (\hat{u}_1'^{*})_j \neq \bot\right) \\
\implies\ &\mathsf{LState}_{c',m',pc'}((\hat{w}_2', \hat{u}_2'); \mathsf{lift}(\hat{v}_2'^{*}; \hat{k}_1')[\mathsf{excpt} \mapsto (\hat{v}_1'^{*})_{\mathsf{excpt}}]; \hat{h}_1'; \hat{k}_1' \mathbin{\hat{\sqcup}} \hat{k}_2')
\end{aligned}$$

This shows that $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$.

- **Remaining cases** The remaining cases are straightforward or very similar to cases we already analysed. For example:
  - (R-SCall): Similar to the (R-Call) case
  - (R-NewIntent): Similar to the (R-NewObj) case
  - (R-NewArr): Similar to the (R-NewObj) case
  - (R-MoveSFld): Similar to the (R-MoveFld) case
  - (R-MoveArr): Similar to the (R-MoveFld) case
  - (R-PutExtra): Similar to the (R-MoveFld) case
  - (R-MoveException) Similar to the (R-MoveFld) case
  - (R-InterruptJoin): Similar to the (R-InterruptWait) case

□

5.3.12 Proof of Lemma 5.3.9

Proof. If $\Psi = \Psi'$ then it suffices the take $\Delta = \Delta'$.

We are just going to prove that this is true if Ψ reduces to Ψ' in one step. The lemma's proof is then obtained by a straightforward induction on the reduction length.

Let $X \in \beta_{Cnf}(\Psi)$ with $(G, (K_i, (\mathsf{lk}^{i,j})_j)_i)$ its configuration decomposition.

- Rule applied is (A-Active):

(A-Active)
$$\frac{\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S \rightsquigarrow \ell \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S'}{\Omega :: \underline{\langle \ell, s, \pi, \gamma, \alpha \rangle} :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \underline{\langle \ell, s, \pi', \gamma', \alpha' \rangle} :: \Omega' \cdot \Xi \cdot H' \cdot S'}$$

We know that:

$$X = \beta^{G}_{Stk}(\Omega :: \underline{\langle \ell, s, \pi, \gamma, \alpha \rangle} :: \Omega', \Xi, (K_l, (\mathsf{lk}^{l,j})_j)_l) \cup \beta^{G}_{Heap}(H) \cup \beta_{Stat}(S)$$

and that :

$$\beta^G_{Frm}(\underline{\langle \ell, s, \pi, \gamma, \alpha\rangle}, K_n, (\mathsf{lk}^{n,j})_j) \subseteq \beta^G_{Stk}(\Omega :: \underline{\langle \ell, s, \pi, \gamma, \alpha\rangle} :: \Omega', \Xi, (K_l, (\mathsf{lk}^{l,j})_j)_l)$$

Moreover $(G, (K_i)_i, K_n, (\mathsf{lk}^{n,j})_j)$ is a local configuration decomposition of $\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S$. We define X_{loc} as follows:

$$\begin{aligned} X_{loc} = \;& \beta^G_{Frm}(\underline{\langle \ell, s, \pi, \gamma, \alpha\rangle}, K_n, (\mathsf{lk}^{n,j})_j) \cup \beta^G_{Heap}(H) \cup \beta_{Stat}(S) \\ = \;& \beta^\ell_{Call}(\alpha, K_n, (\mathsf{lk}^{n,j})_j) \cup \beta^\ell_{Pact}(\pi) \cup \beta^G_{Pthr}(\gamma) \cup \beta^G_{Heap}(H) \cup \beta_{Stat}(S) \\ \in \;& \beta_{Lcnf}(\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S) \end{aligned}$$

Therefore we know that $X_{loc} \in \beta_{Lcnf}(\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S)$ with local configuration decomposition $G, (K_i)_i, K_n, (\mathsf{lk}^{n,j})_j$. Besides $X_{loc} \subseteq X$, hence by Lemma 15 we have $X_{loc} <: \Delta$. By Lemma 22 we know that there exists Δ'_{loc} and $X'_{loc} \in \beta_{Lcnf}(\ell \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S')$ with local configuration decomposition $G, (K_i)'_i, K'_n, (\mathsf{lk}'^{n,j})_j$ such that $\forall i \neq n, K_i = K'_i$, $\Delta'_{loc} :> X'_{loc}$ and $(\!|P|\!) \cup \Delta \vdash \Delta'_{loc}$.
For all j and $l \neq n$, let $\mathsf{lk}'^{l,j} = \mathsf{lk}^{l,j}$. Then it is quite easy to check that $(G', (K'_i, (\mathsf{lk}'^{i,j})_j)_i)$ is a configuration decomposition of Ψ'. We define X' by:

$$X' = \beta^{G'}_{Stk}(\Omega :: \underline{\langle \ell, s, \pi', \gamma', \alpha'\rangle} :: \Omega', \Xi, (K'_l, (\mathsf{lk}'^{l,j})_j)_l) \cup \beta^{G'}_{Heap}(H') \cup \beta_{Stat}(S')$$

Let n be such that Ω is of length $n-1$, n' be the length of Ω' and m be the length of Ξ. We know that:

$$\begin{aligned} &\beta^{G'}_{Stk}(\Omega :: \underline{\langle \ell, s, \pi', \gamma', \alpha'\rangle} :: \Omega', \Xi, (K'_l, (\mathsf{lk}'^{l,j})_j)_l) \backslash \beta^{G'}_{Frm}(\underline{\langle \ell, s, \pi', \gamma', \alpha'\rangle}, K'_n, (\mathsf{lk}'^{n,j})_j) \\ &= \left(\bigcup_{l=1}^{n-1} \beta^{G'}_{Frm}(\Omega_l, K'_l, (\mathsf{lk}'^{l,j})_j)\right) \cup \left(\bigcup_{l=1}^{n'} \beta^{G'}_{Frm}(\Omega'_l, K'_{l+n}, (\mathsf{lk}'^{l+n,j})_j)\right) \\ &\quad \cup \left(\bigcup_{l=1}^{m} \beta^{G'}_{Frm}(\Xi_l, K'_{l+n+n'}, (\mathsf{lk}'^{l+n+n',j})_j)\right) \end{aligned}$$

which by Proposition 13 is equal to

$$\begin{aligned} &= \left(\bigcup_{l=1}^{n-1} \beta^{G}_{Frm}(\Omega_l, K_l, (\mathsf{lk}^{l,j})_j)\right) \cup \left(\bigcup_{l=1}^{n'} \beta^{G}_{Frm}(\Omega'_l, K_{l+n}, (\mathsf{lk}^{l+n,j})_j)\right) \\ &\quad \cup \left(\bigcup_{l=1}^{m} \beta^{G}_{Frm}(\Xi_l, K_{l+n+n'}, (\mathsf{lk}^{l+n+n',j})_j)\right) \end{aligned}$$

Which implies that:

$$X' \backslash X \subseteq \beta^{G'}_{Frm}(\underline{\langle \ell, s, \pi', \gamma', \alpha'\rangle}, K'_n, (\mathsf{lk}'^{n,j})_j) \cup \beta^{G'}_{Heap}(H') \cup \beta_{Stat}(S') = X'_{loc}$$

We define $\Delta' = \Delta \cup \Delta'_{loc}$.We know that $X <: \Delta$ and $X'_{loc} <: \Delta'_{loc}$, therefore by Lemma 16 we have $X \cup X'_{loc} <: \Delta \cup \Delta'_{loc} = \Delta'$. Moreover $X' \subseteq X \cup X'_{loc}$, therefore by Lemma 15 we have $X' <: \Delta'$. We conclude by observing that since $(\!|P|\!) \cup \Delta \vdash \Delta'_{loc}$, we trivially have $(\!|P|\!) \cup \Delta \vdash \Delta'$.

- Rule applied is (A-Deactivate):

$$\frac{\text{(A-Deactivate)}}{\Omega :: \underline{\langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle} :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle :: \Omega' \cdot \Xi \cdot H \cdot S}$$

In this case $\beta_{Cnf}(\Omega :: \underline{\langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle} :: \Omega' \cdot \Xi \cdot H \cdot S) = \beta_{Cnf}(\Omega :: \langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle :: \Omega' \cdot \Xi \cdot H \cdot S)$, hence the conclusion immediately follows from the induction hypothesis.

- Rule applied is (A-Step):

$$\text{(A-Step)}\quad \frac{\begin{array}{c}(s, s') \in \mathit{Lifecycle} \qquad \pi \neq \varepsilon \Rightarrow (s, s') = (\mathit{running}, \mathit{onPause}) \\ H(\ell).\mathsf{finished} = \mathtt{true} \Rightarrow \\ (s, s') \in \{(\mathit{running}, \mathit{onPause}), (\mathit{onPause}, \mathit{onStop}), (\mathit{onStop}, \mathit{onDestroy})\}\end{array}}{\langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \underline{\langle \ell, s', \pi, \gamma, \alpha_{\ell.s'} \rangle} :: \Omega \cdot \Xi \cdot H \cdot S}$$

We have:

$$X = \beta^G_{Stk}(\underline{\langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle} :: \Omega, \Xi, (K_l, (\mathsf{lk}^{l,j})_j)_l) \cup \beta^G_{Heap}(H) \cup \beta_{Stat}(S)$$

Since we only focus on well-formed configurations, we have $H(\ell) = \{\!|c; (f \mapsto u)^*|\!\}$ for some activity class c and $\ell = p_c$ for some pointer p. We then observe that $\alpha_{\ell.s'} = \langle c', m, 0 \cdot v^* \cdot st^* \cdot R \rangle :: \varepsilon$, where $(c', st^*) = \mathit{lookup}(c, m)$ for some $m \in \mathit{cb}(c, s)$, $\mathit{sign}(c', m) = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ and:

$$R = ((r_i \mapsto \mathbf{0})^{i \leq loc}, r_{loc+1} \mapsto \ell, (r_{loc+1+j} \mapsto v_j)^{j \leq n})$$

for some values $v_1, \ldots, v_n$ of the correct type $\tau_1, \ldots, \tau_n$. By Assumption 8, we also have $c \leq c'$.

Given that $\Delta :> X \in \beta_{Cnf}(\Psi)$, we have $\Delta :> \beta^G_{Heap}(H)$. We know that $\ell = p_c \in \mathit{dom}(H)$, and since local heaps contain only locations whose annotations are program points, we know that $\ell \in \mathit{dom}(G)$. Therefore there exists $\mathsf{H}(\lambda, \hat{b}) \in \Delta$ such that $\lambda = \beta_{Lab}(\ell) = c$ and $\beta_{Blk}(\{\!|c; (f \mapsto u)^*|\!\}) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}$. This implies that $\hat{b} = \{\!|c; (f \mapsto \hat{v})^*|\!\}$ for some $\hat{v}^*$ such that $\forall i, \beta_{Val}(u_i) \sqsubseteq^{\mathsf{nfs}} \hat{v}_i$. Hence using the implications Cbk included in $(\!|P|\!)$ we get that:

$$(\!|P|\!) \cup \Delta \vdash \mathsf{LState}_{c',m,0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*) \tag{5.43}$$

Let $\Delta' = \Delta \cup \{\mathsf{LState}_{c',m,0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*)\}$. From Equation 5.43 we get that $(\!|P|\!) \cup \Delta \vdash \Delta'$.

Let $G' = G$, for all $i > 1$ let $K'_i = K_i$ and for all $j > 1$, $(\mathsf{lk}'^{l,j})_j = (\mathsf{lk}^{l,j})_j$. Let also K'_1 be a fresh empty local heap and $(\mathsf{lk}'^{1,j})_j = (\{(\ell \mapsto 0) \mid \ell\}) :: \varepsilon$. Using Assumption 9, it is simple to show that $(G', (K'_i, (\mathsf{lk}'^{i,j})_j)_i)$ is a configuration decomposition of $\underline{\langle \ell, s', \pi, \gamma, \alpha_{\ell.s'}\rangle} :: \Omega \cdot \Xi \cdot H \cdot S$, and that:

$$\Delta' >: \{\mathsf{LState}_{c',m,0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j\leq n}); (\hat{\mathbf{0}}_k)^{k\leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j\leq n}; (\bot)^*; 0^*)\} :> \beta^{\ell}_{\mathit{Call}}(\alpha_{\ell.s'}, K'_1, (\mathsf{lk}'^{1,j})_j) \quad (5.44)$$

Observe that $\beta^{G}_{\mathit{Pthr}}(\gamma) = \beta^{G'}_{\mathit{Pthr}}(\gamma)$. Besides $\Delta :> \beta_{\mathit{Cnf}}(\Omega \cdot \Xi \cdot H \cdot S)$ implies that $\beta^{\ell}_{\mathit{Pact}}(\pi) \cup \beta^{G}_{\mathit{Pthr}}(\gamma) <: \Delta$, and we know that since $\Delta \subseteq \Delta'$ we have $\Delta <: \Delta'$. Therefore by transitivity of $<:$ we have :

$$\beta^{\ell}_{\mathit{Pact}}(\pi) \cup \beta^{G'}_{\mathit{Pthr}}(\gamma) <: \Delta' \quad (5.45)$$

It is easy to check that $X' \in \beta_{\mathit{Cnf}}(\Psi')$, where X' is the following set of facts:

$$X' = \beta^{G'}_{\mathit{Stk}}(\underline{\langle \ell, s', \pi, \gamma, \alpha_{\ell.s'}\rangle} :: \Omega, \Xi, (K'_l, (\mathsf{lk}'^{l,j})_j)_l) \cup \beta^{G}_{\mathit{Heap}}(H) \cup \beta_{\mathit{Stat}}(S)$$

Using Proposition 13, one can check that:

$$X' \backslash X = \beta^{\ell}_{\mathit{Call}}(\alpha_{\ell.s'}, K'_1, (\mathsf{lk}'^{1,j})_j) \cup \beta^{\ell}_{\mathit{Pact}}(\pi) \cup \beta^{G'}_{\mathit{Pthr}}(\gamma)$$

Equation 5.44 and Equation 5.45 give us that $X' \backslash X <: \Delta'$. We conclude by observing that since $X <: \Delta <: \Delta'$ and $X' \subseteq X \cup (X' \backslash X)$, we have $X' <: \Delta'$.

- Rule applied is (A-Hidden):

  (A-Hidden)

  $$\frac{s \in \{\mathit{onResume}, \mathit{onPause}\} \qquad \begin{array}{c}\varphi = \langle \ell, s, \pi, \gamma, \overline{\alpha}\rangle \\ (s', s'') \in \{(\mathit{onPause}, \mathit{onStop}), (\mathit{onStop}, \mathit{onDestroy})\}\end{array}}{\begin{array}{c}\varphi :: \Omega :: \langle \ell', s', \pi', \gamma', \overline{\alpha}'\rangle :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \\ \varphi :: \Omega :: \underline{\langle \ell', s'', \pi', \gamma', \alpha_{\ell'.s''}\rangle} :: \Omega' \cdot \Xi \cdot H \cdot S\end{array}}$$

  This case is analogous to the case (A-Step).

- Rule applied is (A-Destroy):

  (A-Destroy)

  $$\frac{H(\ell).\mathsf{finished} = \mathtt{true}}{\Omega :: \langle \ell, \mathit{onDestroy}, \pi, \gamma, \overline{\alpha}\rangle :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \Omega' \cdot \Xi \cdot H \cdot S}$$

  Let n be the length of Ω. It is easy to check that $(G \cup K_n, (K_l, (\mathsf{lk}^{l,j})_j)_{l\neq n})$ is a configuration decomposition of $\Omega :: \Omega' \cdot \Xi \cdot H \cdot S$, and that $X' \in \beta_{\mathit{Cnf}}(\Psi')$ where:

  $$X' = \beta^{G \cup K_n}_{\mathit{Stk}}(\Omega :: \Omega', \Xi, (K_l, (\mathsf{lk}^{l,j})_j)_{l\neq n}) \cup \beta^{G}_{\mathit{Heap}}(H) \cup \beta_{\mathit{Stat}}(S) \subseteq X$$

  Since $X <: \Delta$, this implies that $X' <: \Delta$. We conclude with the trivial observation that $(\!|P|\!) \cup \Delta \vdash \Delta$.

- Rule applied is (A-Back):

$$\text{(A-Back)}\quad \frac{H' = H[\ell \mapsto H(\ell)[\mathsf{finished} \mapsto \mathtt{true}]]}{\langle \ell, running, \varepsilon, \gamma, \overline{\alpha}\rangle :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \langle \ell, running, \varepsilon, \gamma, \overline{\alpha}\rangle :: \Omega \cdot \Xi \cdot H' \cdot S}$$

Let $b = H(\ell)$. Since we only focus on well-formed configurations, we have $b = \{\!|c; (f \mapsto u)^*, \mathsf{finished} \mapsto v|\!\}$ for some activity class c and some boolean value v. Let then $b' = H'(\ell) = \{\!|c; (f \mapsto u)^*, \mathsf{finished} \mapsto \mathtt{true}|\!\}$ according to the reduction rule.

Given that $\Delta :> X \in \beta_{Cnf}(\Psi)$, we have $\Delta :> \beta^G_{Heap}(H)$. We know that $\ell = p_c \in dom(H)$, and since local heaps contain only locations whose annotations are program points, we know that $\ell \in dom(G)$. Therefore there exists $\mathsf{H}(\lambda, \hat{b}) \in \Delta$ such that $\lambda = \beta_{Lab}(\ell) = c$ and $\beta_{Blk}(\{\!|c; (f \mapsto u)^*, \mathsf{finished} \mapsto v|\!\}) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}$. This implies that $\hat{b} = \{\!|c; (f \mapsto \hat{u})^*, \mathsf{finished} \mapsto \hat{v}|\!\}$ for some $\hat{u}^*, \hat{v}$ such that $\forall i, \beta_{Val}(u_i) \sqsubseteq^{\mathsf{nfs}} \hat{u}_i$ and $\beta_{Val}(v) \sqsubseteq^{\mathsf{nfs}} \hat{v}$. It is easy to check that:

$$\beta_{Blk}(b') = \{\!|c; (f \mapsto \beta_{Val}(u))^*, \mathsf{finished} \mapsto \widehat{\mathtt{true}}|\!\}$$

We define $\Delta' = \Delta \cup \{\mathsf{H}(\lambda, \{\!|c; (f \mapsto \hat{u})^*, \mathsf{finished} \mapsto \top_{\mathtt{bool}}|\!\})\}$. Since $\mathsf{H}(\lambda, \hat{b}) \in \Delta$ we have by using the implication *Fin* in $(\!|P|\!)$ that:

$$(\!|P|\!) \cup \Delta \vdash \mathsf{H}(\lambda, \{\!|c; (f \mapsto \hat{u})^*, \mathsf{finished} \mapsto \top_{\mathtt{bool}}|\!\})$$

Therefore $(\!|P|\!) \cup \Delta \vdash \Delta'$. We then observe that:

$$\begin{aligned} \mathsf{H}(\beta_{Lab}(\ell), \beta_{Blk}(b')) \quad & \sqsubseteq^{\mathsf{nfs}}_{Blk} \quad \mathsf{H}(\lambda, \{\!|c; (f \mapsto \hat{u})^*, \mathsf{finished} \mapsto \widehat{\mathtt{true}}|\!\}) \\ & \sqsubseteq^{\mathsf{nfs}}_{Blk} \quad \mathsf{H}(\lambda, \{\!|c; (f \mapsto \hat{u})^*, \mathsf{finished} \mapsto \top_{\mathtt{bool}}|\!\}) \end{aligned}$$

Hence $\beta^G_{Heap}(H') <: \Delta'$. It is then easy to conclude this case.

- Rule applied is (A-Swap):

$$\text{(A-Swap)}\quad \frac{\begin{array}{c}\varphi' = \langle \ell', onPause, \varepsilon, \gamma', \overline{\alpha}'\rangle \qquad H(\ell').\mathsf{finished} = \mathtt{true} \\ \varphi = \langle \ell, s, i :: \pi, \gamma, \overline{\alpha}\rangle \qquad s \in \{onPause, onStop\} \qquad H(\ell').parent = \ell\end{array}}{\varphi' :: \varphi :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \varphi :: \varphi' :: \Omega \cdot \Xi \cdot H \cdot S}$$

Just take $G' = G, K'_1 = K_2, K'_2 = K_1$, for all $j, \mathsf{lk}'^{1,j} = \mathsf{lk}^{2,j}$, $\mathsf{lk}'^{2,j} = \mathsf{lk}^{1,j}$ (we simply exchange the first local heap and filters with the second local heap and filters). The rest is kept unchanged: for all $l > 2$, for all j, $K'_i = K_i$ and $\mathsf{lk}'^{l,j} = \mathsf{lk}^{l,j}$.

It is quite simple to check that $(G, (K_i, (\mathsf{lk}^{i,j})_j)_i)$ is a configuration decomposition and that the corresponding set of abstract facts are the same.

Therefore $\beta_{Cnf}(\Psi) = \beta_{Cnf}(\Psi')$, which concludes this case.

- Rule applied is (A-Start):

$$
\textsc{(A-Start)}\quad
\frac{\begin{array}{c}
s \in \{\mathit{onPause}, \mathit{onStop}\} \\
i = \{\!|@c; (k \mapsto v)^*|\!\} \qquad \emptyset \vdash \mathit{ser}^{H}_{\mathit{Blk}}(i) = (i', H') \qquad p_c, p'_{\mathit{in}(c)} \notin \mathit{dom}(H, H') \\
o = \{\!|c; (f_\tau \mapsto \mathbf{0}_\tau)^*, \mathsf{finished} \mapsto \mathtt{false}, \mathsf{intent} \mapsto p'_{\mathit{in}(c)}, \mathit{parent} \mapsto \ell|\!\} \\
H'' = H, H', p_c \mapsto o, p'_{\mathit{in}(c)} \mapsto i'
\end{array}}{\begin{array}{c}
\langle \ell, s, i :: \pi, \gamma, \overline{\alpha}\rangle :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \\
\underline{\langle p_c, \mathit{constructor}, \varepsilon, \varepsilon, \alpha_{p_c.\mathit{constructor}}\rangle} :: \langle \ell, s, \pi, \gamma, \overline{\alpha}\rangle :: \Omega \cdot \Xi \cdot H'' \cdot S
\end{array}}
$$

Since we only focus on well-formed configurations, we know that $\ell = p''_{c''}$ for some pointer p'' and some activity class c''. We then observe that $\alpha_{p_c.\mathit{constructor}} = \langle c', m, 0 \cdot v^* \cdot \mathit{st}^* \cdot R\rangle :: \varepsilon$, where $(c', \mathit{st}^*) = \mathit{lookup}(c, \mathit{constructor})$, $\mathit{sign}(c', \mathit{constructor}) = \tau_1, \dots, \tau_n \xrightarrow{loc} \tau$ and:

$$R = ((r_i \mapsto \mathbf{0})^{i \leq loc}, r_{loc+1} \mapsto p_c, (r_{loc+1+j} \mapsto v'_j)^{j \leq n}),$$

for some values $v'_1, \dots, v'_n$ of the correct type $\tau_1, \dots, \tau_n$. By Assumption 8, we also have $c \leq c'$.

Given that $X <: \Delta$, we have $\Delta :> \beta^{\ell}_{\mathit{Pact}}(i :: \pi)$, which implies that there exists $\mathsf{I}_\lambda(\hat{b}) \in \Delta$ such that $\lambda = \beta_{\mathit{Lab}}(\ell) = c'$ and $\beta_{\mathit{Blk}}(i) \sqsubseteq^{\mathsf{nfs}}_{\mathit{Blk}} \hat{b}$. This implies that $\hat{b} = \{\!|@c; \hat{v}|\!\}$ for some $\hat{v}$ such that $\sqcup_i \beta_{\mathit{Val}}(v_i) \sqsubseteq^{\mathsf{nfs}} \hat{v}$. Using the implications Act in $(\!|P|\!)$ we get:

$$(\!|P|\!) \cup \Delta \;\vdash\; \mathsf{H}(\mathit{in}(c), \{\!|@c; \hat{v}|\!\}) \tag{5.46}$$

$$(\!|P|\!) \cup \Delta \;\vdash\; \mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{\mathtt{false}}, \mathit{parent} \mapsto c', \mathsf{intent} \mapsto \mathit{in}(c)|\!\}) \tag{5.47}$$

Hence using the implications Cbk included in $(\!|P|\!)$ we get that:

$$
\begin{aligned}
&(\!|P|\!) \cup \{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{\mathtt{false}}, \mathit{parent} \mapsto c', \mathsf{intent} \mapsto \mathit{in}(c)|\!\})\} \\
&\quad \vdash \mathsf{LState}_{\mathsf{c'},\mathsf{m},\mathsf{0}}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*)
\end{aligned} \tag{5.48}
$$

We define the set of abstract fact:

$$
\begin{aligned}
\Delta' \;=\; & \Delta \cup \{\mathsf{LState}_{\mathsf{c'},\mathsf{m},\mathsf{0}}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*)\} \\
& \cup \{\mathsf{H}(\mathit{in}(c), \{\!|@c; \hat{v}|\!\})\} \\
& \cup \{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{\mathtt{false}}, \mathit{parent} \mapsto c', \mathsf{intent} \mapsto \mathit{in}(c)|\!\})\}
\end{aligned}
$$

From Equation 5.46, Equation 5.47 and Equation 5.48 we get that $(\!|P|\!) \cup \Delta \vdash \Delta'$.

Configuration Decomposition Let K'_0 be an fresh empty local heap. We take $G' = G \cup H' \cup \{p_c, p'_{\mathit{in}(c)}\}$, $(K'_l)_l = K'_0 :: (K)_l$ and $(\mathsf{lk}'^{l,j})_{l,j} = ((\{(\ell \mapsto 0) \mid \ell\}) :: \varepsilon) :: (\mathsf{lk}^{l,j})_{l,j}$.

Since $(G, (K_i), K_1, (\mathsf{lk}^{1,j})_j)$ is a local configuration decomposition of $\ell \cdot \overline{\alpha} \cdot (i :: \pi) \cdot \gamma \cdot H \cdot S$, we know that there exists ℓ' such that $(\ell' \mapsto i) \in G$. Moreover $\Delta :> \beta^{G}_{Heap}(H)$ and $ser^{H}_{Blk}(i) = (i', H')$, therefore by applying Lemma 26 we know that $\Delta :> \beta^{G}_{Heap}(H')$ and that $G \cup H', (K_i)_i$ is a heap decomposition of $H \cup H' \cdot S$.

Since $\ell = p''_c$ we know that $\ell \in G$, hence for all i, $o \not\to_{\mathsf{ref}} K_i$. By Lemma 25 we know that for all i, $i \not\to_{\mathsf{ref}} K_i$. Moreover p_c and $p'_{in(c)}$ are fresh locations, therefore $G', (K_i)_i$ is a heap decomposition of $H'' \cdot S$. Since K'_0 is a fresh empty local heap we easily get from this that $G', (K'_i)_i$ is a heap decomposition of $H'' \cdot S$.

Using Assumption 9, it is simple to check that $(G', (K'_i, (\mathsf{lk}'^{i,j})_j)_i)$ is a configuration decomposition of Ψ'.

Let X' be the corresponding set of facts:

$$\beta^{G'}_{Stk}(\underline{\langle p_c, constructor, \varepsilon, \varepsilon, \alpha_{p_c.constructor}\rangle} :: \langle \ell, s, \pi, \gamma, \overline{\alpha}\rangle :: \Omega, \Xi, (K'_l, (\mathsf{lk}'^{l,j})_j)_l) \\ \cup \beta^{G'}_{Heap}(H'') \cup \beta_{Stat}(S)$$

We are going to prove that X' is over-approximated by the set of abstract facts Δ'.

Heap We already saw that $\Delta :> \beta^{G}_{Heap}(H')$, and by applying Lemma 23 we know that $\beta_{Blk}(i) = \beta_{Blk}(i')$. We then observe that:

$$\begin{array}{rcll} \{\mathsf{H}(in(c), \{\!|@c; \hat{v}|\!\})\} & :> & \{\mathsf{H}(in(c), \beta_{Blk}(i)\} & \text{since } \beta_{Blk}(i) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b} = \{\!|@c; \hat{v}|\!\}) \\ & = & \{\mathsf{H}(in(c), \beta_{Blk}(i')\} & \text{since } \beta_{Blk}(i) = \beta_{Blk}(i') \\ & = & \{\mathsf{H}(\beta_{Lab}(p'_{in(c)}), \beta_{Blk}(i')\} & \text{by definition} \end{array} \tag{5.49}$$

Also notice that:

$$\{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{\mathtt{false}}, parent \mapsto c', \mathsf{intent} \mapsto in(c)|\!\})\} = \\ \mathsf{H}(\beta_{Lab}(p_c), \beta_{Blk}(o)) \tag{5.50}$$

Moreover it is simple to see that we have:

$$\beta^{G'}_{Heap}(H'') = \beta^{G}_{Heap}(H) \cup \beta^{G \cup \mathsf{H}'}_{Heap}(H') \cup \{\mathsf{H}(\beta_{Lab}(p_c), \beta_{Blk}(o))\} \cup \\ \{\{\mathsf{H}(\beta_{Lab}(p'_{in(c)}), \beta_{Blk}(i')\}\}$$

We already saw that $\beta^{G \cup \mathsf{H}'}_{Heap}(H') <: \Delta <: \Delta'$. This together with Equation 5.49 and Equation 5.50 shows that $\beta^{G'}_{Heap}(H'') <: \Delta'$.

Activity Stack Let n be the length of Ω, and let m be the length of Ξ.

$$\begin{aligned}
&\beta^{G'}_{\mathit{Stk}}(\underline{\langle p_c, \mathit{constructor}, \varepsilon, \varepsilon, \alpha_{p_c.\mathit{constructor}}\rangle} :: \langle \ell, s, \pi, \gamma, \overline{\alpha}\rangle :: \Omega, \Xi, (K'_l, (\mathsf{lk}'^{l,j})_j)_l)\\
= \;&\beta^{G'}_{\mathit{Frm}}(\underline{\langle p_c, \mathit{constructor}, \varepsilon, \varepsilon, \alpha_{p_c.\mathit{constructor}}\rangle}, K'_0, (\mathsf{lk}'^{0,j})_j)\\
&\cup \beta^{G'}_{\mathit{Frm}}(\langle \ell, s, \pi, \gamma, \overline{\alpha}\rangle, K'_1, (\mathsf{lk}'^{1,j})_j)\\
&\cup \left(\bigcup_{1\le l\le n} \beta^{G'}_{\mathit{Frm}}(\Omega_l, K'_{l+1}, (\mathsf{lk}'^{l+1,j})_j)\right)\\
&\cup \left(\bigcup_{1\le l\le m} \beta^{G'}_{\mathit{Frm}}(\Xi_l, K'_{l+n+1}, (\mathsf{lk}'^{l+n+1,j})_j)\right)
\end{aligned}$$

By Proposition 13 this is equal to:

$$\begin{aligned}
&\beta^{G'}_{\mathit{Frm}}(\underline{\langle p_c, \mathit{constructor}, \varepsilon, \varepsilon, \alpha_{p_c.\mathit{constructor}}\rangle}, K'_0, (\mathsf{lk}'^{0,j})_j)\\
&\cup \beta^{G}_{\mathit{Frm}}(\langle \ell, s, \pi, \gamma, \overline{\alpha}\rangle, K_1, (\mathsf{lk}^{1,j})_j)\\
&\cup \left(\bigcup_{1\le l\le n} \beta^{G}_{\mathit{Frm}}(\Omega_l, K_{l+1}, (\mathsf{lk}^{l+1,j})_j)\right) \cup \left(\bigcup_{1\le l\le m} \beta^{G}_{\mathit{Frm}}(\Xi_l, K_{l+n+1}, (\mathsf{lk}^{l+n+1,j})_j)\right)
\end{aligned}$$

We then observe that:

$$\begin{aligned}
\Delta' \;&:>\; \{\mathsf{LState}_{\mathsf{c}',\mathsf{m},0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j\le n}); (\hat{\mathbf{0}}_k)^{k\le loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j\le n}; (\bot)^*; 0^*)\}\\
&:>\; \beta^{G'}_{\mathit{Frm}}(\underline{\langle p_c, \mathit{constructor}, \varepsilon, \varepsilon, \alpha_{p_c.\mathit{constructor}}\rangle}, K'_0, (\mathsf{lk}'^{0,j})_{0,j})
\end{aligned}$$

This proves that the changes to the activity stack are over-approximated by Δ'.

- Rule applied is (A-Replace):

(A-Replace)

$$\frac{p_c \notin \mathit{dom}(H) \qquad \begin{array}{c} H(\ell) = \{\!|c; (f_\tau \mapsto v)^*, \mathsf{finished} \mapsto u|\!\}\\ o = \{\!|c; (f_\tau \mapsto \mathbf{0}_\tau)^*, \mathsf{finished} \mapsto \mathtt{false}|\!\}\end{array} \qquad H' = H, p_c \mapsto o}{\begin{array}{c}\langle \ell, \mathit{onDestroy}, \pi, \gamma, \overline{\alpha}\rangle :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow\\ \underline{\langle p_c, \mathit{constructor}, \pi, \gamma, \alpha_{p_c.\mathit{constructor}}\rangle} :: \Omega \cdot \Xi \cdot H' \cdot S\end{array}}$$

Since we only focus on well-formed configurations, we know that c is an activity class and $\ell = p'_c$ for some pointer p'.

We then observe that $\alpha_{p_c.\mathit{constructor}} = \langle c', m, 0 \cdot v^* \cdot st^* \cdot R\rangle :: \varepsilon$, where $(c', st^*) = \mathit{lookup}(c, \mathit{constructor})$, $\mathit{sign}(c', \mathit{constructor}) = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ and:

$$R = ((r_i \mapsto \mathbf{0})^{i\le loc}, r_{loc+1} \mapsto p_c, (r_{loc+1+j} \mapsto v'_j)^{j\le n}),$$

for some values $v'_1, \ldots, v'_n$ of the correct type $\tau_1, \ldots, \tau_n$. By Assumption 8, we also have $c \le c'$.

Given that $\Delta :> X \in \beta_{Cnf}(\Psi)$, we have $\Delta :> \beta^{G}_{Heap}(H)$. We know that $\ell = p'_c \in dom(H)$, and since local heaps contain only locations whose annotations are program points, we know that $\ell \in dom(G)$. Therefore there exists $\mathsf{H}(\lambda, \hat{b}) \in \Delta$ such that $\lambda = \beta_{Lab}(\ell) = c$ and $\beta_{Blk}(\{\!| c; (f \mapsto v)^*, \mathsf{finished} \mapsto u |\!\}) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}$. This implies that $\hat{b} = \{\!| c; (f \mapsto \hat{v})^*, \mathsf{finished} \mapsto \hat{u} |\!\}$ for some $\hat{v}^*, \hat{u}$ such that $\forall i, \beta_{Val}(v_i) \sqsubseteq^{\mathsf{nfs}} \hat{v}_i$ and $\beta_{Val}(u) \sqsubseteq^{\mathsf{nfs}} \hat{u}$. Hence using the implications *Cbk* and *Rep*[2] included in $(\!| P |\!)$ we get that:

$$(\!| P |\!) \cup \Delta \vdash \mathsf{LState}_{\mathsf{c',m,0}}((\mathsf{NFS}(c), (\top_{\tau_j})^{j\le n}); (\hat{\mathbf{0}}_k)^{k\le loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j\le n}; (\bot)^*; 0^*) \tag{5.51}$$

$$(\!| P |\!) \cup \Delta \vdash \mathsf{H}(c, \{\!| c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{\mathtt{false}} |\!\})) \tag{5.52}$$

We define the set of abstract Δ' by:

$$\begin{aligned}\Delta' = \Delta \quad & \cup \left\{ \mathsf{LState}_{\mathsf{c',m,0}}((\mathsf{NFS}(c), (\top_{\tau_j})^{j\le n}); (\hat{\mathbf{0}}_k)^{k\le loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j\le n}; (\bot)^*; 0^*) \right\} \\ & \cup \left\{ \mathsf{H}(c, \{\!| c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{\mathtt{false}} |\!\})) \right\}\end{aligned}$$

Let $G' = G \cup \{p_c\}$, for all $i > 1$ let $K'_i = K_i$ and for all $j > 1, (\mathsf{lk}'^{l,j})_j = (\mathsf{lk}^{l,j})_j$. Let also K'_1 be a fresh empty local heap and $(\mathsf{lk}'^{1,j})_j = (\{(\ell \mapsto 0) \mid \ell\}) :: \varepsilon$. Using Assumption 9, it is simple to show that $(G', (K'_i, (\mathsf{lk}'^{i,j})_j)_i)$ is a configuration decomposition of $\underline{\langle \ell, s', \pi, \gamma, \alpha_{p_c.constructor} \rangle} :: \Omega \cdot \Xi \cdot H' \cdot S$ and that:

$$\begin{aligned}\beta^{\ell}_{Call}(\alpha_{p_c.constructor}, K'_1, (\mathsf{lk}'^{1,j})_j) <: \{ & \mathsf{LState}_{\mathsf{c',m,0}}((\mathsf{NFS}(c), (\top_{\tau_j})^{j\le n}); (\hat{\mathbf{0}}_k)^{k\le loc}, \\ & \mathsf{NFS}(c), (\top_{\tau_j})^{j\le n}; (\bot)^*; 0^*)\} <: \Delta'\end{aligned} \tag{5.53}$$

Observe that $\beta^{G}_{Pthr}(\gamma) = \beta^{G'}_{Pthr}(\gamma)$. Besides $\Delta :> \beta_{Cnf}(\Omega \cdot \Xi \cdot H \cdot S)$ implies that $\beta^{\ell}_{Pact}(\pi) \cup \beta^{G}_{Pthr}(\gamma) <: \Delta$, and we know that since $\Delta \subseteq \Delta'$ we have $\Delta <: \Delta'$. Therefore by transitivity of $<:$ we have :

$$\beta^{\ell}_{Pact}(\pi) \cup \beta^{G'}_{Pthr}(\gamma) <: \Delta' \tag{5.54}$$

Moreover:

$$\begin{aligned}\beta^{G'}_{Heap}(H') &= \beta^{G}_{Heap}(H) \cup \mathsf{H}(\beta_{Lab}(p_c), \beta_{Blk}(o)) \\ &= \beta^{G}_{Heap}(H) \cup \mathsf{H}(c, \beta_{Blk}(\{\!| c; (f_\tau \mapsto \mathbf{0}_\tau)^*, \mathsf{finished} \mapsto \mathtt{false} |\!\})) \\ &<: \quad \Delta \quad \cup \mathsf{H}(c, \{\!| c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{\mathtt{false}} |\!\})) \\ &<: \quad \Delta'\end{aligned} \tag{5.55}$$

[2] We assume here that boolean fields are initialized to `false`. The proof can be adapted to the case where they are initialized to `true` by using the implication in rule *Fin*.

It is easy to check that $X' \in \beta_{Cnf}(\Psi')$, where X' is the following set of facts:

$$X' = \beta^{G'}_{Stk}(\underline{\langle \ell, s', \pi, \gamma, \alpha_{p_c.constructor} \rangle} :: \Omega, \Xi, (K'_l, (\mathsf{lk}'^{l,j})_j)_l) \cup \beta^{G'}_{Heap}(H') \cup \beta_{Stat}(S)$$

Using Proposition 13 one can check that:

$$X' \backslash X = \beta^{\ell}_{Call}(\alpha_{\ell.s'}, K'_1, (\mathsf{lk}'^{1,j})_j) \cup \beta^{\ell}_{Pact}(\pi) \cup \beta^{G'}_{Pthr}(\gamma) \cup \beta^{G'}_{Heap}(H')$$

Equation 5.53, Equation 5.54 and Equation 5.55 give us that $X' \backslash X <: \Delta'$. We conclude by observing that since $X <: \Delta <: \Delta'$ and $X' \subseteq X \cup (X' \backslash X)$ we have $X' <: \Delta'$.

- Rule applied is (A-RESULT):

(A-RESULT)

$$\frac{\begin{array}{c} \varphi' = \langle \ell', onPause, \varepsilon, \gamma', \overline{\alpha}' \rangle \qquad H(\ell').\mathsf{finished} = \mathtt{true} \\ \varphi = \langle \ell, s, \varepsilon, \gamma, \overline{\alpha} \rangle \qquad s \in \{onPause, onStop\} \qquad H(\ell').parent = \ell \\ \emptyset \vdash ser^{H}_{Val}(H(\ell').result) = (w', H') \qquad H'' = (H, H')[\ell \mapsto H(\ell)[\mathsf{result} \mapsto w']] \end{array}}{\varphi' :: \varphi :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \underline{\langle \ell, s, \varepsilon, \gamma, \alpha_{\ell.onActivityResult} \rangle} :: \varphi' :: \Omega \cdot \Xi \cdot H'' \cdot S}$$

Since we focus only on well-formed configurations, we have $\ell = p_c$ and $\ell' = p'_{c'}$ for some pointers p, p' and some activity classes c, c'. Also, let $H(\ell) = \{\!|c; (f \mapsto \hat{v})^*|\!\}$ and $H(\ell') = \{\!|c'; (f' \mapsto \hat{v}')^*, parent \mapsto \ell, result \mapsto w|\!\}$. We then observe that $\alpha_{p_c.onActivityResult} = \langle c'', m, 0 \cdot v^* \cdot st^* \cdot R \rangle :: \varepsilon$, where $(c'', st^*) = lookup(c, onActivityResult)$, $sign(c'', onActivityResult) = \tau_1, \dots, \tau_n \xrightarrow{loc} \tau$ and:

$$R = ((r_i \mapsto \mathbf{0})^{i \le loc}, r_{loc+1} \mapsto p_c, (r_{loc+1+j} \mapsto v'_j)^{j \le n}),$$

for some values $v'_1, \dots, v'_n$ of the correct type $\tau_1, \dots, \tau_n$. By Assumption 8, we also have $c \le c''$.

Given that $\Delta :> X \in \beta_{Cnf}(\Psi)$, we have $\Delta :> \beta^{G}_{Heap}(H)$. We know that $\ell = p_c \in dom(H)$, and since local heaps contain only locations whose annotations are program points, we know that $\ell \in dom(G)$. Therefore there exists $\mathsf{H}(\lambda, \hat{b}) \in \Delta$ such that $\lambda = \beta_{Lab}(\ell) = c$ and $\beta_{Blk}(\{\!|c; (f \mapsto v)^*|\!\}) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}$. This implies that $\hat{b} = \{\!|c; (f \mapsto \hat{v})^*|\!\}$ for some $\hat{v}^*$ such that $\forall i, \beta_{Val}(v_i) \sqsubseteq^{\mathsf{nfs}} \hat{v}_i$. Hence using the implications Cbk included in $(\!|P|\!)$ we get that:

$$(\!|P|\!) \cup \Delta \vdash \mathsf{LState}_{\mathsf{c}'',\mathsf{m},0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \le n}); (\hat{\mathbf{0}}_k)^{k \le loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \le n}; (\bot)^*; 0^*) \tag{5.56}$$

Similarly, there exists $\mathsf{H}(\lambda', \hat{b}') \in \Delta$ such that $\lambda' = \beta_{Lab}(\ell') = c'$ and $\beta_{Blk}(H(\ell')) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}'$, which implies that $\hat{b}' = \{\!|c'; (f' \mapsto \hat{v}')^*, parent \mapsto c, \mathsf{result} \mapsto \hat{w}|\!\}$ for some $\hat{v}'^*, \lambda''$ such that $\forall i. \beta_{Val}(v'_i) \sqsubseteq^{\mathsf{nfs}} \hat{v}'_i$ and $\beta_{Val}(w) \sqsubseteq^{\mathsf{nfs}} \hat{w}$. Hence by using the implication Res we get

$$(\!|P|\!) \cup \Delta \vdash \mathsf{H}(c, \{\!|c; (f \mapsto \hat{v})^*[\mathsf{result} \mapsto \hat{w}]|\!\}) \tag{5.57}$$

We define the following set of facts:

$$\Delta' = \Delta \cup \{\mathsf{LState}_{\mathsf{c}'',\mathsf{m},0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j\leq n}); (\hat{\mathbf{0}}_k)^{k\leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j\leq n}; (\bot)^*; 0^*)\} \\ \cup \{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{v})^*[\mathsf{result} \mapsto \hat{w}]|\!\})\}$$

Equation 5.56 and Equation 5.57 prove that $(\!|P|\!) \cup \Delta \vdash \Delta'$.

Let K_1' be an fresh empty local heap. We take $G' = G[\ell \mapsto H(\ell)[result \mapsto w']]] \cup H'$, $(K_l')_l = K_1' :: K_1 :: (K)_{l>3}$ and $(\mathsf{lk}'^{l,j})_{l,j} = ((\{(\ell \mapsto 0) \mid \ell\}) :: \varepsilon) :: (\mathsf{lk}^{1,j})_j :: (\mathsf{lk}^{l,j})_{l>3,j}$.

Recall that $\ell \in G$, therefore $w = H(\ell).result$ is either a primitive value or in $dom(G)$. Besides $\Delta :> \beta^G_{Heap}(H)$ and $ser^H_{Val}(w) = (w', H')$, therefore by applying Lemma 26 we know that $\Delta :> \beta^{G\cup H'}_{Heap}(H')$ and that $G \cup H', (K_i)_i$ is a heap decomposition of $H \cup H' \cdot S$.

By Lemma 25 we know that for all i, $w' \notin dom(K_i)$, therefore $G', (K_i)_i$ is a heap decomposition of $H'' \cdot S$. Since K_0' is a fresh empty local heap we get from this that $G', (K_i')_i$ is a heap decomposition of $H'' \cdot S$.

Using Assumption 9, it is simple to check that $(G', (K_i', (\mathsf{lk}'^{i,j})_j)_i)$ is a configuration decomposition of Ψ'.

Let X' be the corresponding set of facts in $\beta_{Cnf}(\Psi')$:

$$X' = \beta^{G'}_{Stk}(\underline{\langle \ell, s, \varepsilon, \gamma, \alpha_{\ell.onActivityResult}\rangle} :: \varphi' :: \Omega, \Xi, (K_l', (\mathsf{lk}'^{l,j})_j)_l) \\ \cup \beta^{G'}_{Heap}(H'') \cup \beta_{Stat}(S)$$

We are going to prove that X' is over-approximated by the set of abstract facts Δ'. Similarly to what we did in the previous cases, one can check that:

$$X' \backslash X = \beta^{G'}_{Frm}(\underline{\langle \ell, s, \varepsilon, \gamma, \alpha_{\ell.onActivityResult}\rangle}, K_1', (\mathsf{lk}'^{1,j})_j) \cup \beta^{G'}_{Heap}(H'')$$

And besides:

$$\beta^{G'}_{Heap}(H'') = \beta^G_{Heap}(H_{|dom(H)\backslash \ell}) \cup \beta^{\mathsf{G}\cup\mathsf{H}'}_{Heap}(H') \cup \mathsf{H}(c, \beta_{Blk}(H(\ell)[result \mapsto w']]))$$

$$\begin{aligned} &\mathsf{H}(c, \beta_{Blk}(H(\ell)[result \mapsto w']])) = \\ &\qquad \mathsf{H}(c, \beta_{Blk}(H(\ell))[result \mapsto \beta_{Val}(w')]])) = \text{(by lemma 23)} \\ &\qquad \mathsf{H}(c, \beta_{Blk}(H(\ell))[result \mapsto \beta_{Val}(w)]])) <: \text{(by Proposition 5)} \\ &\qquad \mathsf{H}(c, \hat{b}[result \mapsto \hat{w}]])) <: \Delta' \end{aligned} \tag{5.58}$$

We already saw that $\beta^{\mathsf{G}\cup\mathsf{H}'}_{Heap}(H') <: \Delta <: \Delta'$. Moreover $\beta^G_{Heap}(H_{|dom(H)\backslash\ell}) \subseteq \beta^G_{Heap}(H) <: \Delta <: \Delta'$. These two fact and Equation 5.58 show that $\beta^{G'}_{Heap}(H'') <: \Delta'$.

We can also check that:

$$\begin{aligned}&\beta^{G'}_{Frm}(\underline{\langle \ell, s, \varepsilon, \gamma, \alpha_{\ell.onActivityResult}\rangle}, K'_1, (\mathsf{lk}'^{1,j})_j)\\ &\quad <: \mathsf{LState}_{\mathsf{c}'',\mathsf{m},0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j\leq n}); (\hat{\mathbf{0}}_k)^{k\leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j\leq n}; (\bot)^*; 0^*) <: \Delta'\end{aligned}$$

Hence $X'\backslash X <: \Delta'$. We conclude by observing that since $X <: \Delta <: \Delta'$ and $X' \subseteq X \cup (X'\backslash X)$ we have $X' <: \Delta'$.

- Rule applied is (A-ThreadStart):

(A-ThreadStart)

$$\frac{\begin{array}{c}\varphi = \underline{\langle \ell, s, \pi, \ell'' :: \gamma, \alpha\rangle} \qquad \varphi' = \underline{\langle \ell, s, \pi, \gamma, \alpha\rangle}\\ \psi = \langle\!\langle \ell, \ell'', \varepsilon, \varepsilon, \alpha'\rangle\!\rangle \qquad H(\ell'') = \{\!|c'; (f \mapsto v)^*|\!\} \qquad lookup(c', \mathsf{run}) = (c'', st^*)\\ sign(c'', \mathsf{run}) = \tau \xrightarrow{loc} \tau' \qquad \alpha' = \langle c'', \mathsf{run}, 0 \cdot \ell'' \cdot st^* \cdot (r_k \mapsto \mathbf{0})^{k\leq loc}, r_{loc+1} \mapsto \ell''\rangle\end{array}}{\Omega :: \varphi :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \varphi' :: \Omega' \cdot \psi :: \Xi \cdot H \cdot S}$$

Given that $X <: \Delta$, we have $\Delta :> \beta^G_{Pthr}(\ell'' :: \gamma)$. Moreover $H(\ell'') = \{\!|c'; (f \mapsto v)^*|\!\}$, therefore there exists $\mathsf{T}(\lambda, \hat{b}) \in \Delta$ such that $\lambda = \beta_{Lab}(\ell'')$ and $\beta_{Blk}(\{\!|c'; (f \mapsto v)^*|\!\}) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}$. This implies that $\hat{b} = \{\!|c'; \hat{v}^*|\!\}$ for some $\hat{v}^*$ such that $\forall i, \beta_{Val}(v_i) \sqsubseteq^{\mathsf{nfs}} \hat{v}_i$.

By well-formedness we get that $c' \leq \mathsf{Thread}$, and by Assumption 8 we know that $lookup(c', \mathsf{run}) = (c'', st^*)$ implies that $c' \leq c''$. Moreover since $lookup(c', \mathsf{run}) = (c'', st^*)$ we know that $c'' \in \widehat{lookup}(\mathsf{run})$, hence we can use the rule *Tstart* included in $(\!|P|\!)$:

$$\begin{aligned}&\mathsf{T}(\lambda, \{\!|c'; (f \mapsto \_)^*|\!\}) \wedge c' \leq \mathsf{c}'' \wedge c' \leq \mathsf{Thread}\\ &\qquad \Longrightarrow \mathsf{LState}_{\mathsf{c}'',\mathsf{run},0}((\mathsf{NFS}(\lambda), \mathsf{NFS}(\lambda)); (\hat{\mathbf{0}}_k)^{k\leq loc}, \mathsf{NFS}(\lambda); (\bot)^*; 0^*)\end{aligned} \quad (5.59)$$

We define the set of abstract fact:

$$\Delta' \;=\; \Delta \cup \{\mathsf{LState}_{\mathsf{c}'',\mathsf{run},0}((\mathsf{NFS}(\lambda), \mathsf{NFS}(\lambda)); (\hat{\mathbf{0}}_k)^{k\leq loc}, \mathsf{NFS}(\lambda); (\bot)^*; 0^*)\}$$

From Equation 5.59 we get that $(\!|P|\!) \cup \Delta \vdash \Delta'$.

Let n be the length of $\Omega :: \varphi :: \Omega'$, and m the length of Ξ. Let K'_t be an fresh empty local heap. We take $G' = G$ and :

$$\begin{aligned}(K'_l, (\mathsf{lk}'^{l,j})_j)_{l\leq n+m+1} = (K_l, (\mathsf{lk}^{l,j})_j)_{l\leq n} ::&\\ (K'_t, ((\{(\ell \mapsto 0) \mid \ell\}) :: \varepsilon)) :: (K_l, (\mathsf{lk}^{l,j})_j)_{n+1\leq l\leq n+m}&\end{aligned}$$

Since $(G, (K_i, (\mathsf{lk}^{i,j})_j)_i)$ is a configuration decomposition of Ψ we know that $\ell'' \in dom(G)$. With this one can check that $(G', (K'_i, (\mathsf{lk}'^{i,j})_j)_i)$ is a configuration decomposition of Ψ'.

Let $X' \in \beta_{Cnf}(\Psi')$ be the corresponding set of facts:

$$\beta^{G'}_{Stk}(\Omega :: \varphi' :: \Omega', \psi :: \Xi, (K'_l, (\mathsf{lk}'^{l,j})_j)_l) \cup \beta^{G'}_{Heap}(H) \cup \beta_{Stat}(S)$$

Let n_0 be such that Ω is of length $n_0 - 1$. It is quite easy to check that:

$$X' \backslash X \subseteq \beta^{G'}_{Frm}(\underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}, K'_{n_0}, (\mathsf{lk}'^{n_0,j})_j) \cup \beta^{G'}_{Frm}(\langle\!\langle \ell, \ell'', \varepsilon, \varepsilon, \alpha' \rangle\!\rangle, K'_{n+1}, (\mathsf{lk}'^{n+1,j})_j)$$

Since $\ell'' \in dom(G)$, we have that:

$$\begin{aligned} \Delta' \quad &:> \{\mathsf{LState}_{\mathsf{c}'',\mathsf{run},0}((\mathsf{NFS}(\lambda), \mathsf{NFS}(\lambda)); (\hat{\mathbf{0}}_k)^{k \le loc}, \mathsf{NFS}(\lambda); (\bot)^*; 0^*)\} \\ &:> \beta^{G'}_{Frm}(\langle\!\langle \ell, \ell'', \varepsilon, \varepsilon, \alpha' \rangle\!\rangle, K'_{n+1}, (\mathsf{lk}'^{n+1,j})_j) \end{aligned}$$

Moreover since ϕ' only differ from ϕ in the fact that it has a smaller thread stack, we have:

$$\beta^{G'}_{Frm}(\underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}, K'_{n_0}, (\mathsf{lk}'^{n_0,j})_j) \subseteq \beta^{G}_{Frm}(\underline{\langle \ell, s, \pi, \ell'' :: \gamma, \alpha \rangle}, K_{n_0}, (\mathsf{lk}^{n_0,j})_j) <: \Delta$$

This proves that $X' :> \Delta'$.

- Rule applied is (T-Reduce):

  (T-Reduce)
  $$\frac{\ell' \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S \rightsquigarrow \ell' \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S'}{\Omega \cdot \Xi :: \langle\!\langle \ell, \ell', \pi, \gamma, \alpha \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H' \cdot S'}$$

  Exactly like the (A-Reduce) case.

- Rule applied is (T-Kill):

  (T-Kill)
  $$\frac{\begin{array}{c} H(\ell') = \{\!|c; (f \mapsto v)^*, \mathsf{finished} \mapsto \_|\!\} \\ H' = \ H[\ell' \mapsto \{\!|c; (f \mapsto v)^*, \mathsf{finished} \mapsto \mathtt{true}|\!\}] \end{array}}{\Omega \cdot \Xi :: \langle\!\langle \ell, \ell', \varepsilon, \varepsilon, \overline{\alpha} \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega \cdot \Xi :: \Xi' \cdot H' \cdot S}$$

  Exactly like the (A-Destroy) case.

- Rule applied is (T-Intent):

  (T-Intent)
  $$\frac{(\varphi, \varphi') \in \{(\langle \ell, s, \pi, \gamma, \alpha \rangle, \langle \ell, s, i :: \pi, \gamma, \alpha \rangle), (\underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}, \underline{\langle \ell, s, i :: \pi, \gamma, \alpha \rangle})\}}{\begin{array}{c} \Omega :: \varphi :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', i :: \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \\ \Omega :: \varphi' :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S \end{array}}$$

  Trivial since there are no changes to the abstraction: $\beta_{Cnf}(\Psi) = \beta_{Cnf}(\Psi')$.

- Rule applied is (T-Thread):

$$\frac{(\varphi, \varphi') \in \{(\langle \ell, s, \pi, \gamma, \alpha\rangle, \langle \ell, s, \pi, \ell_t :: \gamma, \alpha\rangle), (\underline{\langle \ell, s, \pi, \gamma, \alpha\rangle}, \underline{\langle \ell, s, \pi, \ell_t :: \gamma, \alpha\rangle})\}}{\begin{array}{c}\Omega :: \varphi :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \ell_t :: \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \\ \Omega :: \varphi' :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi \cdot H \cdot S\end{array}}\ \text{(T-Thread)}$$

Trivial since there are no changes to the abstraction: $\beta_{Cnf}(\Psi) = \beta_{Cnf}(\Psi')$.

□

CHAPTER 6

A Semantic Framework for the Static Analysis of Ethereum smart contracts

6.1 Introduction

One of the determining factors for the growing interest in blockchain technologies is the groundbreaking promise of secure distributed computations even in the absence of trusted third parties. Building on a distributed ledger that keeps track of previous transactions and the state of each account, whose functionality and security is ensured by a delicate combination of incentives and cryptography, software developers can implement sophisticated distributed, transactions-based computations by leveraging the scripting language offered by the underlying cryptocurrency. While many of these cryptocurrencies have an intentionally limited scripting language (e.g., Bitcoin [Nak08]), Ethereum was designed from the ground up with a quasi Turing-complete language[1]. Ethereum programs, called *smart contracts*, have thus found a variety of appealing use cases, such as financial contracts [BKT17], auctions [HSLC17], elections [MFSH17], data management systems [Adh17], trading platforms [NGW17, MM17], permission management [AEVL16] and verifiable cloud computing [DWA$^+$17], just to mention a few. Given their financial nature, bugs and vulnerabilities in smart contracts may lead to catastrophic consequences. For instance, the infamous DAO vulnerability [thea] recently led to a 60M$ financial loss and similar vulnerabilities occur regularly [par17a, par17b]. Furthermore, many smart contracts in the wild are intentionally fraudulent, as highlighted in a recent survey [ABC17].

A rigorous security analysis of smart contracts is thus crucial for the trust of the society in blockchain technologies and their widespread deployment. Unfortunately, this task is a quite challenging since smart contracts are uploaded on the blockchain in the form of Ethereum Virtual Machine (EVM) bytecode, a stack-based low-level code featuring dynamic code creation and invocation and, in general, very little static information, which makes it extremely difficult to analyse.

Related Work Recognizing the importance of solid semantic foundations for smart contracts, the Ethereum foundation published a yellow paper [Woo14] to describe the intended behaviour of smart contracts. This semantics, however, exhibits several under-specifications and does not follow any standard approach for the specification of program semantics, thereby hindering program verification. In order to provide a more precise characterisation, Hirai formalizes the EVM semantics in the proof assistant Isabelle/HOL and uses it for manually proving safety properties for concrete programs [Hir17]. This semantics, however, constitutes just a sound over-approximation of the original semantics [Woo14]. More specifically, once a contract performs a call that is not a self-call, it is assumed that arbitrary code gets executed and consequently arbitrary changes to the account's state and to the global state can be performed. Consequently, this semantics can not serve as a general-purpose basis for static analysis techniques that might not rely on the same over-approximation.

[1]While the language itself is Turing complete, computations are associated with a bounded computational budget (called gas), which gets consumed by each instruction thereby enforcing termination.

In a concurrent work, Hildebrandt et al. [HSR+18] define the EVM semantics in the $\mathbb{K}$ framework [SPY+16] – a language-independent verification framework based on reachability logics. The authors leverage the power of the $\mathbb{K}$ framework in order to automatically derive analysis tools for the specified semantics, presenting as an example a gas analysis tool, a semantic debugger, and a program verifier based on reachability logics. The underlying semantics relies on non-standard local rewriting rules on the system configuration. Since parts of the execution are treated in separation such as the exception behaviour and the gas calculations, one small-step consists of several rewriting steps, which makes this semantics harder to use as a basis for new static analysis techniques. This is relevant whenever the static analysis tools derivable by the $\mathbb{K}$ framework are not sufficient for the desired purposes: for instance, their analysis requires the user to manually specify loop invariants, which is hardly doable for EVM bytecode and clearly does not scale to large programs.

Bhargavan et al. [BDLF+16] introduce a framework to analyse Ethereum contracts by translation into F*, a functional programming language aimed at program verification and equipped with an interactive proof assistant. The translation supports only a fragment of the EVM bytecode and does not come with a justifying semantic argument.

Luu et al. have recently presented Oyente [LCO+16], a state-of-the-art static analysis tool for EVM bytecode that relies on symbolic execution. Oyente comes with a semantics of a simplified fragment of the EVM bytecode and, in particular, misses several important commands related to contract calls and contract creation. Furthermore, it is affected by a major bug related to calls as well as several other minor ones which we discovered while formalizing our semantics, which is inspired by theirs.

Our Contributions This chapter lays the semantic foundations for Ethereum smart contracts. Specifically, we introduce

- The first complete small-step semantics for EVM bytecode;
- A formalisation in F* of a large fragment of our semantics, which can serve as a foundation for verification techniques based on encoding into this language [BDLF+16] as well as machine-checked proofs for other analysis techniques (e.g., [LCO+16]). By compiling F* in OCaml, we could successfully validate our semantics against the official Ethereum test suite;

The complete semantics as well as the formalisation in F* are publicly available [GMS18b].

6.2 Background on Ethereum

Ethereum

Ethereum is a cryptographic currency system built on top of a blockchain. Similar to Bitcoin, network participants publish transactions to the network that are then grouped into blocks by distinct nodes (the so-called *miners*) and appended to the blockchain using a proof of work (PoW) consensus mechanism. The state of the system – that we will also refer to as *global state* – consists of the state of the different accounts populating it. An account can either be an external account (belonging to a user of the system) that carries information on its current balance or it can be a contract account that additionally obtains persistent storage and the contract's code. The account's balances are given in the subunit *wei* of the virtual currency *Ether*.[2]

Transactions can alter the state of the system by either creating new contract accounts or by calling an existing account. Calls to external accounts can only transfer Ether to this account, but calls to contract accounts additionally execute the code associated with the contract. The contract execution might alter the storage of the account or might again perform transactions – in this case we talk about *internal transactions*.

The execution model underlying the execution of contract code is described by a virtual state machine, the *Ethereum Virtual Machine* (EVM). This is *quasi Turing complete* as the otherwise Turing complete execution is restricted by the upfront defined resource *gas* that effectively limits the number of execution steps. The originator of the transaction can specify the maximal gas that should be spent for the contract execution and also determines the gas prize (the amount of wei to pay for a unit of gas). Upfront, the originator pays for the gas limit according to the gas prize and in case of successful contract execution that did not spend the whole amount of gas dedicated to it, the originator gets reimbursed with gas that is left. The remaining wei paid for the used gas are given as a fee to a beneficiary address specified by the miner.

EVM Bytecode

The code of contracts is written in *EVM bytecode* – an Assembler like bytecode language. As the core of the EVM is a stack-based machine, the set of instructions in EVM bytecode consists mainly of standard instructions for stack operations, arithmetics, jumps and local memory access. The classical set of instructions is enriched with an opcode for the SHA3 hash and several opcodes for accessing the environment that the contract was called in. In addition, there are opcodes for accessing and modifying the storage of the account currently running the code and distinct opcodes for performing internal calls and create transactions. Another instruction particular to the blockchain setting is the SELFDESTRUCT code that deletes the currently executed contract - but only after the successful execution of the external transaction.

Gas and Exceptions The execution of each instruction consumes a positive amount of gas. There is a gas limit set by the sender of the transaction. Exceeding the gas limit results in an exception that reverts the effects of the current transaction on the global state. In the case of nested transactions, the occurrence of an exception only reverts its

[2]One Ether is equivalent to 10^{18} wei.

own effects, but not those of the calling transaction. Instead, the failure of an internal transaction is only indicated by writing zero to the caller's stack.

6.3 Small-Step Semantics

We introduce a small-step semantics covering the full EVM bytecode, inspired by the one presented by Luu et al. [LCO+16], which we substantially revise in order to handle the missing instructions, in particular contract calls and call creation. In addition, while formalizing our semantics, we found a major flaw related to calls and several minor ones (cf. § 6.3.8), which we fixed and reported to the authors. In [GMS18b] we present the formal semantic rules specification, below we discuss the most significant ones.

6.3.1 Preliminaries

In the following, we will use $\mathbb{B}$ to denote the set $\{0, 1\}$ of bits and accordingly $\mathbb{B}^x$ for sets of bitstrings of size x. We further let $\mathbb{N}_x$ denote the set of non-negative integers representable by x bits and allow for implicit conversion between those two representations. In addition, we will use the notation $[X]$ (resp. $\mathcal{L}(X)$) for arrays (resp. lists) of elements from the set X. We use standard notations for operations on arrays and lists.

6.3.2 Global state

As mentioned before, the global state is a (partial) mapping from account addresses (that are bitstrings of size 160) to accounts. In the case that an account does not exist, we assume it to map to $\bot$. Accounts, irrespectively of their type, are tuples of the form $(n, b, \mathit{stor}, \mathit{code})$, with $n \in \mathbb{N}_{256}$ being the account's nonce that is incremented with every other account that the account creates, $b \in \mathbb{N}_{256}$ being the account's balance in *wei*, $\mathit{stor} \in \mathbb{B}^{256} \to \mathbb{B}^{256}$ being the accounts persistent storage that is represented as a mapping from 256-bit words to 256-bit words and finally $\mathit{code} \in [\mathbb{B}^8]$ being the contract that is an array of bytes. In contrast to contract accounts, external accounts have the empty bytearray as code. As only the execution of code in the context of the account can access and modify the account's storage, the fact that formally external accounts have persistent storage does not have any effect. In the following, we will denote the set of addresses with $\mathcal{A}$ and the set of global states with Σ and we will assume that $\sigma \in \Sigma$.

6.3.3 Small-Step Relation

In order to define the small-step semantics, we give a small-step relation $\Gamma \vDash S \to S'$ that specifies how a call stack $S \in \mathbb{S}$ representing the state of the execution evolves within one step under the transaction environment $\Gamma \in \mathcal{T}_{env}$.

In Figure 6.1 we give a full grammar for call stacks and transaction environments:

$$\begin{array}{rllll}
\text{Call stacks} & \mathbb{S} & \ni & S & := & \mathit{EXC} :: S_{plain} \mid \mathit{HALT}(\sigma, d, g, \eta) :: S_{plain} \mid S_{plain} \\
\text{Plain call stacks} & \mathbb{S}_{plain} & \ni & S_{plain} & := & (\mu, \iota, \sigma, \eta) :: S_{plain} \\
\text{Machine states} & M & \ni & \mu & := & (\mathit{gas}, \mathit{pc}, m, i, s) \\
\text{Execution environments} & I & \ni & \iota & := & (\mathit{actor}, \mathit{input}, \mathit{sender}, \mathit{value}, \mathit{code}) \\
\text{Global states} & \Sigma & \ni & \sigma & & \\
\text{Account states} & \mathbb{A} & \ni & \mathit{acc} & := & (n, b, \mathit{code}, \mathit{stor}) \mid \bot \\
\text{Transaction effects} & N & \ni & \eta & := & (b, L, S_{\dagger}) \\
\text{Transaction environments} & \mathcal{T}_{env} & \ni & \Gamma & := & (o, \mathit{prize}, H)
\end{array}$$

$$\begin{array}{rc}
\text{Notations:} & d \in [\mathbb{B}^8], \quad g \in \mathbb{N}_{256}, \quad \eta \in N, \quad o \in \mathcal{A}, \quad \mathit{prize} \in \mathbb{N}_{256}, \quad H \in \mathcal{H} \\
& \mathit{gas} \in \mathbb{N}_{256}, \quad \mathit{pc} \in \mathbb{N}_{256}, \quad m \in \mathbb{B}^{256}, \rightarrow \mathbb{B}^8 \quad i \in \mathbb{N}_{256}, \quad s \in \mathcal{L}(\mathbb{B}^{256}) \\
& \mathit{sender} \in \mathcal{A} \quad \mathit{input} \in [\mathbb{B}^8] \quad \mathit{sender} \in \mathcal{A} \quad \mathit{value} \in \mathbb{N}_{256} \quad \mathit{code} \in [\mathbb{B}^8] \\
& b \in \mathbb{N}_{256} \quad L \in \mathcal{L}(\mathit{Ev}_{log}) \quad S_{\dagger} \subseteq \mathcal{A} \quad \Sigma = \mathcal{A} \rightarrow \mathbb{A}
\end{array}$$

Figure 6.1: Grammar for call stacks and transaction environments

Transaction Environments

The transaction environment represents the static information of the block that the transaction is executed in and the immutable parameters given to the transaction as the gas prize or the gas limit. More specifically, the transaction environment $\Gamma \in \mathcal{T}_{env} = \mathcal{A} \times \mathbb{N}_{256} \times \mathcal{H}$ is a tuple of the form (o, prize, H) with $o \in \mathcal{A}$ being the address of the account that made the transaction, $\mathit{prize} \in \mathbb{N}_{256}$ denoting the amount of wei that needs to be paid for a unit of gas in this transaction and $H \in \mathcal{H}$ being the header of the block that the transaction is part of. We do not specify the format of block headers here, but just assume a set $\mathcal{H}$ of block headers.

Callstacks

A call stack S is a stack of execution states which represents the state of the execution within one internal transaction. We give a formal definition of the set of possible callstacks $\mathbb{S}$ as follows:

$$\begin{aligned}
\mathbb{S} := \{&\mathit{EXC} :: S_{plain},\ \mathit{HALT}(\sigma, \mathit{gas}, d, \eta) :: S_{plain},\ S_{plain} \\
&\mid \sigma \in \Sigma,\ \mathit{gas} \in \mathbb{N},\ d \in [\mathbb{B}^8],\ \eta \in N\ , S_{plain} \in \mathcal{L}(M \times I \times \Sigma \times N)\}
\end{aligned}$$

Syntactically, a call stack is a stack of regular execution states of the form $(\mu, \iota, \sigma, \eta)$ that can optionally be topped with a halting state $\mathit{HALT}(\sigma, \mathit{gas}, d, \eta)$ or an exception state EXC. We summarize these three types of states as execution states $\mathcal{S}$. Semantically, halting states indicate regular halting of an internal transaction, exception states indicate exceptional halting, and regular execution states describe the state of internal transactions in progress. Halting and exception states can only occur as top elements of the call stack as they represent terminated internal transactions. Exception states of the form EXC do not carry any information as in the case of an exception all effects of the terminated internal transaction are reverted and the caller state therefore stays unaffected, except for the gas. Halting states instead are of the form $\mathit{HALT}(\sigma, \mathit{gas}, d, \eta)$ specifying the global state σ the execution halted in, the gas $gas \in \mathbb{N}_{256}$ remaining from the execution, the return data $d \in [\mathbb{B}^8]$ and the additional transaction effects $\eta \in N$ of the internal transaction. The additional transaction effects carry information that are accumulated during execution, but do not influence the small-step execution itself. Formally, the additional transaction effects are a triple of the form $(b, L, S_\dagger) \in N = \mathbb{N}_{256} \times \mathcal{L}(Ev_{log}) \times \mathcal{P}(\mathcal{A})$ with $b \in \mathbb{N}_{256}$ being the refund balance that is increased by account storage operations and will finally be paid to the transaction's beneficiary, $L \in \mathcal{L}(Ev_{log})$ being the sequence of log events that the bytecode execution invoked during execution and $S_\dagger \subseteq \mathcal{A}$ being the so-called suicide set – the set of account addresses that executed the SELFDESTRUCT command and therefore registered their account for deletion. The information held by the halting state is carried over to the calling state.

The state of a non-terminated internal transaction is described by a regular execution state of the form $(\mu, \iota, \sigma, \eta)$. The state is determined by the current global state σ of the system as well as the execution environment $\iota \in I$ that specifies the parameters of the current transaction (including inputs and the code to be executed), the local state $\mu \in M$ of the stack machine, and the transaction effects $\eta \in N$ collected during execution so far.

Execution Environment

The execution environment ι of an internal transaction specifies the static parameters of the transaction. It is a tuple of the form $(\mathit{actor}, \mathit{input}, \mathit{sender}, \mathit{value}, \mathit{code}) \in I = \mathcal{A} \times [\mathbb{B}^8] \times \mathcal{A} \times \mathbb{N}_{256} \times [\mathbb{B}^8]$ with the following components:

- $\mathit{actor} \in \mathcal{A}$ is the address of the account currently executing;
- $\mathit{input} \in [\mathbb{B}^8]$ is the data given as an input to the internal transaction;
- $\mathit{sender} \in \mathcal{A}$ is the address of the account that initiated the internal transaction;
- $\mathit{value} \in \mathbb{N}_{256}$ is the value transferred by the internal transaction;
- $\mathit{code} \in [\mathbb{B}^8]$ is the code currently executed.

This information is determined at the beginning of an internal transaction execution and it can be accessed, but not altered during the execution.

Machine State

The local machine state μ represents the state of the underlying state machine used for execution and is a tuple of the form $(\mathit{gas}, \mathit{pc}, m, i, s)$ where

- $\mathit{gas} \in \mathbb{N}_{256}$ is the current amount of gas still available for execution;
- $\mathit{pc} \in \mathbb{N}_{256}$ is the current program counter;

- $m \in \mathbb{B}^{256} \to \mathbb{B}^8$ is a mapping from 256-bit words to bytes that represents the local memory;
- $i \in \mathbb{N}_{256}$ is the current number of active words in memory;
- $s \in \mathcal{L}(\mathbb{B}^{256})$ is the local 256-bit word stack of the stack machine.

The execution of each internal transaction starts in a fresh machine state, with an empty stack, memory initialized to all zeros, and program counter and active words in memory set to zero. Only the gas is instantiated with the gas value available for the execution.

6.3.4 Auxiliary Definitions

For extracting the command that is currently executed, the instruction at position $\mu.\mathsf{pc}$ of the code code provided in the execution environment needs to be accessed. For the sake of presentation, we define a function doing so:

Definition 29 (Currently executed command)**.** The currently executed command in the machine state μ and execution environment ι is denoted by $\omega_{\mu,\iota}$ and defined as follows:

$$\omega_{\mu,\iota} := \begin{cases} \iota.\mathsf{code}\,[\mu.\mathsf{pc}] & \mu.\mathsf{pc} < |\iota.\mathsf{code}| \\ \mathsf{STOP} & \text{otherwise} \end{cases}$$

All EVM instructions have in common that running out of gas as well as over and under flows of the local machine stack cause an exception. We define a function $\mathit{valid}\,(\cdot,\cdot,\cdot) : \mathbb{N}_{256} \times \mathbb{N}_{256} \times \mathbb{N} \to \mathbb{B}$ that given the available gas, the instruction cost and the new stack size determines whether one of the conditions mentioned above is satisfied. We do not check for stack underflows as this is realized by pattern matching in the individual small step rules.

$$\mathit{valid}\,(g, c, s) := \begin{cases} 1 & g \geq c \wedge s < 1024 \\ 0 & \mathit{otherwise} \end{cases}$$

We also write $\mathit{valid}\,(g, c, s)$ for $\mathit{valid}\,(g, c, s) = 1$ and $\neg\mathit{valid}\,(g, c, s)$ for $\mathit{valid}\,(g, c, s) = 0$.

In EVM bytecode jump potential destinations are explicitly marked by the distinct $\mathsf{JUMPDEST}$ instruction. Jumps to other destination cause an exception. For simplifying this check, we define the set of valid jump destinations as follows:

Definition 30. Valid jump destinations [Woo14]. $D\,(\cdot) : [\mathbb{B}^8] \to \mathcal{P}(\mathbb{N})$ determines the set of valid jump destinations given the code $\mathit{code} \in [\mathbb{B}^8]$, that is being run. It is defined as any position in the code occupied by a $\mathsf{JUMPDEST}$ instruction. Formally $D\,(c) = D_H\,(c, 0)$, where:

$$D_H(\cdot,\cdot) : [\mathbb{B}^8] \times \mathbb{N} \to \mathcal{P}(\mathbb{N})$$

$$D_H(c,i) := \begin{cases} \emptyset & i \geq |c| \\ \{i\} \cup D_H(c, N(i, c[i])) & c[i] = \mathsf{JUMPDEST} \\ D_H(c, N(i, c[i])) & \text{otherwise} \end{cases}$$

where $N(\cdot,\cdot) : \mathbb{N} \times \mathbb{B}^8 \to \mathbb{N}$ is the next valid instruction position in the code, skipping the data of a $\mathsf{PUSH}n$ instruction, if any:

$$N(i,\omega) := \begin{cases} i+n+1 & \omega = \mathsf{PUSH}n \\ i+1 & \text{otherwise} \end{cases}$$

6.3.5 Small-Step Rules

In the following, we will present a selection of interesting small-step rules in order to illustrate the most important features of the semantics.

Binary Stack Operations For demonstrating the overall design of the semantics, we start with the example of the arithmetic expression ADD performing addition of two values on the machine stack. Note that as the word size of the stack machine is 256, all arithmetic operations are performed modulo 2^{256}.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{ADD} \qquad \mu.\mathsf{s} = a :: b :: s \\ \mathit{valid}(\mu.\mathsf{gas}, 3, |s|+1) \qquad \mu' = \mu[\mathsf{s} \to (a+b) :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 3]\end{array}}{\Gamma \vDash (\mu,\iota,\sigma,\eta) :: S \to (\mu',\iota,\sigma,\eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{ADD} \qquad (\neg \mathit{valid}(\mu.\mathsf{gas}, 3, |s|+1) \lor |\mu.\mathsf{s}| < 2)}{\Gamma \vDash (\mu,\iota,\sigma,\eta) :: S \to \mathit{EXC} :: S}$$

We use a dot notation, in order to access components of the different state parameters. We name the components with the variable names introduced for these components in the last section written in sans-serif-style. In addition, we use the usual notation for updating components: $t[\mathsf{c} \to v]$ denotes that the component c of tuple t is updated with value v. For expressing incremental updates in a simpler way, we additionally use the notation $t[\mathsf{c} \mathrel{+}= v]$ to denote that the (numerical) component of c is incremented by v and similarly $t[\mathsf{c} \mathrel{-}= v]$ for decrementing a component c of t.

The execution of the arithmetic instruction ADD only performs local changes in the machine state affecting the local stack, the program counter, and the gas budget. For deciding upon the correct instruction to execute, the currently executed code (that is part of the execution environment) is accessed at the position of the current program counter. The cost of an ADD instruction is constantly three units of gas that get subtracted from the gas budget in the machine state. As every other instruction, ADD can fail due to lacking gas or due to underflows on the machine stack. In this case, the exception state is entered and the execution of the current internal transaction is terminated. For better readability, we use here the slightly sloppy $\vee$ notation for combining the two error cases in one inference rule.

The rules for the other arithmetic operations and comparison operations (e.g., less than LT, modulo MOD) are similar to the ones for ADD as all these instructions remove their argument(s) from the local stack and then put their result to the local stack, consuming a constant amount of gas.

Stack Operations There are 32 instructions for pushing values to the stack. We summarize the behaviour of all these instructions with the following rules by parameterising the instruction with number of following bytecodes that are pushed to the stack. The $\mathsf{PUSH}n$ (with $m \in [1, 32]$) command pushes the bytecodes at the next n program counter position to the stack.

$$\frac{\begin{array}{ccc} & \omega_{\mu,\iota} = \mathsf{PUSH}n & \\ k = \mathit{min}\,(|\iota.\mathit{code}|, \mu.\mathsf{pc} + x) & \mathit{valid}\,(\mu.\mathsf{gas}, 3, |\mu.\mathsf{s}| + 1) & d = \iota.\mathit{code}\,[\mu.\mathsf{pc} + 1, k] \\ d' = d \cdot 0^{8\cdot(32-(k-\mu.\mathsf{pc}))} & \mu' = \mu[\mathsf{s} \to d' :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= (x+1)][\mathsf{gas} \mathrel{-}= 3] & \end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

Jumps The JUMP command updates the program counter to i (specified in the stack) if i is a valid jump destination.

$$\frac{\begin{array}{ccc} & \omega_{\mu,\iota} = \mathsf{JUMP} & \mathit{valid}\,(\mu.\mathsf{gas}, 8, |s|) \\ \mu.\mathsf{s} = i :: s & i \in D\,(\iota.\mathit{code}) & \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \to i][\mathsf{gas} \mathrel{-}= 8] \end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

The conditional jump command JUMPI conditionally jumps to position i depending on b.

$$\frac{\begin{array}{ccc} \omega_{\mu,\iota} = \mathsf{JUMPI} & \mathit{valid}\,(\mu.\mathsf{gas}, 10, |s|) & \mu.\mathsf{s} = i :: b :: s \\ i \in D\,(\iota.\mathit{code}) & j = (b = 0)\,?\,\mu.\mathsf{pc} + 1 \;:\; i & \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \to j][\mathsf{gas} \mathrel{-}= 10] \end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{ccc} & \omega_{\mu,\iota} = \mathsf{JUMPI} & \mathit{valid}\,(\mu.\mathsf{gas}, 10, |s|) \\ \mu.\mathsf{s} = i :: 0 :: s & i \notin D\,(\iota.\mathit{code}) & \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \to \mu.\mathsf{pc} + 1][\mathsf{gas} \mathrel{-}= 10] \end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

The JUMPDEST command marks a valid jump destination. It does not trigger any execution and consequently the only effect of the command is the increasing of the program counter and charging the fee for the command execution.

$$\frac{\omega_{\mu,\iota} = \mathsf{JUMPDEST} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 1, |\mu.\mathsf{s}|) \qquad \mu' = \mu[\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

Accessing the Transaction Environment Several instructions allow for accessing the transaction environment information (e.g., BLOCKHASH, NUMBER). The BLOCKHASH command writes the hash of one of the 256 most recently completed blocks (that is specified on the stack by the previous execution of NUMBER opcode) to the stack:

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{NUMBER}\\ \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \rightarrow (\Gamma.H).\mathsf{number} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{BLOCKHASH} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 20, |\mu.\mathsf{s}|)\\ \mu.\mathsf{s} = n :: s \qquad h = P\,(\iota.\mathsf{parent}, n, 0) \qquad \mu' = \mu[\mathsf{s} \rightarrow h :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 20]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

Calling A more interesting example of a semantic rule is the one of the CALL instruction that initiates an internal call transaction. In the case of calling, several corner cases need to be treated which results in several inference rules for this case. Here, we only present one rule for illustrating the main functionality. More precisely, we present the case in that the account that should be called exists, the call stack limit of 1024 is not reached yet, and the account initiating the transaction has a sufficiently large balance for sending the specified amount of wei to the called account.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: \mathit{to} :: \mathit{va} :: \mathit{io} :: \mathit{is} :: \mathit{oo} :: \mathit{os} :: s\\ \mathit{to}_a = \mathit{to} \mod 2^{160} \qquad \sigma(\mathit{to}_a) \neq \bot \qquad |A| + 1 \leq 1024\\ \sigma(\iota.\mathsf{actor}).\mathsf{b} \geq \mathit{va} \qquad \mathit{aw} = M\,(M\,(\mu.\mathsf{i}, \mathit{io}, \mathit{is}), \mathit{oo}, \mathit{os}) \qquad c_{call} = C_{gascap}\,(\mathit{va}, 1, g, \mu.\mathsf{gas})\\ c = C_{base}\,(\mathit{va}, 1) + C_{mem}\,(\mu.\mathsf{i}, \mathit{aw}) + c_{call} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, c, |s| + 1)\\ \sigma' = \sigma\langle \mathit{to}_a \rightarrow \sigma(\mathit{to}_a)[\mathsf{b} \mathrel{+}= \mathit{va}]\rangle\langle \iota.\mathsf{actor} \rightarrow \sigma(\iota.\mathsf{actor})[\mathsf{b} \mathrel{-}= \mathit{va}]\rangle\\ d = \mu.\mathsf{m}\,[\mathit{io}, \mathit{io} + \mathit{is} - 1] \qquad \mu' = (c_{call}, 0, \lambda x.\, 0, 0, \epsilon)\\ \iota' = \iota[\mathsf{sender} \rightarrow \iota.\mathsf{actor}][\mathsf{actor} \rightarrow \mathit{to}_a][\mathsf{value} \rightarrow \mathit{va}][\mathsf{input} \rightarrow d][\mathsf{code} \rightarrow \sigma(\mathit{to}_a).\mathsf{code}]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota', \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

For performing a call, the parameters to this call need to be specified on the machine stack. These are the amount of gas g that should be given as a budget to the call, the recipient *to* of the call and the amount *va* of wei to be transferred with the call. In addition, the caller needs to specify the input data that should be given to the transaction and the place in memory where the return data of the call should be written after successful execution. To this end, the remaining arguments specify the offset and size of the memory fragment that input data should be read from (determined by *io* and *is*) and return data should be written to (determined by *oo* and *os*).

Calculating the cost in terms of gas for the execution is quite complicated in the case of CALL as it is influenced by several factors including the arguments given to the call and the current machine state. First of all, the gas that should be given to the call (here denoted by c_{call}) needs to be determined. This value is not necessarily equal to the value g specified on the stack, but also depends on the value *va* transferred by the call and the currently available gas. In addition, as the memory needs to be accessed for reading the input value and writing the return value, the number of active words in memory might be increased. This effect is captured by the memory extension function M. As accessing additional words in memory costs gas, this cost needs to be taken into account in the overall cost. The costs resulting from an increase in the number of active words are calculated by the function C_{mem}. Finally, there is also a base cost charged for the call that depends on the value *va*. As the cost also depends on the specific case for calling that is considered, the cost calculation functions receive a flag (here 1) as arguments. These technical details are spelled out in the full version [GMS18b].

The call itself then has several effects: First, it transfers the balance from the executing state (*actor* in the execution environment) to the recipient (*to*). To this end, the global state is updated. Here we use a special notation for the functional update on the global state using $\langle\rangle$ instead of $[]$. Second, for initializing the execution of the initiated internal transaction, a new regular execution state is placed on top of the execution stack. The internal transaction starts in a fresh machine state at program counter zero. This means that the initial memory is initialized to all zeros and consequently the number of active words in memory is zero as well and additionally the initial stack is empty. The gas budget given to the internal transaction is c_{call} calculated before. The transaction environment of the new call records the call parameters. This includes the sender that is the currently executing account *actor*, the new active account that is now the called account *to* as well as the value *va* sent and the input data given to the call. To this end the input data is extracted from the memory using the offset *io* and the size *is*. We use an interval notation here to denote that a part of the memory is extracted. Finally, the code in the execution environment of the new internal transaction is the code of the called account.

Note that the execution state of the caller stays completely unaffected at this stage of the execution. This is a conscious design decision in order to make the semantics more suitable to abstractions.

Besides CALL there are two different instructions for initiating internal call transactions that implement slight variations of the simple CALL instruction. These variations are called CALLCODE and DELEGATECALL, which both allow for executing another's account code in the context of the caller. The difference is that in the case of CALLCODE a new internal transaction is started and the currently executed account is registered as the sender of this transaction while in the case of DELEGATECALL an existing call is really forwarded in the sense that the sender and the value of the initiating transaction are propagated to the new internal transaction.

Analogously to the instructions for initiating internal call transactions, there is also one instruction CREATE that allows for the creation of a new account. The semantics of this instruction is similar to the one of CALL, with the exception that a fresh account is created, which gets the specified transferred value, and that the input provided to this internal transaction, which is again specified in the local memory, is interpreted as the initialization code to be executed in order to produce the newly created account's code as output. In contrast to the call transaction, a create transaction does not await a return value, but only an indication of success or failure.

For discussing how to return from an internal transaction, we show the rule for returning from a successful internal call transaction.

$$\frac{\begin{array}{c} \iota.code\,[\mu.\mathsf{pc}] = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \\ flag = \sigma(to) = \bot\,?\,0\,:\,1 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, flag, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, flag) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to 1 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \to \mu.\mathsf{m}[[oo, oo + s - 1] \to d]] \end{array}}{\Gamma \vDash HALT(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma', \eta') :: S}$$

Leaving the caller state unchanged at the point of calling has the negative side effect that the cost calculation needs to be redone at this point in order to determine the new gas value of the caller state. But besides this, the rule is straightforward: the program counter is incremented as usual and the number of active words in memory is adjusted as memory accesses for reading the input and return data have been made. The gas is decreased, meaning that the overall amount of gas c allocated for the execution is subtracted. However, as this cost already includes the gas budget given to the internal transaction, the gas gas that is left after the execution is refunded again. In addition, the return data d is written to the local memory of the caller at the place specified by oo and os. Finally, the value one is written to the caller's stack in order to indicate the success of the internal call transaction. As the execution was successful, as indicated by the halting state, the global state and the transaction effects of the callee are adopted by the caller.

6.3.6 Transaction Execution

The outcome of an external transaction execution does not only consist of the result of the EVM bytecode execution. Before executing the bytecode, the transaction environment and the execution environment are determined from the transaction information and the block header. In the following we assume $\mathcal{T}$ to denote the set of transactions. An (external) transaction $T \in \mathcal{T}$, similar to the internal transactions, specifies a gas limit, a recipient and a value to be transferred. In addition, it also contains the originator and the gas prize that will be recorded in the transaction environment. Finally, it specifies an input to the transaction and the transaction type that can either be a call or a create transaction. The transaction type determines whether the input will be interpreted as input data to a call transaction or as initialization code for a create transaction. In addition to the transaction of the environment initialization, some initial changes on the global state and validity checks are performed. For the sake of presentation we assume in the following a function $\mathit{initialize}\,(\cdot,\cdot,\cdot) \in \mathcal{T} \times \mathcal{H} \times \Sigma \to (\mathcal{T}_{env} \times \mathcal{S}) \cup \{\bot\}$ performing the initialization phase and returning a transaction environment and initial execution state in the case of a valid transaction and $\bot$ otherwise. Similarly, we assume a function $\mathit{finalize}\,(\cdot,\cdot,\cdot) \in T \times \mathcal{S} \times N \times \Sigma$ that given the final global state of the execution, the accumulated transaction effects and the transaction, computes the final effects on the global state. These include for example the deletion of the contracts from the suicide set and the payout to the beneficiary of the transaction.

Formally we can define the execution of a transaction $T \in \mathcal{T}$ in a block with header $H \in \mathcal{H}$ as follows:

$$\frac{\begin{array}{c}(\Gamma, s) = \mathit{initialize}\,(T, H, \sigma) \\ \Gamma \vDash s :: \epsilon \to^{*} s' :: \epsilon \qquad \mathit{final}\,(s') \qquad \sigma' = \mathit{finalize}\,(s', \eta', T)\end{array}}{\sigma \xrightarrow{T,H} \sigma'}$$

where $\to^{*}$ denotes the reflexive and transitive closure of the small-step relation and the predicate $\mathit{final}\,(\cdot)$ characterises a state that cannot be further reduced using the small-step relation.

6.3.7 Formalisation in F*

We provide a formalisation of a large fragment of our small-step semantics in the proof assistant F* [fst]. At the time of writing, we are formalizing the remaining part, which only consists of straightforward local operations, such as bitwise operators and opcodes to write code to (resp. read code from) the memory. F* is an ML-dialect that is optimized for program verification and allows for performing manual proofs as well as automated proofs leveraging the power of SMT solvers.

Our formalisation strictly follows the small-step semantics as presented in this chapter. The core functionality is implemented by the function `step` that describes how an execution stack evolves within one execution state. To this end it has two possible outcomes: either it performs an execution step and returns the new callstack or – in the case that a final configuration is reached (which is a stack containing only one element that is either a halting or an exception state) – it reports the final state. In order to provide a total function for the step relation, we needed to introduce a third execution outcome that signalizes that a problem occurred due to an inconsistent state. When running the semantics from a valid initial configuration this result, however, should never be produced. For running the semantics, the function `execution` is defined that subsequently performs execution steps using `step` until reaching the final state and reports it.

The current implementation encompasses approximately thousand lines of code. Since F* code can be compiled into OCaml, we validate our semantics against the official EVM test suite [evm]. Our semantics passes 304 out of 624 tests, failing only in those involving any of the missing functionalities.

We make the formalisation in F* publicly available [GMS18b] in order to facilitate the design of static analysis techniques for EVM bytecode as well as their soundness proofs.

6.3.8 Comparison with the Semantics by Luu et al. [LCO$^+$16]

The small-step semantics defined by Luu et al. [LCO$^+$16] encompasses only a variation of a subset of EVM bytecode instructions (called EtherLite) and assumes a heavily simplified execution configuration. The instructions covered span simple stack operations for pushing and popping values, conditional branches, binary operations, instructions for accessing and altering local memory and account storage, as well as the ones for calling, returning and destructing the account. Essential instructions as CREATE and those for accessing the transaction and block information are omitted. The authors represent a configuration as a tuple of a call stack of activation records and the global state. An activation record contains the code to be executed, the program counter, the local memory and the machine stack. The global state is modelled as a mapping from addresses to accounts, with the latter consisting of code, balance and persistent storage.

The overall abstraction contains a conceptual flaw, as not including the global state in the activation records of the call stack does not allow for modelling that, in the case of an exception in the execution of the callee, the global state is rolled back to the one of the caller at the point of calling. In addition, the model cannot be easily extended with further instructions – such as further call instructions or instructions accessing the environment – without major changes in the abstraction as a lot of information, e.g., the one captured in our small-step semantics in the transaction and the execution environment, are missing.

CHAPTER 7

A Static Analysis for the Sound Control Flow Reconstruction of Ethereum smart contracts

7.1 Introduction

Most tools that analyse Ethereum smart contracts at the level of bytecode base their analysis on the contract's control flow graph (CFG). However, the design of the EVM bytecode language does not allow for an easy reconstruction of a contract's control flow since jump destinations are not provided statically, but might be dynamically computed. More precisely, in EVM bytecode jump destinations are read from the stack and hence can be subject to prior computations. Even though the set of potential jump destinations is statically determined (since only program counters with a JUMPDEST instruction constitute valid jump destinations), the concrete destination of a jump instruction might only be dispatched at runtime. The challenge hence lies in statically narrowing down the set of possible jump destinations for each jump instruction (JUMP or JUMPI).

To this end, the state-of-the-art analyser [TDDC+18] deploys a custom algorithm, another popular solution [oB18] uses an external open-source tool [cfg20] for control flow graph reconstruction. When reviewing the algorithms used in [TDDC+18] and [cfg20], we found issues in both approaches as we will discuss in the following. In Figure 7.1 we show a compact example of a smart contract's control flow that is recovered incorrectly by [TDDC+18, cfg20] with no errors reported.

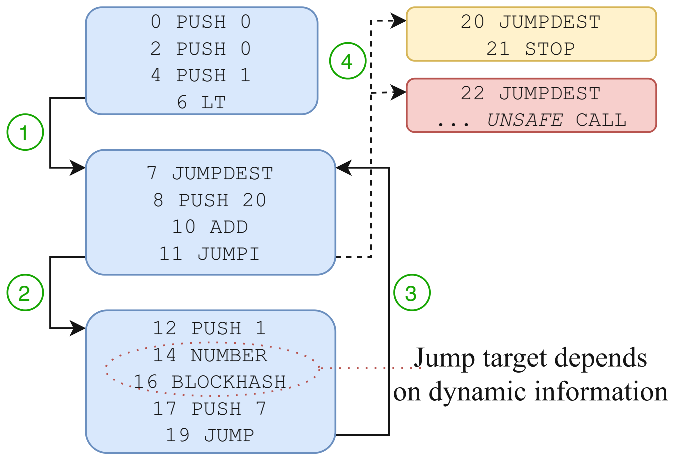


Figure 7.1: Problematic Control Flow Example

Intuitively, the control flow of this contract should not be fully recoverable because one of its jump destinations depends on some blockchain information (the block hash and the block number) which cannot be statically predicted, but will only be fixed once the contract has been published on the blockchain.

The smart contract is structured into five basic blocks. The first block (starting at program counter 0), initializes the local machine stack with two 0 values and continues with the execution of the second block starting at program counter 7 (1). The second block can be entered via a jump (since it starts with a JUMPDEST instruction). It intuitively takes two stack values as arguments, the first one functioning as jump offset and the second being the jump condition: it computes the next jump destination as the sum of 20 and the top stack element and conditionally jumps to this destination based on the second stack value. In the first iteration since both of these values are 0 (and so particularly the condition is 0), no jump is performed, but instead the execution proceeds with block three (starting at program counter 12) with the empty stack (②). This block pushes the current block number and hash to the stack and jumps back to the second block (③). Since at this point the input to the second block are values that are not statically determinable, it needs to be assumed that the jump condition as well as the jump offset could have any value.

It is hence possible during the real execution to jump to arbitrary jump destinations from program counter 10 (④). This includes the block starting at program counter 20 where the execution of the contract is stopped and most importantly the block starting at program counter 22 that executes an unsafe call. Thus, if this jump destination is undiscovered, false correctness results can be produced in a subsequent analysis.

There are two sound approaches for handling the usage of unpredictable information in jump destination reconstruction: conservatively, a smart contract can be rejected by the analysis and hence be considered potentially vulnerable in this case or the analysis could assume that all JUMPDEST instructions of the contract are potentially reachable.

The tools that we reviewed, however, did not follow any of these options, but produced the following results: [cfg20] correctly discovers the basic blocks, but cannot recover jumps to targets 20 and 22 (④). The result of [TDDC$^+$18] is even more surprising: the algorithm does not manage to recover any of the blocks shown in Figure 7.1, but reports as CFG of this contract a single block consisting of a modulo instruction followed by the STOP opcode. Consequently, all analyses that use either of these CFG reconstruction solutions will consider the unsafe call of the example contract to be unreachable and will based on that label the contract as safe. In general, the properties of smart contracts based on the unreachability of unsafe functionality, e.g., the single-entrancy property [GMS18a] ruling out bugs such as those found in the DAO contract [thea], require a sound CFG reconstruction procedure.

While correctness for both tools has never been discussed, flaws in the CFG reconstruction can lead to catastrophic consequences: an unsound reconstruction that erroneously excludes possible jump destinations, can deem parts of the contract code unreachable that carries critical and potentially unsafe functionality.

Our Contributions This chapter discusses a sound CFG reconstruction algorithm for EVM bytecode. In order to leverage the power of modern solvers, we follow the principles discussed in Chapter 2 and Chapter 4, i.e., the Horn-clause based abstraction techniques form the core of our approach that soundly abstracts the concrete semantics of EVM bytecode. In particular:

- We formalize our analysis technique in terms of Horn clauses and prove its soundness with respect to the concrete semantics of EVM bytecode presented in Chapter 6;

- We refine a set of abstractions utilized by the analysis technique, in order to obtain the scalability to large real-world smart contracts without sacrificing the soundness. For instance, in contrast to our abstractions for Dalvik bytecode (Cf. § 2.2 and § 4.2), we conservatively explore both branches for every conditional instruction. This keeps the soundness intact, and at the same time allows for the quick jump destinations recovery.

- We implement our analysis as a practical tool that employs the state-of-the-art solver *Soufflé* [JSS16] to compute the fixedpoint of recursive relations. We discover that for the CFG reconstruction task, *Soufflé* provides better performance than the general-purpose solver *z3* used by HornDroid and fsHornDroid in Chapter 2 and Chapter 4 respectively.

- We provide an extensive performance evaluation of our tool on a recent benchmark obtained from 22493 real-world smart contracts [KGDS18].

7.2 Static Analysis for Control Flow Reconstruction of EVM Bytecode

Our static analysis approach for control flow reconstruction follows the principles of the static analysis techniques for Android applications presented in Chapter 2 and Chapter 4. Considering the small-step concrete semantics for EVM bytecode $\rightarrow$ discussed in Chapter 6, we define a sound analysis for CFG reconstruction. Our analysis is expressed in terms of a Horn-clause based abstraction faithfully over-approximating the semantics of smart contract. Our abstraction is characterised by the abstraction function α converting concrete configurations into abstract configurations, i.e., sets of predicate applications characterised by a signature $\mathcal{S}$. The predicates range over the values from abstract domains. These abstract arguments are equipped with an order $\leq$ that can be canonically lifted to predicates and further to abstract configurations, hence establishing a notion of precision on the latter. Intuitively, α translates a concrete configuration into its most precise abstraction. The *abstract semantics* is specified by a set of Constrained Horn clauses Λ over the predicates from $\mathcal{S}$ and describes how abstract configurations evolve during abstract execution. An abstract execution hence consists of logical derivations from an abstract configuration using Λ. A Horn-clause based abstraction constitutes a sound approximation of small-step semantics $\rightarrow$ if every concrete (multi-step) execution $st \rightarrow^* st'$ can be simulated by an abstract execution: More precisely, from the abstract configuration one can logically derive using Λ an abstract configuration that constitutes an over-approximation of the st' configuration (so is at least as abstract as $\alpha(st')$). Our property of interest (i.e., CFG reconstruction) is local. Consequently, we show a special case of soundness - *local soundness* that does not take into account the evolution of call stacks.

7.2.1 Main Abstractions

Our analysis abstracts from several details of the original small-step semantics. In the following, we overview the main abstractions:

Arithmetic Operations For the performance reasons, the arithmetic operations are over-approximated; each of them results in an arbitrary natural number.

Blockchain Environment The analysis describes the invocation of the contact's bytecode in an arbitrary blockchain environment, hence is not modelling the execution environment as well as the global state.

Gas Modelling The contract gas consumption is not modeled. The gas resource, which is meant to bound the contract execution, is set by the transaction initiator and hence not necessarily known at analysis time. For this reason, our analysis considers arbitrary contract invocations (and hence arbitrary gas limits).

Memory Model The modelling of the EVM memory manipulations is computationally expensive. As the information about jump destinations rarely enters the memory (we never encountered it in our large-scale experiment), we abstract the memory operations, i.e., the operations on the local memory and the storage: writes to the memory are not modeled, and reads from the memory result in an arbitrary natural number.

Halting States & Exceptions As reachable jump instructions are the main focus of our analysis, we omit the modelling of the halting states and the exceptional states.

Callstack Since our abstraction captures all possible execution environments, we can abstract from call stacks. This is sufficient as the required property is local, i.e., they only affect single local execution states and do not relate different execution states on a call stack.

7.2.2 Analysis Definition

In the following, we formally specify our analysis by defining the underlying Horn-clause based abstraction.

Abstract configurations are modeled using predicates as specified by the predicate signature in Figure 7.2: Predicates of the form stack approximate execution states of the contract's at the program counter pc, while push, dup, swap addtop, $\mathsf{onlypop}$, jd, jump, and jumpi represent bytecode operations of the contract. Intuitively, a predicate application of the form $\mathsf{stack}(pc, \widehat{s})$ means that an execution at program counter pc has a local stack with elements as described by the list $\widehat{s}$.

Accordingly, a predicate application $\mathsf{push}(pc, b, v)$ denotes that contract under analysis has a $\mathsf{PUSH}b\ v$ opcode at position pc, and value v is represented by b bytes, predicate

$$
\begin{array}{rcl}
\mathcal{S} \ni p & := & \\
& | & \mathsf{stack} : \mathbb{N} \times \mathcal{L}(\hat{D}) \\
& | & \mathsf{push} : \mathbb{N} \times b \times \mathbb{N} \\
& | & \mathsf{dup} : \mathbb{N} \times n \\
& | & \mathsf{swap} : \mathbb{N} \times n \\
& | & \mathsf{addtop} : \mathbb{N} \times \mathbb{N} \\
& | & \mathsf{onlypop} : \mathbb{N} \times \mathbb{N} \\
& | & \mathsf{jd} : \mathbb{N} \\
& | & \mathsf{jump} : \mathbb{N} \\
& | & \mathsf{jumpi} : \mathbb{N} \\
& | & \mathsf{target} : \mathbb{N} \times \mathbb{N} \\
& & n \in \{1, \ldots, 16\} \\
& & b \in \{1, \ldots, 32\} \\
\hat{D} & := & \mathbb{N} \cup \{\top\}
\end{array}
$$

Figure 7.2: Definition of the predicate signature $\mathcal{S}$ and the abstract domain $\hat{D}$

applications $\mathsf{dup}(pc, n)$ and $\mathsf{swap}(pc, n)$ indicate that that the contact has correspondingly a DUPn/SWAPn instruction at position pc.

A predicate application $\mathsf{addtop}(pc, n)$ stands for having an operation among

ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, PC, MSIZE, GAS, ISZERO, NOT, BALANCE, CALLDATALOAD, ADD, MUL, SUB, DIV, SDIV, MOD, SMOD, EXP, SIGNEXTEND, LT, GT, SLT, SGT, EQ, AND, OR, XOR, BYTE, SHA3, ADDMOD, MULMOD, CREATE, DELEGATECALL, STATICCALL, CALL, CALLCODE

in the contract's code at position pc. All those operations take a certain amount of arguments from the stack and produce a result on the stack. The analysis abstracts these operations by a simplified stack modification behaviour, in particular, by eliminating n elements starting from the top of the stack and adding $\top$ element on top of it. A predicate $\mathsf{onlypop}(pc, n)$ is similar to $\mathsf{addtop}(pc, n)$ with the only difference that it does not add $\top$ on top of the stack. Having $\mathsf{onlypop}(pc, n)$ indicates that the code has an opcode among

MSTORE, MSTORE8, SSTORE, CALLDATACOPY, CODECOPY, EXTCODECOPY, LOG0, LOG1, LOG2, LOG3, LOG4

at position pc.

We mark location pc as a potential jump destination via predicate application $\mathsf{jd}(pc)$, and predicate applications $\mathsf{jump}(pc)$ and $\mathsf{jumpi}(pc)$ express that c has respectively a non-conditional and a conditional jump at position pc.

Moreover, the predicate $\mathsf{target}(pc, t)$ represents the fact that a jump instruction at program counter pc has jump destination t. We use this predicate to capture the *destination unreachability* property described later in § 7.2.5 since target relates every jump instruction with its possible jump destinations.

The argument domains of the predicates are composed of natural numbers, and the abstract domain $\hat{D}$ that enriches $\mathbb{N}$ with the join element $\top$ representing an arbitrary natural number.

The definition of the execution state abstraction function α that maps EVM execution states to abstract configurations is given in Figure 7.3. As the analysis aims at verifying the destination unreachability property for a specific contract's bytecode, the abstraction function α_s translates the bytecode instructions (i.e., α_{pc} introduces the push, dup, swap, addtop, onlypop, jd, jump, and jumpi predicates representing the relevant features of the contract's instructions), and the local execution states of the bytecode (i.e., stack).

The abstract semantics of contracts is a set of Horn clauses Λ over-approximating the semantics of the EVM bytecode instructions. These Horn clauses are depicted in Figure 7.4 and will be discussed in the following.

Push Push operations are represented by the general rule (P) for a successful push operation which is the same for all push opcodes: program counter pc is advanced by the number of pushed bytes b and the value v is added on top of stack $\widehat{s}$.

Dup, Swap The (D) rule states how an operation DUPn (which duplicates the $n-1$th stack element) is captured in the analysis: program counter pc is incremented by one, and the value $\widehat{v_{n-1}}$ at the position $n-1$ (counted from the top of the stack) is added to the top of the stack. The rule for SWAP instructions is similar to the one for DUP with the value $\widehat{v_n}$ at position n swapped with the value $\widehat{v}$ from the top of the stack.

Addtop, Onlypop The abstract semantics for ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, PC, MSIZE, GAS, ISZERO, NOT, BALANCE, CALLDATALOAD, ADD, MUL, SUB, DIV, SDIV, MOD, SMOD, EXP, SIGNEXTEND, LT, GT, SLT, SGT, EQ, AND, OR, XOR, BYTE, SHA3, ADDMOD, MULMOD, CREATE, DELEGATECALL, STATICCALL, CALL, and CALLCODE instructions is summarized by a Horn clause (A) describing the removal of several elements from the stack and the pushing of $\top$ on the top of the stack. A prerequisite for a successful execution is the existence of a sufficient amount of arguments (n) on the machine stack. In this case, the top stack values are discarded, and the stack at the next program counter (modeled by the predicate stack$(pc+1)$) is updated with $\top$. The rule that describes the abstract semantics for the successful execution of POP, MSTORE, MSTORE8, SSTORE, LOG0, CALLDATACOPY, CODECOPY, LOG1, EXTCODECOPY, LOG2, LOG3, and LOG4 instructions (O) is similar to (A), that is (O) also pops n elements from the stack but does not add anything to the stack.

$$\alpha_s(\mathcal{S}) := \begin{cases} \{\mathsf{stack}(pc,\ \widehat{s}\ )\} \cup \alpha_{pc}(0, \iota.\mathsf{code}) & \mathcal{S} = ((gas, pc, m, i, s), \iota, \sigma, \eta) \\ \emptyset & otherwise \end{cases}$$

$$\alpha_{pc}(pc, c) := \begin{cases} \{\mathsf{push}(pc, \mathsf{b}, \mathsf{val})\} \cup \alpha_{pc}(pc + \mathsf{b} + 1, c) & c[pc] = \mathsf{PUSHb} \wedge c[pc+1, pc+\mathsf{b}] = \mathsf{val} \\ \{\mathsf{dup}(pc, \mathsf{n})\} \cup \alpha_{pc}(pc + 1, c) & c[pc] = \mathsf{DUPn} \\ \{\mathsf{swap}(pc, \mathsf{n})\} \cup \alpha_{pc}(pc + 1, c) & c[pc] = \mathsf{SWAPn} \\ \{\mathsf{jd}(pc)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] = \mathsf{JUMPDEST} \\ \{\mathsf{jump}(pc)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] = \mathsf{JUMP} \\ \{\mathsf{jumpi}(pc)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] = \mathsf{JUMPI} \\ \{\mathsf{addtop}(pc, 0)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] \in \{\mathsf{ADDRESS}, \mathsf{ORIGIN}, \mathsf{CALLER}, \\ & \mathsf{CALLVALUE}, \mathsf{CALLDATASIZE}, \mathsf{CODESIZE}, \\ & \mathsf{GASPRICE}, \mathsf{COINBASE}, \mathsf{TIMESTAMP}, \\ & \mathsf{NUMBER}, \mathsf{DIFFICULTY}, \mathsf{GASLIMIT}, \\ & \mathsf{PC}, \mathsf{MSIZE}, \mathsf{GAS}\} \\ \{\mathsf{addtop}(pc, 1)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] \in \{\mathsf{ISZERO}, \mathsf{NOT}, \mathsf{BALANCE}, \\ & \mathsf{CALLDATALOAD}, \mathsf{EXTCODESIZE}, \\ & \mathsf{BLOCKHASH}, \mathsf{MLOAD}, \mathsf{SLOAD}\} \\ \{\mathsf{addtop}(pc, 2)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] \in \{\mathsf{ADD}, \mathsf{MUL}, \mathsf{SUB}, \mathsf{DIV}, \mathsf{SDIV}, \mathsf{MOD}, \\ & \mathsf{SMOD}, \mathsf{EXP}, \mathsf{SIGNEXTEND}, \mathsf{LT}, \mathsf{GT}, \mathsf{SLT}, \\ & \mathsf{SGT}, \mathsf{EQ}, \mathsf{AND}, \mathsf{OR}, \mathsf{XOR}, \mathsf{BYTE}, \mathsf{SHA3}\} \\ \{\mathsf{addtop}(pc, 3)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] \in \{\mathsf{ADDMOD}, \mathsf{MULMOD}, \mathsf{CREATE}\} \\ \{\mathsf{addtop}(pc, 6)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] \in \{\mathsf{DELEGATECALL}, \mathsf{STATICCALL}\} \\ \{\mathsf{addtop}(pc, 7)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] \in \{\mathsf{CALL}, \mathsf{CALLCODE}\} \\ \{\mathsf{onlypop}(pc, 1)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] = \mathsf{POP} \\ \{\mathsf{onlypop}(pc, 2)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] \in \{\mathsf{MSTORE}, \mathsf{MSTORE8}, \mathsf{SSTORE}, \\ & \mathsf{LOG0}\} \\ \{\mathsf{onlypop}(pc, 3)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] \in \{\mathsf{CALLDATACOPY}, \mathsf{CODECOPY}, \\ & \mathsf{LOG1}\} \\ \{\mathsf{onlypop}(pc, 4)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] \in \{\mathsf{EXTCODECOPY}, \mathsf{LOG2}\} \\ \{\mathsf{onlypop}(pc, 5)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] = \mathsf{LOG3} \\ \{\mathsf{onlypop}(pc, 6)\} \cup \alpha_{pc}(pc + 1, c) & c[pc] = \mathsf{LOG4} \\ \emptyset & (pc < 0 \vee pc \geq |c| \vee c[pc] \notin inst) \end{cases}$$

Figure 7.3: Definition of the configuration abstraction function, where *inst* is a set of all valid EVM instructions

$$
\begin{array}{ll}
\Lambda := & \\
\quad \{\mathsf{push}(pc, b, v) \wedge \mathsf{stack}(pc, \widehat{s}) \implies \mathsf{stack}(pc + b + 1, v{::}\, \widehat{s}), & \text{(P)} \\
\quad \mathsf{dup}(pc, n) \wedge \mathsf{stack}(pc, \widehat{s_1} \mathbin{+\!\!+} (\widehat{v_{n-1}} :: \widehat{s_2})) \wedge |\widehat{s_1}| = n - 1 & \\
\qquad\qquad \implies \mathsf{stack}(pc + 1, \widehat{v_{n-1}} :: (\widehat{s_1} \mathbin{+\!\!+} (\widehat{v_{n-1}} :: \widehat{s_2}))), & \text{(D)} \\
\quad \mathsf{swap}(pc, n) \wedge \mathsf{stack}(pc, \widehat{v} :: (\widehat{s_1} \mathbin{+\!\!+} (\widehat{v_n} :: \widehat{s_2}))) \wedge |\widehat{s_1}| = n & \\
\qquad\qquad\quad \implies \mathsf{stack}(pc + 1, \widehat{v_n} :: (\widehat{s_1} \mathbin{+\!\!+} (\widehat{v} :: \widehat{s_2}))), & \text{(S)} \\
\quad \mathsf{addtop}(pc, n) \wedge \mathsf{stack}(pc, \widehat{s_1} \mathbin{+\!\!+} \widehat{s_2}) \wedge |\widehat{s_1}| = n \implies \mathsf{stack}(pc + 1, \top{::}\, \widehat{s_2}), & \text{(A)} \\
\quad \mathsf{onlypop}(pc, n) \wedge \mathsf{stack}(pc, \widehat{s_1} \mathbin{+\!\!+} \widehat{s_2}) \wedge |\widehat{s_1}| = n \implies \mathsf{stack}(pc + 1, \widehat{s_2}), & \text{(O)} \\
\quad \mathsf{jd}(pc) \wedge \mathsf{stack}(pc, \widehat{s}) \implies \mathsf{stack}(pc + 1, \widehat{s}), & \text{(JD)} \\
\quad \mathsf{jump}(pc) \wedge \mathsf{stack}(pc, t{::}\, \widehat{s}) \wedge \mathsf{jd}(t) \implies \mathsf{stack}(t, \widehat{s}), & \text{(J1)} \\
\quad \mathsf{jump}(pc) \wedge \mathsf{stack}(pc, \top{::}\, \widehat{s}) \wedge \mathsf{jd}(pc') \implies \mathsf{stack}(pc', \widehat{s}), & \text{(J2)} \\
\quad \mathsf{jumpi}(pc) \wedge \mathsf{stack}(pc, t{::}br{::}\, \widehat{s}) \wedge \mathsf{jd}(t) \implies \mathsf{stack}(t, \widehat{s}), & \text{(JI1)} \\
\quad \mathsf{jumpi}(pc) \wedge \mathsf{stack}(pc, \top{::}br{::}\, \widehat{s}) \wedge \mathsf{jd}(pc') \implies \mathsf{stack}(pc', \widehat{s}), & \text{(JI2)} \\
\quad \mathsf{jumpi}(pc) \wedge \mathsf{stack}(pc, t{::}br{::}\, \widehat{s}) \implies \mathsf{stack}(pc + 1, \widehat{s}), & \text{(JI3)} \\
\quad \mathsf{jump}(pc) \wedge \mathsf{stack}(pc, t{::}\, \widehat{s}) \wedge \mathsf{jd}(t) \implies \mathsf{target}(pc, t), & \text{(T1)} \\
\quad \mathsf{jump}(pc) \wedge \mathsf{stack}(pc, \top{::}\, \widehat{s}) \wedge \mathsf{jd}(pc') \implies \mathsf{target}(pc, pc'), & \text{(T2)} \\
\quad \mathsf{jumpi}(pc) \wedge \mathsf{stack}(pc, t{::}\, \widehat{s}) \wedge \mathsf{jd}(t) \implies \mathsf{target}(pc, t) & \text{(T3)} \\
\quad \mathsf{jumpi}(pc) \wedge \mathsf{stack}(pc, \top{::}\, \widehat{s}) \wedge \mathsf{jd}(pc') \implies \mathsf{target}(pc, pc')\} & \text{(T4)}
\end{array}
$$

Figure 7.4: Abstract semantics rules Λ

Jumps On the level of EVM, bytecode jump operations enable complex control flows while the rest of operations pass the control directly to the next operation, advancing the program counter by one (or by a number of bytes pushed in case of push operations).

Rule (JD) describes that the execution of this opcode will not have any changes in the state when encountering the JUMPDEST opcode since the sole purpose of these opcodes is to mark their locations in the code as jump destinations.

Since stack values range over the abstract domain $\hat{D}$ defined in Figure 7.2 one needs to distinguish three cases when a jump instruction is encountered, namely the value t on top of the stack can be a valid jump destination, an invalid jump destination, or it can be abstracted as $\top$. The first case is captured by rule (J1): if t is not abstracted as $\top$ and t is a valid jump destination (i.e., $\mathsf{jd}(t)$ holds) the successful non-conditional jump to the destination t is performed, that is the program stack remains intact, but the program counter is changed to t. The second case where t is an invalid jump destination produces an exception, they are not modelled by our analysis as irrelevant to the property in question. Rule (J2) describes the last case where t is abstracted as $\top$. As jd is populated by the valid program counters before the analysis, the jump is performed to all possible locations marked as the jump destinations in the code by the JUMPDEST opcodes, that is pc' values are determined by the range of the function D that determines contract's potential jump destinations. The definition of the function D is given in § 6.3.4.

Rules (JI1), (JI2) and (JI3) model the abstraction of the conditional jumps: the analysis ignores the value of the branching condition br and conservatively explores all possible outcomes of the branch condition evaluation: (JI1) describes the successful jump performed if its branching condition is not zero and $t \neq \top$, (JI2) is similar to (JI1) but here $t = \top$ (hence the jump is performed to all possible destinations pc'), and (JI3) describes advancing the program counter by one with stack unchanged in the case where the branching condition is zero.

7.2.3 Scope of the Analysis

Before presenting the soundness result, we discuss the scope of the analysis. The analysis targets local properties (e.g., CFG reconstruction), hence it reasons only about evolutions of the local execution states. Consequently the execution environments are persistent during these evolutions which we formally state in the following lemma:

Lemma 29 (Environment preservation)**.** *Let Γ be a transaction environment and let S and S' be arbitrary callstacks. Then for all local execution states $\mathcal{S}$ and $\mathcal{S}'$ it holds that*

$$\Gamma \vDash \mathcal{S}{::}S \rightarrow^{*} S' + \mathcal{S}'{::}S \implies \mathcal{S}.\iota = \mathcal{S}'.\iota$$

Proof. The proof is by induction on the number of small-steps and a case distinction on the structure of the callstack using Lemma 32. □

The locality mentioned above also allows us to avoid heavy over-approximations of the behavior of all contracts that the contract under analysis might interact with.

We now briefly illustrate the key design choices behind our abstraction, which we carefully crafted to find the sweet spot between accuracy and practicality. The analysis is value-sensitive for stack values in that concrete stack values are tracked until they get abstracted due to the influence of unknown components, and value-insensitive for memory, and storage values. For local computations, the analysis is partly flow-sensitive (considering the order of instructions, but merging abstract configurations at the same program counters) and path-insensitive, i.e., insensitive to branch conditions as described in the definition of JUMPI Horn clause. On the level of contract calls, a context insensitivity is given as the calls are not tracked.

7.2.4 Soundness Result

As our property of interest is local we prove that the defined Horn-clause based abstraction soundly over-approximates the local state evolution specified by the small-step semantics presented in Chapter 6. Formally, this property is stated as follows:

Theorem 4 (Local Soundness)**.** *Let Γ be a transaction environment and let S and S' be arbitrary callstacks. Then for all local execution states $\mathcal{S}$ and $\mathcal{S}'$ it holds that*

$$\begin{aligned}\Gamma \vDash \mathcal{S}{::}S \ \rightarrow^* \ S' +\!\!+ \ \mathcal{S}'{::}S \implies \forall \Delta_I.\ \alpha_s(\mathcal{S}) \leq \Delta_I \\ \implies \exists \Delta.\ \Delta_I, \Lambda \vdash \Delta \quad \wedge \ \alpha_s(\mathcal{S}') \leq \Delta\end{aligned}$$

For better readability hereafter we use a coma instead of the $\cup$ sign in the statements of the form similar to the statement $\Delta_I, \Lambda \vdash \Delta$ above.

The locality is captured in Theorem 4 by having the same S before and after performing the execution steps $\rightarrow^*$. Intuitively, it implies that the execution states evolve within the same call. For the proof of Theorem 4 we refer the reader to § 8.3.

7.2.5 Control Flow Reconstruction as a Reachability Property

In this subsection we show how our reachability analysis can be used to reconstruct the control flow of Ethereum smart contracts.

All possible jump destinations are marked in EVM bytecode with JUMPDEST instructions. Hence, for every JUMP and JUMPI instruction, a full collection of possible jump destinations is known and it is sound to consider them all when reconstructing the control flow. However, this makes even simple programs intractable for the analysis as it introduces complex control flow structures even for simplistic programs. Hence, a CFG reconstruction analysis needs to eliminate *unreachable destinations* from the consideration. With our static analysis technique, we can statically check whether jump destinations can be removed from the set of possible jump destinations resulting in tractable control flow graphs. By the soundness of the analysis, if the jump instruction cannot be reached with a destination value on the top of the stack, the destination becomes unreachable for the jump instruction and can be removed. Formally, we can characterise this property as the following reachability property:

Definition 31 (Destination unreachability)**.** Let $\mathcal{S} = (\mu, \iota, \sigma, \eta)$ be a regular execution state such that $\mu = (g, 0, \lambda x. 0, 0, \epsilon)$ for some $g \in \mathbb{N}$. Let Γ be an arbitrary transaction environment, $\mathcal{S}'$ be some execution state, and S and S' be arbitrary callstacks. Then the unreachable destination check for destination t is defined as follows:

$$\begin{aligned}&DU(c, pc, t) := \\ &\quad \neg\exists \mathcal{S}, S, S'.\, \Gamma \vDash (\mu, \iota, \sigma, \eta){::}S \ \rightarrow^* \ S' +\!\!+ \ \mathcal{S}{::}S \\ &\qquad\qquad \wedge c = \mathcal{S}.\iota.code \ \wedge \ pc = \mathcal{S}.\mu.pc \\ &\qquad\qquad \wedge (c\,[pc] = \mathsf{JUMP} \ \vee \ c\,[pc] = \mathsf{JUMPI}) \\ &\qquad\qquad \wedge \ \mathcal{S}.\mu.\mathsf{s} = t{::}s \ \wedge \ t \in D\,(c)\end{aligned}$$

Intuitively this property says that during an execution of contract code c it should never be possible to reach a JUMP/JUMPI instruction at position pc with jump destination t on the top of the stack.

In Figure 7.4 the rules (T1), (T2), (T3), and (T4) are relevant for capturing the *DU* property in our analysis. Specifically rule (T1) saves the value of the jump destination t (if it is not over-approximated as $\top$) in the relation target for an unconditional jump instruction at pc. In the case when the jump destination is over-approximated as $\top$ the rule (T2) is applied saving all possible jump destinations pc' (the range of the function D) in the relation target for an unconditional jump instruction at pc. The rules (T3) and (T4) do the same as correspondingly (T1) and (T2) for a conditional jump instruction. In other words, whenever the code under analysis can reach a state when the jump (corresponding to JUMP or JUMPI instruction at program counter pc) is successfully performed to the destination t, our analysis guarantees the destination derivability, i.e., it can derive the $\mathsf{target}(pc, t)$ relation using the rules (T1), (T2) (T3), and (T4) $\in \Lambda$. The destination derivability is detailed in Lemma 30.

Lemma 30 (Destination derivability)**.** *Let* $\mathcal{S} = (\mu, \iota, \sigma, \eta)$ *for some* $g \in \mathbb{N}$ *be a regular execution state such that* $c = \mathcal{S}.\iota.code$, $pc = \mathcal{S}.\mu.pc$, *and* $(c\,[pc] = \mathsf{JUMP} \vee c\,[pc] = \mathsf{JUMPI})$*. Let* $\mathcal{S}.\mu.\mathsf{s} = t :: s$ *such that* $t \in D\,(c)$*. Then the destination derivability holds for the destination* t *when:*

$$\forall \Delta \geq \alpha_s(\mathcal{S}) \implies \Lambda, \Delta \vdash \mathsf{target}(pc, t)$$

Proof. Let $(c\,[pc] = \mathsf{JUMP}$ (a) or $c\,[pc] = \mathsf{JUMPI})$ (b). Now we proceed by case distinction.

(a) In this case $(c\,[pc] = \mathsf{JUMP}$. By definition of α_s presented in Figure 7.3 and $\Delta \geq \alpha_s(\mathcal{S})$ we know that $\{\mathsf{jump}(pc)\} \cup \{\mathsf{stack}(pc, t' :: s)\} \cup \{\mathsf{jd}(t)\} \in \Delta$. Now either $t' = t$ or $t' = \top$. In the case where $t' = t$, since jd only takes arguments from $\mathbb{N}$ we can apply the rule (T1) $\in \Lambda$ and derive $\mathsf{target}(pc, t)$, in the case where $t' = \top$ we can apply the rule (T2) $\in \Lambda$ and derive $\mathsf{target}(pc, t)$. Hence, $\Lambda, \Delta \vdash \mathsf{target}(pc, t)$ which concludes the proof.

(b) In this case $(c\,[pc] = \mathsf{JUMPI}$ By definition of α_s presented in Figure 7.3 and $\Delta \geq \alpha_s(\mathcal{S})$ we know that $\{\mathsf{jumpi}(pc)\} \cup \{\mathsf{stack}(pc, t' :: s)\} \cup \{\mathsf{jd}(t)\} \in \Delta$. Now either $t' = t$ or $t' = \top$. In the case where $t' = t$, since jd only takes arguments from $\mathbb{N}$ we can apply the rule (T3) $\in \Lambda$ and derive $\mathsf{target}(pc, t)$, in the case where $t' = \top$ we can apply the rule (T4) $\in \Lambda$ and derive $\mathsf{target}(pc, t)$. Hence, $\Lambda, \Delta \vdash \mathsf{target}(pc, t)$ which concludes the proof.

□

Due to the soundness property of the analysis presented in Theorem 4 and proven in § 8.3, and the destination derivability captured in Lemma 30 and proven in this section, if the destination value for a jump instruction can be derived with the abstract semantics, it is saved in target, and due to the soundness property, the abstract semantics can derive all possible destinations for every jump instruction. In other words, if the $DU(c, pc, t)$ property cannot be shown for the jump destination t (i.e., t is a reachable destination) then the analysis must derive $\mathsf{target}(pc, t)$. We formally state the latter in Theorem 5.

Theorem 5 (Reachable destinations)**.** *Let c be a code under analysis. Then for all jump destinations t and program counters pc it holds that*

$$\Lambda, \{\mathsf{stack}(0, \epsilon)\}, \alpha_{pc}(0, c) \nvdash \mathsf{target}(pc, t) \implies DU(c, pc, t)$$

Proof. We proceed by contraposition. Assume $\neg DU(c, pc, t)$ (1). We show $\Lambda, \alpha_{pc}(0, c) \vdash \mathsf{target}(pc, t)$. By (1) and by Definition 31 we know that there is a regular execution state $\mathcal{S}$, an arbitrary transaction environment Γ, some execution state $\mathcal{S}'$, and arbitrary callstacks S and S' such that $\mathcal{S} = (\mu, \iota, \sigma, \eta)$, $\mu = (g, 0, \lambda x. 0, 0, \epsilon)$ for some $g \in \mathbb{N}$, and

$$\begin{aligned}
&\Gamma \vDash \mathcal{S}{::}S \ \to^* \ S' + \mathcal{S}'{::}S \\
&\wedge c = \mathcal{S}'.\iota.code \ \wedge \ pc = \mathcal{S}'.\mu.pc \\
&\wedge (c\,[pc] = \mathsf{JUMP} \ \vee \ c\,[pc] = \mathsf{JUMPI}) \\
&\wedge \ \mathcal{S}'.\mu.\mathsf{s} = t{::}s \ \wedge \ t \in D\,(c).
\end{aligned}$$

By Theorem 4 we know that $\forall \Delta_I.\ \alpha_s(\mathcal{S}) \leq \Delta_I \implies \exists \Delta.\ \Delta_I, \Lambda \vdash \Delta \ \wedge \ \alpha_s(\mathcal{S}') \leq \Delta$. In particular the latter holds for $\alpha_s(\mathcal{S}) = \Delta_I$. By Lemma 29 we know that $\mathcal{S}.\iota = \mathcal{S}'.\iota$, and consequently also $\mathcal{S}.\iota.code = \mathcal{S}'.\iota.code = c$. Hence, we know that $\alpha_s(\mathcal{S}) = \{\mathsf{stack}(0, \epsilon)\} \cup \alpha_{pc}(0, c)$ consequently then also $\{\mathsf{stack}(0, \epsilon)\}, \alpha_{pc}(0, c), \Lambda \vdash \Delta$. By Lemma 30 we know that $\Lambda, \Delta \vdash \mathsf{target}(pc, t)$. This concludes the proof. □

Intuitively Theorem 5 states that the destination unreachability property is guaranteed for all jump destinations that are not considered by our analysis.

7.3 Implementation

We employ the general-purpose solver *z3* in the analysis of Android applications presented in Chapter 2 and Chapter 4. However, since the problem of control flow reconstruction falls into the fragment supported by modern datalog solvers, we found the high-performance datalog engine *Soufflé* more performant than using the general-purpose solver *z3* in this context. *z3* also implements a standard datalog engine which is restricted to work with predicates over finite domains. This constraint is used to ensure that Horn clauses do not leave the classical datalog-solvable fragment. However, *Soufflé* overcomes this restriction in favor of a more liberal characterisation of the solvable fragment.

Also *Soufflé* allows for deriving the full target relation instead of showing the destination unreachability property for each pair of a jump instruction and a jump destination that is necessary when using *z3*. In other words, when using *z3* the number of queries we need to invoke equals the product of the number of jump instructions and the number of possible jump destinations.

The general approach of using *Soufflé* for CFG reconstruction is not new and has been employed by a decompiler Gigahorse [GBSS19]. However, the Gigahorse's approach is not focusing on soundness, and consequentiality the soundness property of this framework is not established.

The outline of our implementation is presented in Figure 7.5.

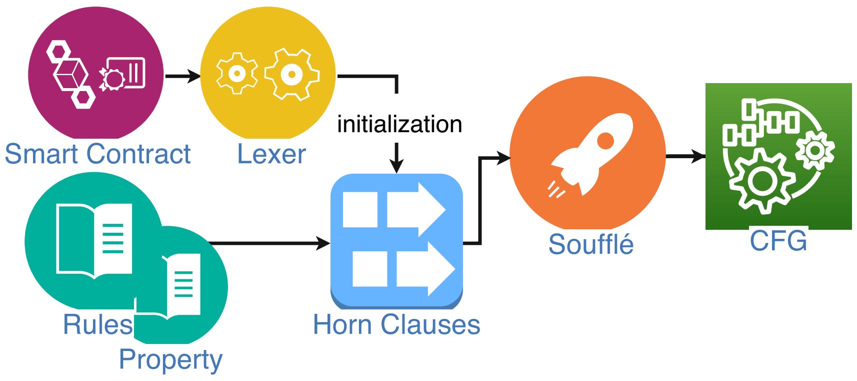


Figure 7.5: Implementation Outline

Our implementation uses the initialization information about the code under analysis and the general rules. The initialization information is provided by a lexer which takes the analysed program and encodes it into the set of facts about the bytecode's operations, their examples are depicted in the part specifying α_{pc} in Figure 7.3. The general rules consist of the rules for each instruction and the Horn clauses relevant to the control flow reconstruction property. The general rules for each bytecode instruction represent the state transformations for each type of the opcode, in Figure 7.4 they are (P), (D), (S), (A), (O), (J1), (J2), (JD), (JI1), (JI2), and (JI3). The Horn clauses relevant for the CFG reconstruction property are (T1), (T2), (T3) and (T4).

As the general rules do not depend on the particular bytecode, the solution is fully automated. We append to each initialization information provided by the lexer the general Horn clause rules, and then invoke the *Soufflé* solver. In order to perform the resolution *Soufflé* requires that: Firstly, all Horn clauses constitute valid datalog rules[soua], in particular, the initial state is well-grounded, that is following the concrete semantics specification, we use the empty stack configuration $\mathsf{stack}(0, \epsilon)$ as our initial configuration; and Secondly, the predicates relevant for the property are marked as output, in our case we mark the target predicate as the output predicate. During the resolution *Soufflé* computes the least fixedpoint of the output relation. This allows for computation of the mapping between all jump instructions and their destinations. However, in the case where we derive $\top$ as one of the possible jump destinations, our current implementation will conservatively reject the smart contract, e.g.,the smart contract from Figure 7.1. We allow this detour from the analysis specification as it is sound, and permits a clear implementation within the fully supported *Soufflé* fragment, i.e., without custom functors[soub].

7.4 Evaluation

We evaluated the state-of-the-art CFG reconstruction solutions [TDDC$^+$18, BJK$^+$18], and our implementation against the recent benchmark presented in [KGDS18]. We excluded from the evaluation two popular approaches: Gigahorse [GBSS19] (mentioned at the beginning of § 7.3) as it is not publicly available, and [cfg20] (discussed in § 7.1) as it does not report the errors relevant to the CFG reconstruction.

The authors of the benchmark [KGDS18] extracted 22493 real-world contracts from the Ethereum blockchain over a period of three months and (after deduplication) made available a list of 1524 contract addresses, but unfortunately, no source or bytecode. Of these 1524 contracts, 21 have a name that does not resemble an address, 397 have a truncated address, the remaining addresses still contain duplicates. After removing them, we arrive at 1033 contracts. For 286 of the contract addresses we were not able to obtain the code: 53 have been self-destructed, 232 have no recorded transaction that created them, and 1 is an address with no code deployed. This leaves us with 747 contracts. After removing contracts with the same bytecode we arrive at 720 distinct contracts.

	Percent of reconstructed CFGs	*Soundness*
[BJK$^+$18]	90% (73 failures for 720 contracts)	no
[TDDC$^+$18]	95% (34 failures for 720 contracts)	no
This work	98% (14 failures for 720 contracts)	yes

Table 7.1: Percent of successfully reconstructed CFGs and Soundness guarantees in analyses for EVM CFG reconstruction

All of the experiments were conducted on the machine with 8 Cores at 2.6 GHz and 16 GiB of RAM. Each experiment had 30 minutes timeout. The results are depicted in Table 7.1. Within the timeout, Vandal [BJK$^+$18] finished CFG reconstruction for 90% of all contracts: for 73 contracts the tool reported at least one unresolved jump destination; [TDDC$^+$18] produced results for 95% of all contracts: for 34[1] contracts the tool reported decompilation problems; and our solution successfully reconstructed the control flow graph for 98% of smart contracts (all but 14 smart contracts for which the analysis did not terminate within the timeout) while being the first sound CFG reconstruction approach for the EVM bytecode.

Moreover, using our control flow reconstruction results, we conducted a further analysis of various reachability properties and achieved excellent performance [SGSM20]. Therefore, we conclude that our CFG reconstruction can be used by practical analysis tools while being the first one to be proven sound.

[1]However, this number is questionable: as we discuss in § 7.1 [TDDC$^+$18] together with [cfg20] do not produce any error for the obviously unsuccessful CFG reconstruction.

CHAPTER 8

Proofs of Chapter 7

Chapter Outline: In § 8.1 we describe Horn-clause based abstractions, in Section 8.2 we characterise the analysis definitions; in Section 8.3 we give the soundness proof.

8.1 Horn-clause based Abstraction

In this section, we more formally characterize the aim and scope of this work. Generally, we focus on the reachability analysis of smart contracts with the small-step semantics as the one described in Chapter 2, which we over-approximate by an abstract program semantics based on Horn clauses similar to the analyses described in Chapter 2 and Chapter 4. More formally, we will assume a program's small-step semantics to be a binary relation S_s over program configurations $c \in \mathcal{C}$. A Horn-clause based abstraction for such a small-step semantics S_s is then fully specified by a tuple $(\mathcal{D}, \mathcal{S}, \alpha, \Lambda)$ where $\mathcal{S}$ defines the signature of predicates with arguments ranging over (partially) ordered subsets of $\mathcal{D}$. For a given a predicate signature $\mathcal{S}$, an abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$ maps concrete program configurations $c \in \mathcal{C}$ to abstract program configurations $\Delta \in \mathcal{A}$ consisting of instances of predicates in $\mathcal{S}$.

Formally, a predicate signature $\mathcal{S} \in \mathcal{N} \nrightarrow \prod(\mathcal{P}(\mathcal{D}) \times (\mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{D})))$ is a partial function from predicate names $\mathcal{N}$ to their argument types (formally written as a product over the subsets of some abstract superdomain $\mathcal{D}$, equipped with a corresponding order). We require for all $n \in \mathcal{N}$ that $(D, \leq) \in \mathcal{S}(n)$ that $(D, \leq)$ forms a partially ordered set. Correspondingly, the set of abstract configurations $\mathcal{A}_{\mathcal{S}}$ over $\mathcal{S}$ can be defined as $\mathcal{P}(\{n(\vec{v}) \mid n \in \mathcal{N} \ \wedge \ \forall i \in \{1, \dots, |\mathcal{S}(n)|\}.\, \pi_i(\mathcal{S}(n)) = (D, \leq) \implies \pi_i(\vec{v}) \in D\})$ where $\pi_i(\cdot)$ denotes the usual projection operator. The abstraction of a small-step semantics is then a set of constrained Horn clauses $\Lambda \subseteq \mathcal{H}(\mathcal{S})$ that approximates the small-step execution rules.

A constrained Horn clause is a first order formula of the form

$$\forall X.\ \Phi, P \Rightarrow c$$

Where $X \subseteq \mathit{Vars} \times \mathcal{P}(\mathcal{D})$ is a (functional) set of typed variables, and Φ is a set of quantifier free constraints over the variables in X. Conclusions c are predicate applications $n(\vec{z}) \in P_X := \{n(\vec{x}) \mid |\vec{x}| = |\mathcal{S}(n)| \ \wedge\ \forall i \in \{1, \dots, |\vec{x}|\}. \pi_i(\vec{x}) = x \wedge \pi_i(\mathcal{S}(n)) = (D, \leq) \implies (x, D) \in X\}$ over variables in X that respect the variable type. Correspondingly, the premises $P \subseteq P_X$, are a set of predicate applications over variables in X.

We lift the suborders of $\mathcal{S}$ to an order on abstract configurations $\Delta_1, \Delta_2 \in \mathcal{A}_{\mathcal{S}}$ as follows:

$$\begin{aligned} n_1(\vec{t_1}) \leq_p n_2(\vec{t_2}) &:= n_1 = n_2 \\ &\quad \wedge\ \forall i \in \{1, \dots, |\vec{t_1}|\}.\ \pi_i(\vec{t_1}) \leq_{n_1,i} \pi_i(\vec{t_2}) \\ \text{given } \pi_i(\mathcal{S}(n)) = (D_{n,i}, \leq_{n,i}) & \\ \Delta_1 \leq \Delta_2 &:= \forall p_1 \in \Delta_1.\ \exists p_2 \in \Delta_2.\ p_1 \leq_p p_2 \end{aligned}$$

Finally, we adopt from [SGSM20] the notion of soundness for a Horn-clause based abstraction.

Definition 32. A Horn-clause based abstraction $(\mathcal{D}, \mathcal{S}, \alpha, \Lambda)$ *soundly approximates* a small-step semantics S_s if

$$\forall (c, c') \in S_s^*.\ \forall \Delta.\ \alpha(c) \leq \Delta \Rightarrow \exists \Delta'.\ \Delta, \Lambda \vdash \Delta' \ \ \wedge\ \alpha(c') \leq \Delta'$$

This statement requires that, whenever a concrete configuration c' is reachable from configuration c (meaning that (c, c') is contained in the reflexive and transitive closure of S_s, denoted as S_s^*), it shall hold that from all abstractions Δ of c, the Horn clause abstraction allows us to logically derive ($\vdash$) a valid abstraction Δ' of c'. Note that α intuitively yields the most concrete abstraction of a configuration, hence to make the property hold for all possible abstractions of a configuration, we strengthen the property to hold for all abstractions that are more abstract than $\alpha(c)$. The soundness theorem implies that whenever we can show that from some abstraction Δ of a configuration c there is no abstract configuration Δ' derivable such that Δ' abstracts c', then c' is not reachable from c. Consequently, if it is possible to enumerate all abstractions of c', checking non-derivability (as it is supported by the fixedpoint engines of modern SMT and datalog solvers) gives us a procedure for proving unreachability of program configurations.

8.2 Abstract Semantics

We overview additional details of the analysis definition introduced in § 7.2.

First, we formally define the orders on the abstract argument domains for the predicates defined in Figure 7.2.

$$\begin{aligned}
\leq_{\hat{D}} &:= \{(\hat{a}, \hat{b}) \mid \hat{b} = \top \vee \hat{a} = \hat{b}\} \\
\leq_{\mathbb{N}} &:= \{(m, n) \mid m = n\} \\
\leq_{\mathcal{L}(\hat{D})} &:= \{(\widehat{s}, \widehat{s'}) \mid \widehat{s} = \widehat{s'} = \epsilon \vee (\widehat{s} = \hat{a}{::}\,\widehat{s''} \wedge \widehat{s'} = \hat{b}{::}\,\widehat{s'''} \\
&\qquad \wedge \hat{a} \leq_{\hat{D}} \hat{b} \wedge \widehat{s''} \leq_{\mathcal{L}(\hat{D})} \widehat{s'''})\}
\end{aligned}$$

We assume that the same orders apply to the same argument domains of different predicates.

Some of the partially ordered sets described by the argument domains and their corresponding order, have a supremum, as formally stated in the following lemma:

Lemma 31 (Suprema of argument domains)**.** *The following statements hold:*

- $\forall \hat{a} \in \hat{D}.\ \hat{a} \leq_{\hat{D}} \top$
- $\forall \widehat{s} \in \mathcal{L}(\hat{D}).\ \widehat{s} \leq_{\mathcal{L}(\hat{D})} \widehat{T}$, *s.t.* $\widehat{T} = (\lambda l. \top{::}l)^{|\widehat{s}|}(\epsilon)$

8.3 Proofs

We will go for a direct proof of the statement in Theorem 4 proceeding by complete induction on the number of small-steps.

For reasoning about the soundness, we first need to state a general property of the small-step execution. We adopt from [SGSM20] the following lemma which summarizes a general property of callstack evolution during the execution. The small-step semantics is designed such that the callstack records the execution state as at the point of calling. The corresponding states only get modified when returning from an internal transaction. In this case, modification is guaranteed, since the gas for the execution is subtracted. As a consequence, an unmodified (sub) callstack indicates that the execution of the same internal transaction is still executed. More formally this is captured by the following lemma:

Lemma 32 (Callstack preservation during execution [SGSM20])**.** *Let* (Γ, S) *be a configuration such that* $\Gamma \vDash U +\!\!+ S \to^* U' +\!\!+ S$. *Then the following properties hold:*

- *if* $U' = \epsilon$ *then* $U = \epsilon$
- *if* $U = \epsilon$ *and* $U' \neq \epsilon$ *then there is* $\mathcal{S} \in \mathcal{S}$ *such that* $\Gamma \vDash S \to \mathcal{S}{::}S$ *and* $\Gamma \vDash \mathcal{S}{::}S \to^* U' +\!\!+ S$.
- *if* $U' \neq \epsilon$ *and* $\Gamma \vDash U +\!\!+ S \to^* S'$ *and* $\Gamma \vDash S' \to^* U' +\!\!+ S$ *then there exists* U'' *such that* $|U''| > 0$ *and* $S' = U'' +\!\!+ S$

We introduce the notion of a *call state* for characterizing those states that invoke internal transactions.

Definition 33 (Call states)**.** A regular execution state $\mathcal{S}$ is a call state if $\Gamma \vDash \mathcal{S}::S \rightarrow \mathcal{S}'::\mathcal{S}::S$ for some Γ, S and $\mathcal{S}'$.

Lemma 33 [SGSM20] states that in a regular execution all elements of a callstack but its top element are call states.

Lemma 33. *Let $\Gamma \vDash s::S \rightarrow^* s'::S' + S$, then every execution state $s'' \in S'$ is a call state.*

8.3.1 Monotonicity of Abstract Rules

We prove separately, that all abstract semantics rules are monotone. This facilitates the reasoning in the main proof, since it allows us to argue about most concrete abstractions only.

Since monotoniticy is independent of the small-step semantics, we will in the following consider an abstract semantics specified by $(\mathcal{D}, \mathcal{S}, \Lambda)$. First, we define monotonicity for an abstract semantics $(\mathcal{D}, \mathcal{S}, \Lambda)$ as follows:

Definition 34 (Monotonicity of abstract Semantics [SGSM20])**.** An abstract semantics $(\mathcal{D}, \mathcal{S}, \Lambda)$ is monotone if for all abstract configurations Δ_I, Δ_I', $\Delta_F \in \mathcal{A}_\mathcal{S}$ such that $\Delta_I \leq \Delta_I'$ it holds that

$$\Delta_I \cup \Lambda \vdash \Delta_F \implies \exists \Delta_F'.\ \Delta_I' \cup \Lambda \vdash \Delta_F' \wedge \Delta_F \leq \Delta_F'$$

We will prove the following theorem:

Theorem 6 (Monotonicity)**.** *It holds that $(\mathcal{D}_{evm}, \mathcal{S}_{evm})$ is monotone. (Where $\mathcal{D}_{evm}$ is the super domain and $\mathcal{S}_{evm}$ is the signature induced by definition in Figure 7.2.)*

We prove this property by proving (one-step) monotonicity of the individual rules.

We define one-step derivations of a Horn clause H from some abstract configuration Δ. To this end, we use the notion of a variable assignment $V \in \mathit{Vars} \rightarrow \mathcal{D}$ that maps the variables to values of the corresponding abstract domain. We write $V(n(\vec{z}))$ for $n(V(\vec{z}))$ and $V(\{f_1, \ldots, f_n\})$ for $\{V(f_1), \ldots, V(f_n)\}$. By $V \vDash \Phi$ we denote that replacing all variables in Φ according to V yields a tautology.

Definition 35 (One-step derivability from Horn clause [SGSM20])**.** Let $(\mathcal{D}, \mathcal{S}, \Lambda)$ be an abstract semantics and $(\forall X.\ \Phi, P \Rightarrow c) \in \Lambda$. Further let $f \in \mathcal{A}_\mathcal{S}$ Then the one-step derivability relation $\vdash^1$ on abstract configurations is defined as follows:

$$\Delta, (\forall X.\ \Phi, P \Rightarrow c) \vdash^1 f := \exists V.\ V(P) \subseteq \Delta\ \wedge\ V \vDash \Phi\ \wedge\ f = V(c)$$

Note that this intuition implicitly enforces that the valuation V respects the argument types of the predicates.

We extend the notion of derivability to sets of Horn clauses and abstract configurations:

Definition 36 (One-step derivability from abstract semantics [SGSM20])**.** Let $(\mathcal{D}, \mathcal{S}, \Lambda)$ be an abstract semantics. Then the one-step derivability relation $\vdash^1$ on Λ is defined as follows

$$\Delta, \Lambda \vdash^1 \Delta' := \exists f.\ \Delta = \Delta \cup \{f\}\ \wedge\ \exists H \in \Lambda.\ \Delta, H \vdash^1 f$$

Finally, we define $\vdash$ to be the reflexive, transitive closure of $\vdash^1$.

We define the monotonicity of a Horn clause as follows:

Definition 37 (Monotonicity of Horn clauses [SGSM20])**.** Let $(\mathcal{D}, \mathcal{S}, \Lambda)$ be an abstract semantics. A constrained Horn clause $H \in \Lambda$ is monotone if for all $\Delta' \geq \Delta$

$$\Delta, H \vdash^1 f \implies \exists f'.\ \Delta', H \vdash^1 f'\ \wedge f' \geq f$$

Evidently, the (one-step) monotonicity of all Horn clauses in an abstract semantics implies the (multi-step) monotonicity of the abstract semantics [SGSM20]:

Lemma 34. *Let $(\mathcal{D}, \mathcal{S}, \Lambda)$ be an abstract semantics such that $\Lambda = \Lambda_1 \uplus ... \uplus \Lambda_n$ for some $n \in \mathbb{N}$. Then for all $i \in \{1...n\}$, if Λ_i is (one-step) monotone, then so is Λ.*

It is hence sufficient to prove the (one-step) monotonicity of all Horn clauses in $(\mathcal{D}_{evm}$, $\mathcal{S}_{evm})$.

For facilitating the proofs, we give a more syntactic characterisation of Horn clause monotonicity [SGSM20]:

Lemma 35. *Let $H = \forall X.\ \Phi, P \Rightarrow c$ be a Horn clause. If for all variable assignments V, V' with $(x, D) \in X \implies V(x) \in D\ \wedge V'(x) \in D$ it holds that*

$$\begin{aligned} & V'(P) \geq V(P) \wedge V \vDash \Phi \\ \implies & \exists V^*.\ V^*(P) = V'(P) \wedge V^*(c) \geq V(c) \wedge V^* \vDash \Phi \end{aligned}$$

then H is monotone.

Proof. Assume that (1)

$$\begin{aligned} & V'(P) \geq V(P) \wedge V \vDash \Phi \\ \implies & \exists V^*.\ V^*(P) = V'(P) \wedge V^*(c) \geq V(c) \wedge V^* \vDash \Phi \end{aligned}$$

holds for valuations as defined above. We show the monotonicity of $H = \forall X.\ \Phi, P \Rightarrow c$. To this end we assume some (2) $\Delta\ \ \geq \Delta$ and (3) $\Delta, H \vdash^1 f$ and show that there is some valuation V' such that $V'(P) \subseteq \Delta'$, $V' \vDash \Phi$ and $V'(c) \geq f$. From (3) it is known that there is some valuation V such that $V(P) \subseteq \Delta$, $V \vDash \Phi$ and $f = V(c)$. From (2), we get that for every $p \in V(P)$ there exists a $p' \in \Delta'$ such that $p \leq p'$. Given that the variables of all premises are distinct, we can easily construct a valuation V' such that $V'(q) = p$ for some $q \in P$ and consequently $V'(P) \subseteq \Delta'$ and $V(P) \leq V'(P)$. Using (1), we get that there is some V^* such that $V^*(P) = V'(P)$ and $V^*(c) \geq V(c)$ and $V^* \vDash \Phi$. Consequently, since $V^*(P) = V'(P) \subseteq \Delta'$ and $V^*(c) \geq V(c) = f$, V^* satisfies all required conditions. □

This lemma reduces proving monotonicity of the constrained Horn clause to proving the monotonicity of the clause's constraints.

We now give a proof for Theorem 6, illustrating the general proof strategy.

Proof. For showing the monotonicity it is sufficient to show the one-step derivability of all rules for all different bytecode instructions in Λ and an arbitrary program counter pc. Hence, let $pc \in \mathbb{N}$ be arbitrary. The proof proceeds using Lemma 35 by case distinction on the instruction set.

PUSH Recall the definition of the rule for PUSH instruction:

$$\Lambda_p = \{\mathsf{push}(pc_1, b, v) \wedge \mathsf{stack}(pc_2, \ \widehat{s}\ ) \wedge pc_1 = pc_2 \implies \mathsf{stack}(pc_2 + b + 1, v{::}\ \widehat{s}\ )\}$$

We want to show that Λ_p is monotone within one step, formally $\Delta_I, \Lambda_p \vdash^1 f \implies \forall \Delta'_I \geq \Delta_I . \exists f'. \Delta'_I, \Lambda_p \vdash^1 f' \wedge f' \geq f$. Assume that $\Delta_I, \Lambda_p \vdash^1 f$ then the rule got applied. So $\{\mathsf{push}(pc_1, b, v), \mathsf{stack}(pc_2, \ \widehat{s}\ )\} \in \Delta_I$ such that $pc_1 = pc_2$ and $\mathsf{stack}(pc_2 + b + 1, v{::}\ \widehat{s}\ ) = f$.

Let now be $\Delta'_I \geq \Delta_I$. Then we know that $\{\mathsf{push}(pc'_1, b', v'), \mathsf{stack}(pc'_2, \ \widehat{s}\ ')\} \in \Delta'_I$ such that $pc'_1 \geq pc_1$, $pc'_2 \geq pc_2$, $b' \geq b$, $v' \geq v$, and $\widehat{s}\ ' \geq \widehat{s}$.

Since $pc_1 \in \mathbb{N}$, by the definition of $\leq_{\mathbb{N}}$ from $pc'_1 \geq pc_1$ we conclude that $pc'_1 = pc_1$, since $pc_2 \in \mathbb{N}$ from $pc'_2 \geq pc_2$ we similarly conclude that $pc'_2 = pc_2$. From $pc'_1 = pc_1$, $pc'_2 = pc_2$, and $pc_1 = pc_2$ we know that $pc'_1 = pc'_2$.

Hence we can apply the rule and within one step derive $\mathsf{stack}(pc'_2 + b' + 1, v'{::}\ \widehat{s}\ ') = f'$. By assumption we know that $pc'_2 \geq pc_2$, $b' \geq b$, $v' \geq v$, and $\widehat{s}\ ' \geq \widehat{s}$, hence $f' \geq f$ which concludes the case.

DUP Recall the definition of the rule for DUP instruction:

$$\Lambda_d = \begin{array}{l} \{\mathsf{dup}(pc_1, n_1) \wedge \mathsf{stack}(pc_2, \ \widehat{s_1} \ +\!\!+ (\ \widehat{v_{n_2-1}} :: \widehat{s_2} \ )) \wedge | \ \widehat{s_1} \ | = n_2 - 1 \\ \wedge\ pc_1 = pc_2 \wedge n_1 = n_2 \implies \mathsf{stack}(pc_2 + 1, \ \widehat{v_{n-1}} ::(\ \widehat{s_1} \ +\!\!+ (\ \widehat{v_{n-1}} :: \widehat{s_2} \ )))\} \end{array}$$

We want to show that Λ_d is monotone within one step, formally $\Delta_I, \Lambda_d \vdash^1 f \implies \forall \Delta'_I \geq \Delta_I . \exists f'. \Delta'_I, \Lambda_d \vdash^1 f' \wedge f' \geq f$. Assume that $\Delta_I, \Lambda_d \vdash^1 f$ then the rule got applied. So $\{\mathsf{dup}(pc_1, n_1), \mathsf{stack}(pc_2, \ \widehat{s}\ )\} \in \Delta_I$ such that $pc_1 = pc_2$, $n_1 = n_2$,

$\widehat{s} = \widehat{s_1} +\!\!+ (\widehat{v_{n_2-1}} :: \widehat{s_2}), |\widehat{s_1}| = n_2 - 1$, and $\mathsf{stack}(pc_2 + 1, \widehat{s_c}) = f$ such that $\widehat{s_c} = \widehat{v_{n_2-1}} ::(\widehat{s_1} +\!\!+ (\widehat{v_{n_2-1}} :: \widehat{s_2}))$.

Let now be $\Delta'_I \geq \Delta_I$. Then we know that $\{\mathsf{dup}(pc'_1, n'_1), \mathsf{stack}(pc'_2, \widehat{s}\,')\} \in \Delta'_I$ such that $pc'_1 \geq pc_1$, $pc'_2 \geq pc_2$, $n'_1 \geq n_1$, and $\widehat{s}\,' \geq \widehat{s}$. We know that $\widehat{s} \in \mathcal{L}(\hat{D})$, since $\widehat{s}\,' \geq \widehat{s}$ we conclude that $|\widehat{s}\,'| = |\widehat{s}|$, and $\widehat{s}\,' = \widehat{s_1}\,' +\!\!+ (\widehat{v_{n'_2-1}} :: \widehat{s_2}\,')$ such that $\widehat{s_1}\,' \geq \widehat{s_1}$, $n'_2 = |\widehat{s_1}\,'|$, $\widehat{v_{n'_2-1}} \geq \widehat{v_{n_2-1}}$, and $\widehat{s_2}\,' \geq \widehat{s_2}$. Consequently, by the definition of $\leq_{\mathcal{L}(\hat{D})}$ we know that $|\widehat{s_1}\,'| = |\widehat{s_1}|$ and $n'_2 = n_2$.

Since $pc_1 \in \mathbb{N}$, from $pc'_1 \geq pc_1$ we conclude that $pc'_1 = pc_1$, since $pc_2 \in \mathbb{N}$ from $pc'_2 \geq pc_2$ we similarly conclude that $pc'_2 = pc_2$. From $pc'_1 = pc_1$, $pc'_2 = pc_2$, and $pc_1 = pc_2$ we know that $pc'_1 = pc'_2$. Now we similarly show that $n'_1 = n'_2$. Since $n_1 \in \mathbb{N}$, from $n'_1 \geq n_1$ we conclude that $n'_1 = n_1$. From $n'_1 = n_1$, $n'_2 = n_2$, and $n_1 = n_2$ we know that $n'_1 = n'_2$.

Now we can apply the rule and within one step derive $\mathsf{stack}(pc'_2 + 1, \widehat{s_c}\,') = f'$ such that $\widehat{s_c}\,' = \widehat{v_{n'_2-1}} ::(\widehat{s_1}\,' +\!\!+ (\widehat{v_{n'_2-1}} :: \widehat{s_2}\,'))$. By assumption we know that $\widehat{v_{n'_2-1}} \geq \widehat{v_{n_2-1}}$, $\widehat{s_1}\,' \geq \widehat{s_1}$, and $\widehat{s_2}\,' \geq \widehat{s_2}$. Thus, again by definition of $\leq_{\mathcal{L}(\hat{D})}$ we know that $\widehat{s_c}\,' \geq \widehat{s_c}$. Hence $f' \geq f$ which concludes the case.

SWAP Recall the definition of the rule for SWAP instruction:

$$\Lambda_s = \begin{array}{l}\{\mathsf{swap}(pc_1, n_1) \wedge \mathsf{stack}(pc_2, \widehat{v} ::(\widehat{s_1} +\!\!+ (\widehat{v_{n_2}} :: \widehat{s_2}))) \wedge |\widehat{s_1}| = n_2 \\ \wedge\ pc_1 = pc_2 \wedge n_1 = n_2 \implies \mathsf{stack}(pc_2 + 1, \widehat{v_n} ::(\widehat{s_1} +\!\!+ (\widehat{v} :: \widehat{s_2})))\}\end{array}$$

This case is similar to DUP case, we only need to account for the v value. Assume that there is a variable assignment satisfying the rule constraints, in particular there is value $\widehat{s}$ such that $\widehat{s} = \widehat{v} ::(s_1 +\!\!+ (\widehat{v_{n_2}} :: s_2))$. Now assume $\widehat{s}\,'$ such that $\widehat{s}\,' \geq_{\mathcal{L}(\hat{D})} \widehat{s}$. From $\widehat{s}\,' \geq_{\mathcal{L}(\hat{D})} \widehat{s}$ we know that $|\widehat{s}\,'| = |\widehat{s}|$. Hence, there is $\widehat{v_{n'_2}}$ and $\widehat{v'}$ such that $\widehat{s}\,' = \widehat{v'} ::(s'_1 +\!\!+ (\widehat{v_{n'_2}} :: s'_2))$. Then $\widehat{v'} \geq_{\hat{D}} \widehat{v}$ follows from $\widehat{s''} \geq_{\mathcal{L}(\hat{D})} \widehat{s}$ immediately which concludes the case.

ADD... We now prove the monotonicity of the rule for that adds $\top$ to the stack and pops several stack's elements. By this rule we abstract the following operations: ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, PC, MSIZE, GAS, ISZERO, NOT, BALANCE, CALLDATALOAD, ADD, MUL, SUB, DIV, SDIV, MOD, SMOD, EXP, SIGNEXTEND, LT, GT, SLT, SGT, EQ, AND, OR, XOR, BYTE, SHA3, ADDMOD, MULMOD, CREATE, DELEGATECALL, STATICCALL, CALL, and CALLCODE. Recall the definition of the clause for addtop in Figure 7.4.

$$\Lambda_a = \begin{array}{l}\{\mathsf{addtop}(pc_1, n_1) \wedge \mathsf{stack}(pc_2, \widehat{s_1} +\!\!+ \widehat{s_2}) \wedge |\widehat{s_1}| = n_2 \\ \wedge\ pc_1 = pc_2 \wedge n_1 = n_2 \implies \mathsf{stack}(pc_2 + 1, \top :: \widehat{s_2})\}\end{array}$$

We want to show that Λ_a is monotone within one step, formally $\Delta_I, \Lambda_a \vdash^1 f \implies \forall \Delta'_I \geq \Delta_I. \exists f'. \Delta'_I, \Lambda_a \vdash^1 f' \wedge f' \geq f$. Assume that $\Delta_I, \Lambda_a \vdash^1 f$ then the rule got applied. So $\{\mathsf{addtop}(pc_1, n_1), \mathsf{stack}(pc_2, \widehat{s})\} \in \Delta_I$ such that $pc_1 = pc_2$, $n_1 = n_2$, $\widehat{s} = \widehat{s_1} + \widehat{s_2}$, $|\widehat{s_1}| = n_2$, and $\mathsf{stack}(pc_2 + 1, \widehat{s_c}) = f$ such that $\widehat{s_c} = \top :: \widehat{s_2}$.

Let now be $\Delta_I' \geq \Delta_I$. Then we know that $\{\mathsf{addtop}(pc_1', n_1'), \mathsf{stack}(pc_2', \widehat{s}\,')\} \in \Delta_I'$ such that $pc_1' \geq pc_1$, $pc_2' \geq pc_2$, $n_1' \geq n_1$, $\widehat{s}\,' \geq \widehat{s}$. We know that $\widehat{s} \in \mathcal{L}(\hat{D})$, since $\widehat{s}\,' \geq \widehat{s}$ we conclude that $|\widehat{s}\,'| = |\widehat{s}|$, and $\widehat{s}\,' = \widehat{s_1}\,' + \widehat{s_2}\,'$ such that $\widehat{s_1}\,' \geq \widehat{s_1}$ and $\widehat{s_2}\,' \geq \widehat{s_2}$. Consequently, by the definition of $\leq_{\mathcal{L}(\hat{D})}$ we know that $|\widehat{s_1}\,'| = |\widehat{s_1}| = n_2$.

Since $pc_1 \in \mathbb{N}$, from $pc_1' \geq pc_1$ we conclude that $pc_1' = pc_1$, since $pc_2 \in \mathbb{N}$ from $pc_2' \geq pc_2$ we similarly conclude that $pc_2' = pc_2$. From $pc_1' = pc_1$, $pc_2' = pc_2$, and $pc_1 = pc_2$ we know that $pc_1' = pc_2'$. Now we similarly show that $n_1' = n_2$. Since $n_1 \in \mathbb{N}$, from $n_1' \geq n_1$ we conclude that $n_1' = n_1$. From $n_1' = n_1$, and $n_1 = n_2$ we know that $n_1' = n_2$.

Now we can apply the rule and within one step derive $\mathsf{stack}(pc_2' + 1, \widehat{s_c}\,') = f'$ such that $\widehat{s_c}\,' = \top :: \widehat{s_2}\,'$. By assumption we know that $\widehat{s_2}\,' \geq \widehat{s_2}$. The observation that $\top \geq_{\hat{D}} \top$ which trivially follows from Lemma 31. Thus, again by definition of $\leq_{\mathcal{L}(\hat{D})}$ we know that $\widehat{s_c}\,' \geq \widehat{s_c}$. Hence $f' \geq f$ which concludes the case.

MSTORE... Several bytecode instructions are abstracted in the analysis such that they only remove a number of elements starting from the top of the stack: POP, MSTORE, MSTORE8, SSTORE, LOG0, CALLDATACOPY, CODECOPY, LOG1, EXTCODECOPY, LOG2, LOG3, and LOG4. This is captured by the onlypop rule in Figure 7.4:

$$\begin{aligned}\Lambda_o = &\{\mathsf{onlypop}(pc_1, n_1) \wedge \mathsf{stack}(pc_2, \widehat{s_1} + \widehat{s_2}) \wedge |\widehat{s_1}| = n_2 \\ &\wedge\ pc_1 = pc_2 \wedge n_1 = n_2 \implies \mathsf{stack}(pc_2 + 1, \widehat{s_2})\}\end{aligned}$$

As the rules are similar, the reasoning for the previous addtop case applies here, only without the last observation (i.e., $\top \geq_{\hat{D}} \top$).

JUMPDEST The JUMPDEST opcode marks its program counter as a jump destination. We recall the rule for it:

$$\Lambda_{jd} = \{\mathsf{jd}(pc_1) \wedge \mathsf{stack}(pc_2, \widehat{s}) \wedge\ pc_1 = pc_2 \implies \mathsf{stack}(pc_2 + 1, \widehat{s})\}$$

We only sketch this case as it is trivial. Assume that there is some variable assignment satisfying the rule constraints, meaning that there are values pc_1, pc_2, and $\widehat{s}$ such that $pc_1 = pc_2$. Now assume $pc_1' \geq_{\mathbb{N}} pc_1$, $pc_2' \geq_{\mathbb{N}} pc_2$ and $\widehat{s}\,' \geq_{\mathcal{L}(\hat{D})} \widehat{s}$. From $pc_1' \geq_{\mathbb{N}} pc_1$ we know that $pc_1' = pc_1$, from $pc_2' \geq_{\mathbb{N}} pc_2$ we know that $pc_2' = pc_2$, thus $pc_1' = pc_2'$. We know that $\widehat{s}\,' \geq_{\mathcal{L}(\hat{D})} \widehat{s}$ by assumption which concludes the case.

JUMP Recall the definition of the rules for non conditional jump: (J1) describes the situation when the jump is performed to a location which is not $\top$, (J2) describes jumping to all possible destinations when $\top$ is specified as a destination.

$$\begin{aligned}\Lambda_j = \{&\mathsf{jump}(pc_1) \wedge \mathsf{stack}(pc_2, t_1 :: \widehat{s}) \wedge \mathsf{jd}(t_2) \\ &\wedge\ pc_1 = pc_2 \wedge\ t_1 = t_2 \implies \mathsf{stack}(t_2, \widehat{s}), && \text{(J1)} \\ &\mathsf{jump}(pc_1) \wedge \mathsf{stack}(pc_2, \top :: \widehat{s}) \wedge \mathsf{jd}(pc') \wedge\ pc_1 = pc_2 \implies \mathsf{stack}(pc', \widehat{s})\} && \text{(J2)}\end{aligned}$$

We want to show that Λ_j is monotone within one step, formally $\Delta_I, \Lambda_j \vdash^1 f \implies \forall \Delta'_I \geq \Delta_I . \exists f' . \Delta'_I, \Lambda_j \vdash^1 f' \wedge f' \geq f$. Assume that $\Delta_I, \Lambda_j \vdash^1 f$. Then either (J1) or (J2) got applied. So $\{\mathsf{jump}(pc_1), \mathsf{stack}(pc_2, t_1 :: \widehat{s}\ ), \mathsf{jd}(t_2)\} \in \Delta_I$ such that $pc_1 = pc_2$ and either

(i) $t_1 = t_2$, $t_1 \in \mathbb{N}$, and $\mathsf{stack}(t_2, \widehat{s}\ ) = f$;

(ii) $t_1 = \top$, and $\mathsf{stack}(t_2, \widehat{s}\ ) = f$.

Let now be $\Delta'_I \geq \Delta_I$. Then we know that $\{\mathsf{jump}(pc'_1), \mathsf{stack}(pc'_2, t'_1 :: \widehat{s}\ '), \mathsf{jd}(t'_2)\} \in \Delta'_I$ such that $pc'_1 \geq pc_1$, $pc'_2 \geq pc_2$, $\widehat{s}\ ' \geq \widehat{s}$ (1), $t'_1 \geq t_1$ (2), and $t'_2 \geq t_2$ (3). From $pc'_1 \geq pc_1$ and $pc_1 \in \mathbb{N}$ we conclude that $pc'_1 = pc_1$, from $pc'_2 \geq pc_1$ and $pc_2 \in \mathbb{N}$ we conclude that $pc'_2 = pc_2$. Since $pc'_1 = pc_1$, $pc'_2 = pc_2$, and $pc_1 = pc_2$ it also holds that $pc'_1 = pc'_2$. We proceed by case distinction.

(i) We know that $t_1 = t_2$, $t_1 \in \mathbb{N}$, and $\mathsf{stack}(t_2, \widehat{s}\ ) = f$. Since $t'_1 \in \hat{D}$ either $t'_1 = \top$ (a) or $t'_1 = t_1$ (b). We proceed by case distinction.

   (a) In this case $t'_1 = \top$, hence we can apply (J2) and derive with one step $\mathsf{stack}(t'_2, \widehat{s}\ ') = f'$. We need to show that $\widehat{s}\ ' \geq \widehat{s}$ (holds by assumption (1)) and $t'_2 \geq t_2$ (holds by assumption (3)). Then it follows that $f' \geq f$.

   (b) In this case as $t'_1 = t_1$ and consequently $t'_1 = t_2$, hence we can apply (J1) and derive within one step $\mathsf{stack}(t_2, \widehat{s}\ ') = f'$. We need to show that $\widehat{s}\ ' \geq \widehat{s}$ (holds by assumption (1)) and $t_2 \geq t_2$ (trivially holds by the definition of $\leq_{\mathbb{N}}$). Then it follows that $f' \geq f$.

(ii) We know that $t_1 = \top$, and $\mathsf{stack}(t_2, \widehat{s}\ ) = f$. Since $t'_1 \geq t_1$ by assumption (2) we also know that $t'_1 = \top$ by the definition of $\leq_{\hat{D}}$. Consequently we can apply the (J2) rule and derive within one step $\mathsf{stack}(t'_2, \widehat{s}\ ') = f'$. We need to show that $\widehat{s}\ ' \geq \widehat{s}$ (holds by assumption (1)) and $t'_2 \geq t_2$ (holds by assumption (3)). Then it follows that $f' \geq f$.

This concludes the proof.

JUMPI Recall the definition of the rules for conditional jump:

$$\begin{aligned}
\Lambda_{ji} = \{&\mathsf{jumpi}(pc_1) \wedge \mathsf{stack}(pc_2, t_1 :: br :: \widehat{s}\ ) \wedge \mathsf{jd}(t_2) \\
&\wedge\ pc_1 = pc_2 \wedge\ t_1 = t_2 \implies \mathsf{stack}(t_2, \widehat{s}\ ), && \text{(JI1)} \\
&\mathsf{jumpi}(pc_1) \wedge \mathsf{stack}(pc_2, \top :: br :: \widehat{s}\ ) \wedge \mathsf{jd}(pc') \\
&\wedge\ pc_1 = pc_2 \implies \mathsf{stack}(pc', \widehat{s}\ ), && \text{(JI2)} \\
&\mathsf{jumpi}(pc_1) \wedge \mathsf{stack}(pc_2, t :: br :: \widehat{s}\ ) \wedge\ pc_1 = pc_2 \implies \mathsf{stack}(pc_2 + 1, \widehat{s}\ )\} && \text{(JI3)}
\end{aligned}$$

This case is similar to the JUMP case since there are no conditions on the br value.

This concludes the proof. □

8.3.2 Main Proof

As the property of interest in this work is local, we show *local soundness* that reasons only about the local execution states.

Theorem (Local Soundness)*.* Let Γ be a transaction environment and let S and S' be arbitrary callstacks. Then for all local execution states $\mathcal{S}$ it holds that

$$\begin{aligned}\Gamma \vDash \mathcal{S}::S \rightarrow^{*} S' \mathbin{+\!\!+} \mathcal{S}'::S \implies \forall \Delta_I.\ \alpha_s(\mathcal{S}) \leq \Delta_I \\ \implies \exists \Delta.\ \Delta_I, \Lambda \vdash \Delta \quad \wedge\ \alpha_s(\mathcal{S}') \leq \Delta\end{aligned}$$

We will give a proof for the most interesting cases of the soundness proof, providing formal arguments for the soundness of local operations.

Proof. By complete induction on the number n of small-steps.

- Case $n = 0$. In the case of the empty reduction sequence, we have that $\mathcal{S}' = \mathcal{S}$ and consequently the claim trivially follows by the reflexivity of $\vdash$.
- Case $n > 0$. Let $\Gamma \vDash \mathcal{S}::S \rightarrow^{n-1} S^{+}$ and $\Gamma \vDash S^{+} \rightarrow S' \mathbin{+\!\!+} \mathcal{S}'::S$. By Lemma 32, it holds that $S^{+} = S^{*} \mathbin{+\!\!+} \mathcal{S}^{*}::S$ for some S^{*} and $\mathcal{S}^{*}$. By the inductive hypothesis we know that for all $\Delta_I \geq \alpha_s(\mathcal{S})$ there is some $\Delta_{\mathcal{S}^*} \geq \alpha_s(\mathcal{S}^*)$ such that $\Delta_I, \Lambda \vdash \Delta_{\mathcal{S}^*}$. Consequently, for proving the claim, it is sufficient to show that there is some $\Delta_{\mathcal{S}'} \geq \alpha_s(\mathcal{S}')$ such that $\Lambda, \Delta_{\mathcal{S}^*} \vdash \Delta_{\mathcal{S}'}$. We proceed by case distinction on the shapes of S' and S^{*}.

  (i) $S^{*} = S' = \epsilon$. In this case the last step that is performed is a local execution step. Assume $\mathcal{S}^{*} = (\mu, \iota, \sigma, \eta)$ and $\mathcal{S}' = (\mu', \iota', \sigma', \eta')$. Let in the following $c = \iota.\mathsf{code}$ and $pc = \mu.pc$. By Lemma 29 we know that $\mathcal{S}.\iota = \mathcal{S}'.\iota'$, and consequently also $\mathcal{S}.\iota.code = \mathcal{S}'.\iota'.code = c$. We proceed by case analysis on the rule applied in the last reduction step:

  PUSH In this case $c[pc] = \mathsf{PUSHb}$, $c[pc+1, pc+\mathsf{b}] = \mathsf{val}$, $\mathcal{S}.\mu.\mathsf{s} = s$, $\mathcal{S}'.\mu'.\mathsf{s} = \mathsf{val}::s$, and $\mu'.pc = pc+1$. Also in this case by the definition of α_s presented in Figure 7.3 we observe that $\alpha_s(\mathcal{S}^*) = \{\mathsf{stack}(pc, s)\} \cup \alpha_{pc}(0, c)$ and $\mathsf{push}(pc) \in \alpha_{pc}(0, c)$. From these observations one can conclude applying (P) that $\alpha_s(\mathcal{S}^*), \Lambda \vdash \{\mathsf{stack}(pc + \mathsf{b} + 1, \mathsf{val}::s)\} \cup \alpha_{pc}(0, c) = \alpha_s(\mathcal{S}')$. Consequently also by the monotonicity (Theorem 6) there is some $\Delta_{\mathcal{S}'} \geq \alpha_s(\mathcal{S}')$ such that $\Lambda, \Delta_{\mathcal{S}^*} \vdash \Delta_{\mathcal{S}'}$ which concludes the proof.

  DUP, SWAP The cases for DUP and SWAP are similar to PUSH, that is we start by reasoning about the corresponding rules applicability and conclude using the monotonicity result.

  ADD... The case covers a number of instructions i, such that $i \in$ [ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, PC, MSIZE, GAS, ISZERO, NOT, BALANCE, CALLDATALOAD, ADD, MUL, SUB, DIV, SDIV, MOD, SMOD, EXP, SIGNEXTEND, LT, GT, SLT, SGT, EQ, AND, OR, XOR, BYTE, SHA3, ADDMOD, MULMOD]. We will show the case for ADD instruction as the reasoning for the other instructions may only differ on the number of the stack elements that are discarded from the stack. In this case $c[pc] = \mathsf{ADD}$, $\mathcal{S}.\mu.\mathsf{s} = v_1{::}v_2{::}s$, $\mathcal{S}'.\mu'.\mathsf{s} = v_1 + v_2{::}s$, and $\mu'.pc = pc+1$. Also in this case by the definition of α_s presented in Figure 7.3 we observe that $\alpha_s(\mathcal{S}^*) = \{\mathsf{stack}(pc, v_1{::}v_2{::}s)\} \cup \alpha_{pc}(0, c)$ and $\mathsf{addtop}(pc, 2) \in \alpha_{pc}(0, c)$. From these observations one can conclude applying (A) that $\alpha_s(\mathcal{S}^*), \Lambda \vdash \{\mathsf{stack}(pc+1, \top{::}s)\} \cup \alpha_{pc}(0, c) \geq \alpha_s(\mathcal{S}')$. Consequently also by the monotonicity (Theorem 6) there is some $\Delta_{\mathcal{S}'} \geq \alpha_s(\mathcal{S}')$ such that $\Lambda, \Delta_{\mathcal{S}^*} \vdash \Delta_{\mathcal{S}'}$ which concludes the proof.

MSTORE... The case covers a number of instructions i, such that $i \in$ [POP, MSTORE, MSTORE8, SSTORE, LOG0, CALLDATACOPY, CODECOPY, LOG1, EXTCODE-COPY, LOG2, LOG3, LOG4]. This case is similar to the ADD... case, as the values are discarded from the stack in the same way as in the ADD... case, but nothing is added to the stack.

JUMPDEST This case is similar to MSTORE..., only JUMPDEST makes no changes to the stack advancing the program counter by one.

JUMPI In this case $c[pc] = \mathsf{JUMPI}$, $\mathcal{S}.\mu.\mathsf{s} = t{:}br{::}s$, $\mathcal{S}'.\mu'.\mathsf{s} = s$ and either $\mu'.pc = pc{+}1$ (a) or $\mu'.pc = t$ (b). We proceed by case distinction.

(a) In this case by the definition of α_s presented in Figure 7.3 we observe that $\alpha_s(\mathcal{S}^*) = \{\mathsf{stack}(pc, t :: br{::}s)\} \cup \alpha_{pc}(0, c)$ and $\mathsf{jumpi}(pc) \in \alpha_{pc}(0, c)$. From these observations one can conclude applying (JI3) that $\alpha_s(\mathcal{S}^*), \Lambda \vdash \{\mathsf{stack}(pc+1, s)\} \cup \alpha_{pc}(0, c) = \alpha_s(\mathcal{S}')$. Consequently also by the monotonicity (Theorem 6) there is some $\Delta_{\mathcal{S}'} \geq \alpha_s(\mathcal{S}')$ such that $\Lambda, \Delta_{\mathcal{S}^*} \vdash \Delta_{\mathcal{S}'}$ which concludes the proof.

(b) In this case by the definition of α_s presented in Figure 7.3 we observe that $\alpha_s(\mathcal{S}^*) = \{\mathsf{stack}(pc, t' :: br{::}s)\} \cup \alpha_{pc}(0, c)$, and $\mathsf{jumpi}(pc)\} \in \alpha_{pc}(0, c)$. As we know $\mathcal{S}'$ is a regular execution state, we know that $t' \in D(c)$, consequently $\mathsf{jd}(t') \in \alpha_{pc}(0, c)$. From these observations applying (JI1) one can conclude that $\alpha_s(\mathcal{S}^*), \Lambda \vdash \{\mathsf{stack}(t, \mu)s\} \cup \alpha_{pc}(0, c) = \alpha_s(\mathcal{S}')$. Consequently also by the monotonicity (Theorem 6) there is some $\Delta_{\mathcal{S}'} \geq \alpha_s(\mathcal{S}')$ such that $\Lambda, \Delta_{\mathcal{S}^*} \vdash \Delta_{\mathcal{S}'}$ which concludes the proof.

JUMP This case is similar to JUMPI case, but it does not include the reasoning for the branching condition, and the rule for advancing the program counter by one.

(ii) $(|S^*| > 0 \wedge S^* \neq [HALT(\sigma, g, d, \eta)] \wedge S^* \neq [EXC])$, or $|S'| > 0$. Consequently there is some execution in S^* or S', so the execution state $\mathcal{S}^*$ stays unchanged, therefore the claim follows from the inductive hypothesis.

(iii) $(S^* = [HALT(\sigma, g, d, \eta)] \vee S^* = [EXC]) \wedge S' = \epsilon$. By Lemma 33 we know that $\mathcal{S}^*$ is a call state, hence we know that one of the call instructions among CREATE, DELEGATECALL, STATICCALL, CALL, and CALLCODE is at $\mathcal{S}^*.\iota.\mathsf{code}[pc]$. Consequently the effects of the rules that alter the call stack apply. In the following we consider the CALL instruction, but the reasoning for the other call instructions is similar. Let $\mathcal{S}^* = (\mu, \iota, \sigma, \eta)$ and $\mathcal{S}' = (\mu', \iota', \sigma', \eta')$. Let in the following $c = \iota.\mathsf{code}$, $pc = \mu.pc$, and $\mu.\mathsf{s} = v1 :: v2 :: v3 :: v4 :: v5 :: v_6 :: v_7 :: s$. By Lemma 29 we know that $\mathcal{S}.\iota = \mathcal{S}'.\iota'$, and consequently also $\mathcal{S}.\iota.code = \mathcal{S}'.\iota'.code = c$. Consequently, as the call from which S' returns is at program counter pc, that is $c[pc] = \mathsf{CALL}$, we know that $\mu'.pc = pc + 1$. As the effect of the call needs to be stored in the stack we know that $\mathcal{S}'.\mu'.\mathsf{s} = 1 :: s$ in case $S' = HALT(\sigma', g', d', \eta')$ or $\mathcal{S}'.\mu'.\mathsf{s} = 0 :: s$ in case $S' = EXC$. In our abstraction we over-approximate both outcomes by $\top$ as depicted in Figure 7.3. From the definition of α_s also presented in Figure 7.3 we observe that $\alpha_s(\mathcal{S}^*) = \{\mathsf{stack}(pc, \mu.v1 :: v2 :: v3 :: v4 :: v5 :: v_6 :: v_7 :: s)\} \cup \alpha_{pc}(0, c)$ and $\mathsf{addtop}(pc, 7) \in \alpha_{pc}(0, c)$. From these observations one can conclude applying (A) that $\alpha_s(\mathcal{S}^*), \Lambda \vdash \{\mathsf{stack}(pc + 1, \top :: s)\} \cup \alpha_{pc}(0, c) \geq \alpha_s(\mathcal{S}')$. Consequently also by the monotonicity (Theorem 6) there is some $\Delta_{\mathcal{S}'} \geq \alpha_s(\mathcal{S}')$ such that $\Lambda, \Delta_{\mathcal{S}^*} \vdash \Delta_{\mathcal{S}'}$ which concludes the proof.

This concludes the proof. $\square$

CHAPTER 9

Conclusions

This thesis presents novel static analysis techniques to verify security properties of low-level code, which have been successfully applied in the context of Android applications and smart contracts. These techniques are built on *Horn-clause based abstraction*, which allows for leveraging the state-of-the-art Satisfiability Modulo theory and Datalog solvers that perform the analysis task via Horn clause resolution.

Specifically, this work discusses HornDroid, a tool for the static analysis of Android applications. HornDroid is the first static analysis tool for Android that comes with a formal proof of soundness. Based on an available benchmark proposed by the static analysis community, we experimentally show that HornDroid is the first tool to detect all the existing explicit information flows. However, HornDroid employs flow-insensitive heap abstraction, consequently, its precision can be improved. Therefore, this thesis makes a further step towards sound information flow analysis of real Android applications, presenting the first static analysis for Android applications which is both flow-sensitive on the heap abstraction and provably sound with respect to a rich formal model of the Android ecosystem. In this work, we adapt ideas from recency abstraction [BR06] to hit a sweet spot in the analysis design space: our proposal is sound, precise, and efficient in practice. We substantiate these claims by implementing the analysis as a tool fsHornDroid and performing an experimental evaluation of our tool.

Moreover, this work formalizes a large fragment of small-step semantics of EVM bytecode in the F* proof assistant, successfully validating it against the official Ethereum test suite. An in-depth study of EVM semantics facilitates our design of an efficient static analysis technique for the control flow graph recovery. We implemented our solution and successfully conducted its large-scale evaluation, demonstrating the practicality of our approach. The soundness of the analysis is formally proven against the semantics mentioned above.

Although this thesis focuses on using Horn-clause based abstraction techniques to build sound static analysis solutions, one of the future directions is to use Horn clause resolution in bug finding: our ongoing work employs a Horn-clause based analysis tool for LLVM bytecode [GKKN15] in order to discover exploitation primitives in dynamic allocator implementations. Other future directions include formulating non-interference properties in terms of Horn clauses to consider implicit flows in Android applications and incorporating declassification mechanisms to make the analysis aware of intended information flows.

List of Figures

List of Tables

Bibliography

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

[Adh17] Chandra Adhikari. Secure framework for healthcare data management using Ethereum-based blockchain technology. 2017.

[AEVL16] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. In *Open and Big Data (OBD), International Conference on*, pages 25–30. IEEE, 2016.

[And94] Lars Ole Andersen. Program analysis and specialization for the C programming language. Technical report, University of Copenhagen, 1994.

[ARF$^+$14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, pages 259–269. ACM, 2014.

[BCS13] Michele Bugliesi, Stefano Calzavara, and Alvise Spanò. Lintent: Towards security type-checking of Android applications. In *FMOODS/FORTE*, pages 289–304, 2013.

[BDLF$^+$16] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016.

[BJK$^+$18] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[BKT17] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. Findel: Secure derivative contracts for Ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 453–467. Springer, 2017.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM, 2011.

[BMR12] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. In *SMT*, pages 3–11. ACM, 2012.

[BR06] Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In *SAS*, pages 221–239. Springer-Verlag, 2006.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245. ACM, 1993.

[cfg20] evm-cfg-builder. https://github.com/crytic/evm_cfg_builder, 2020.

[CGKM17] Stefano Calzavara, Ilya Grishchenko, Adrien Koutsos, and Matteo Maffei. A sound flow-sensitive heap abstraction for the static analysis of Android applications. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2fry017*, pages 22–36. IEEE Computer Society, 2017.

[CGM16] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. HornDroid: Practical and sound static analysis of Android applications by SMT solving. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 47–62. IEEE, 2016.

[Cha09] Avik Chaudhuri. Language-based security on Android. In *PLAS*, pages 1–7. ACM, 2009.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. *SIGPLAN Not.*, 35(5):35–46, May 2000.

[DDA11] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *POPL*, pages 187–200. ACM, 2011.

[DMB08a] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[dMB08b] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340. Springer-Verlag, 2008.

[DWA$^+$17] Changyu Dong, Yilei Wang, Amjad Aldweesh, Patrick McCorry, and Aad van Moorsel. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. 2017.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *SIGPLAN Not.*, 29(6):242–256, June 1994.

[EGH$^+$14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, 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. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, 2014.

[evm] Consensus test suite. Available at `https://github.com/ethereum/tests`.

[FCF09] Adam P. Fuchs, Avik Chaudhuri, , and Jeffrey S. Foster. Scandroid: Automated security certification of Android applications. Technical report, University of Maryland, 2009.

[FCH$^+$11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *CCS*, pages 627–638, 2011.

[FD12] S. Fink and J. Dolby. WALA – The TJ Watson libraries for analysis, 2012.

[fsh] fsHornDroid. Available online at `https://secpriv.tuwien.ac.at/tools/horndroid/`.

[fst] F*. Available at `https://fstar-lang.org`.

[FWM$^+$11] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[GBCS07] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277. ACM, 2007.

[GBSS19] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: Thorough, declarative decompilation of smart contracts. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 1176–1186. IEEE Press, 2019.

[GCEC12] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *TRUST*, pages 291–307. Springer-Verlag, 2012.

[GKKN15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 343–361, 2015.

[GKP⁺15] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information flow analysis of Android applications in DroidSafe. In *NDSS*. IEEE, 2015.

[GMS18a] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of Ethereum smart contracts. In *Proceedings of the 7th International Conference on Principles of Security and Trust (POST)*, pages 243–269. Springer, 2018.

[GMS18b] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of Ethereum smart contracts - technical report and F* formalisation, 2018. Available at `https://secpriv.tuwien.ac.at/tools/ethsemantics`.

[HB12] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171. Springer-Verlag, 2012.

[HHJ⁺11] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart E. Schechter, and David Wetherall. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *CCS*, pages 639–652. ACM, 2011.

[Hir17] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *1st Workshop on Trusted Smart Contracts*, 2017.

[HL07] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. *SIGPLAN Not.*, 42(6):290–299, June 2007.

[HSLC17] Adam Hahn, Rajveer Singh, Chen-Ching Liu, and Sijie Chen. Smart contract-based campus demonstration of decentralized transactive energy auctions. In *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2017 IEEE*, pages 1–5. IEEE, 2017.

[HSR⁺18] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. Kevm: A complete formal semantics of the Ethereum virtual machine. In *Proceedings - IEEE 31st Computer Security Foundations Symposium, CSF 2018*, Proceedings - IEEE Computer Security Foundations Symposium, pages 204–217. IEEE Computer Society, August 2018. 31st IEEE Computer Security Foundations Symposium, CSF 2018 ; Conference date: 09-07-2018 Through 12-07-2018.

[JAF+13] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on Android - (extended abstract). In *ESORICS*, pages 775–792. ACM, 2013.

[Java] Java 8 Documentation on Object. `https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html`. last accessed on February 2017.

[Javb] Java 8 Documentation on Thread. `https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html`. last accessed on February 2017.

[JMF12] Jinseong Jeon, Kristopher K Micinski, and Jeffrey S Foster. SymDroid: Symbolic execution for Dalvik bytecode. Technical report, University of Maryland, 2012.

[JSS16] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.

[Kah08] Vineet Kahlon. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. *SIGPLAN Not.*, 43(6):249–259, June 2008.

[KGDS18] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing safety of smart contracts. NDSS, 2018.

[KK14] Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *CoRR*, abs/1403.4910, 2014.

[KS13] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. *SIGPLAN Not.*, 48(6):423–434, June 2013.

[KYY+12] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in Android applications. In *MoST*, 2012.

[LBB+15] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *ICSE*, pages 280–291. IEEE Press, 2015.

[LC11] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. *SIGPLAN Not.*, 46(1):3–16, January 2011.

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[LLW$^+$12] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS*, pages 229–240. ACM, 2012.

[LMS$^+$14] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a certifying app store for Android. In *SPSM@CCS*, pages 93–104. ACM, 2014.

[Loc14] Andreas Lochbihler. Making the java memory model safe. *ACM Trans. Program. Lang. Syst.*, 35(4):12:1–12:65, January 2014.

[MFSH17] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. *Proceedings of the Financial Cryptography and Data Security Conference*, 2017.

[MM17] Florian Mathieu and Ryno Mathee. Blocktix: Decentralized event hosting and ticket distribution network. 2017. Available at `https://blocktix.io/public/doc/blocktix-wp-draft.pdf`.

[MS12] Christopher Mann and Artem Starostin. A framework for static detection of privacy leaks in Android applications. In *SAC*, pages 1457–1462. ACM, 2012.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. Available at `http://bitcoin.org/bitcoin.pdf`.

[NGW17] Benedikt Notheisen, Magnus Gödde, and Christof Weinhardt. Trading stocks on blocks-engineering decentralized markets. In *International Conference on Design Science Research in Information Systems*, pages 474–478. Springer, 2017.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of program analysis.* Springer-Verlag, 1999.

[oB18] Trail of Bits. Manticore: Symbolic execution for humans. 2018.

[par17a] The Parity wallet breach, 30 million ether reported stolen, 2017. Available at `https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/`.

[par17b] The Parity wallet vulnerability, 2017. Available at `https://paritytech.io/blog/security-alert.html`.

[PB09] Fernando Magno Quintão Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *GCO*, pages 126–135, 2009.

[PS12] Étienne Payet and Fausto Spoto. Static analysis of Android programs. *Information & Software Technology*, 54(11):1192–1201, 2012.

[PS14] Étienne Payet and Fausto Spoto. An operational semantics for Android activities. In *PEPM*, pages 121–132. ACM, 2014.

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *NDSS*, 2014.

[RR99] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. *SIGPLAN Not.*, 34(5):77–90, May 1999.

[SB15] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015.

[SBL11a] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick Your Contexts Well: Understanding Object-Sensitivity. In *POPL*, pages 17–30. ACM, 2011.

[SBL11b] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, pages 17–30. ACM, 2011.

[SCD$^+$13] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming*, pages 196–232. Springer-Verlag, Berlin, Heidelberg, 2013.

[SGSM20] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and provably sound static analysis of ethereum smart contracts. *arXiv preprint arXiv:2005.06227*, 2020.

[SKB14] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. More sound static handling of Java reflection. Technical report, 2014.

[soua] Soufflé: Datalog specification. Available at `https://souffle-lang.github.io/datalog`.

[soub] Soufflé: Functors. Available at `https://souffle-lang.github.io/functors`.

[SPY$^+$16] Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 74–91. ACM, 2016.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118. ACM, 1999.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41. ACM, 1996.

[TDDC+18] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 67–82. ACM, 2018.

[thea] The DAO smart contract. Available at `http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code`.

[Theb] The Collection of Android Apps and Metadata. `https://archive.org/details/android_apps&tab=about`. last accessed on February 2017.

[Thec] The Java Language Specification. `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`. last accessed on February 2017.

[The16a] The Android Developers Guide. App Components, 2016. Available online at `http://developer.android.com/guide/components/index.html`.

[The16b] The Android Developers Guide. Fragments, 2016. Available online at `http://developer.android.com/guide/components/fragments.html`.

[TR14] Omer Tripp and Julia Rubin. A bayesian approach to privacy enforcement in smartphones. In *USENIX*, pages 175–190. USENIX, 2014.

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC*, pages 18–34. Springer-Verlag, 2000.

[WKOH14] Erik Ramsgaard Wognsen, Henrik Søndberg Karlsen, Mads Chr. Olesen, and René Rydhof Hansen. Formalisation and analysis of Dalvik bytecode. *Sci. Comput. Program.*, 92:25–55, 2014.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014. Available at `https://ethereum.github.io/yellowpaper/paper.pdf`.

[WROR14] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *CCS*, pages 1329–1341. ACM, 2014.

[YY12] Zhemin Yang and Min Yang. LeakMiner: Detect information leakage on Android with static taint analysis. In *WCSE*, pages 101–104. IEEE, 2012.

[ZO12] Zhibo Zhao and Fernando C. Colón Osorio. TrustDroid: Preventing the use of smartphones for information leaking in corporate networks through the use of static analysis taint tracking. In *MALWARE*, pages 135–143. IEEE, 2012.