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

Stefano Calzavara*, Ilya Grishchenko†, Adrien Koutsos‡, Matteo Maffei†

* Università Ca' Foscari Venezia † TU Wien, ‡ LSV, CNRS, ENS Paris-Saclay

Abstract—**The present paper proposes 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 platform. We formulate the analysis as a set of Horn clauses defining a sound over-approximation of the semantics of the Android application to analyse, borrowing ideas from recency abstraction and extending them to our concurrent setting. Moreover, we implement the analysis in HornDroid, a state-of-the-art information flow analyser for Android applications. Our extension allows HornDroid to perform strong updates on heap-allocated data structures, thus significantly increasing its precision, without sacrificing its soundness guarantees. We test our implementation on DroidBench, a popular benchmark of Android applications developed by the research community, and 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. Finally, we assess the scalability of our tool to the analysis of real applications.**

I. INTRODUCTION

Android is today the most popular operating system for mobile phones and tablets, and it boasts the largest application market among all its competitors. Though the huge number of available applications is arguably one of the main reasons for the success of Android, it also poses an important security challenge: there are way too many applications to ensure that they go through a timely and thorough security vetting before their publication on the market. Automated analysis tools thus play a critical role in ensuring that security verification does not fall behind with respect to the release of malicious (or buggy) applications.

There are many relevant security concerns for Android applications, e.g., privilege escalation [11], [5] and component hijacking [26], 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 [41], [42], [27], [13], [22], [3], [40], [14], [6]. 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.

A. Soundness vs. Precision in Android Analyses

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 I shows a leaky code snippet that cannot be detected by FlowDroid [3], a state-of-the-art taint tracker for Android applications[1].

```
1 public class Leaky extends Activity {
2   Storage st = new Storage();
3   Storage st2 = new Storage();
4   onRestart() { st2 = st; }
5   onResume() { st2.s = getDeviceId(); }
6   onPause() { send(st.s, "http://www.myapp.com/"); }
7 }
```

TABLE I
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 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 [6] is the only provably sound static analyser for Android applications to date and, as such, it correctly deals with the code snippet in Table I. 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

---

[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.

limitation of the tool, since it prevents *strong updates* [23] 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 II.

```
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 II
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.

B. Contributions

In the present paper we make the following contributions:

1) we extend an operational semantics for a core fragment of the Android ecosystem [6] with multi-threading and exception handling, in order to provide a more accurate representation of the control flow of Android applications;
2) 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* [4] in order to hit a sweet spot between precision and efficiency, extending it for the first time to a concurrent setting;
3) we implement our analysis as an extension of Horn-Droid [6]. This extension allows HornDroid to perform strong updates on heap-allocated data structures, thus significantly increasing the precision of the tool;
4) we test our extension of HornDroid against DroidBench, a popular benchmark proposed by the research community [3]. 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 and more details on the experiments are available online [1].

II. DESIGN AND KEY IDEAS

A. Our Proposal

Our proposal starts from the pragmatic observation that statically predicting the control flow of an Android application is daunting and error-prone [14]. 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 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 [32], [16], [23], [21]. 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* [4], 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

---

[2]We are aware of the fact that the Java Memory Model allows more behaviours than an interleaving semantics (see [24] 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.

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.

B. Examples

By being conservative on the execution order of callbacks, our analysis is able to soundly analyse the leaky example of Table I. We recall it in Table III, 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.

```
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 III
A SUBTLE INFORMATION LEAK (DETECTED)

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 $H(1, \{|\texttt{Leaky}; \texttt{st} \mapsto \mathsf{NFS}(2), \texttt{st2} \mapsto \mathsf{NFS}(3)|\})$ is introduced, and two facts $H(2, \{|\texttt{Storage}; \texttt{s} \mapsto \texttt{""}|\})$ and $H(3, \{|\texttt{Storage}; \texttt{s} \mapsto \texttt{""}|\})$ abstract the objects pointed by the activity fields st and st2. Then the life-cycle events are abstracted: the onRestart method performs a weak update on the activity object, adding a fact $H(1, \{|\texttt{Leaky}; \texttt{st} \mapsto \mathsf{NFS}(2), \texttt{st2} \mapsto \mathsf{NFS}(2)|\})$ 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 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 II thanks to recency abstraction. We show a simplified version of the facts generated by the analysis in Table IV. 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.

```
1  public class Anon extends Activity {
   H(1, {|Anon; m ↦ NFS(2)|})
   // flow-insensitivity on activity object
2    Contact[] m = new Contact[]();
   H(2, []) // new empty array is created
3    onStart() {
   LState₃(c ↦ null; 5 ↦ ⊥)
   // no allocated contact at location 5 yet
4      for (int i = 0; i < contacts.length(); i++) {
   LState₄(c ↦ null; 5 ↦ ⊥) ∧ LState₄(c ↦ NFS(5); 5 ↦ ⊥)
   // loop invariant (see below)
5        Contact c = contacts.getContact(i);
   LState₅(c ↦ FS(5); 5 ↦ o_c) // flow-sensitivity
6        c.phone = anonymise(c.phone);
   LState₆(c ↦ FS(5); 5 ↦ o_c{phone ↦ ""}) // strong update
7        m[i] = c;
   LState₇(c ↦ NFS(5); 5 ↦ ⊥) ∧ H(5, o_c{phone ↦ ""}) ∧
   H(2, [NFS(5)]) // lifting is performed
8      }
9      send(m, "http://www.cool-apps.com/");
   Sink([o_c{phone ↦ ""}]) // no leak is detected
10   }
11 }
```

TABLE IV
ANONYMIZING CONTACT INFORMATION (ALLOWED)

The local state fact LState_{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 $H(2, [])$. 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 $H(5, o_c[\texttt{phone} \mapsto \texttt{""}])$ 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 $H(2, [\mathsf{NFS}(5)])$. 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.

III. 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 [6]. 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 life-cycle 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. For space reasons, the presentation focuses on a relatively high-level overview of our extensions: the formal details, including the full operational semantics, are provided in Appendix **??**.

A. 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 V a few basic syntactic categories. A program P is a sequence of classes. A class cls $c \leq c'$ imp c^* $\{fld^*; mtd^*\}$ 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 $prim$.

$$
\begin{array}{lll}
P & ::= & cls^* \\
cls & ::= & \text{cls } c \leq c' \text{ imp } c^* \ \{fld^*; mtd^*\} \\
\tau_{prim} & ::= & \text{bool} \mid \text{int} \mid \ldots \\
\tau & ::= & c \mid \tau_{prim} \mid \text{array}[\tau] \\
fld & ::= & f : \tau \\
mtd & ::= & m : \tau^* \xrightarrow{n} \tau \ \{st^*\} \\
lhs & ::= & r \mid r[r] \mid r.f \mid c.f \\
prim & ::= & true \mid false \mid \ldots \\
rhs & ::= & lhs \mid prim
\end{array}
$$

TABLE V
BASIC SYNTACTIC CATEGORIES

Table VI 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 [6] 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.

B. Local Reduction

a) Notation: Table VII 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 IV. 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 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 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.

b) Local Reduction Relation: The *local reduction* relation $\Sigma \leadsto \Sigma'$ 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 simplest rule defining a local reduction step $\Sigma \leadsto \Sigma'$ just fetches the next statement st to run and performs a look-up

$st ::=$

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$_\oslash\ r_1\ r_2$ then pc	jump to program counter pc if $r_1 \oslash r_2$	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 VI
SYNTAX AND INFORMAL SEMANTICS OF SELECTED STATEMENTS

Pointers	p	$\in$	*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	$::=$	$L^\# \mid \text{AbNormal}(L^\#)$
Objects	o	$::=$	$\{\!\lvert c; (f_\tau \mapsto v)^* \rvert\!\}$
Arrays	a	$::=$	$\tau[v^*]$
Intents	i	$::=$	$\{\!\lvert @c; (k \mapsto v)^* \rvert\!\}$
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 VII
SEMANTIC DOMAINS FOR LOCAL REDUCTION

on the auxiliary relation $\Sigma, st \Downarrow \Sigma'$. Formally, assuming a function *get-stm*(Σ) fetching the next statement based on the program counter of the top-most local state in Σ, we have:

$$\text{(R-NEXTSTM)} \quad \frac{\Sigma, \textit{get-stm}(\Sigma) \Downarrow \Sigma'}{\Sigma \rightsquigarrow \Sigma'}$$

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

c) 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 $\text{ExcptTable}(pp, c) = pc$ which provides, for all program points pp where exceptions can be thrown and for all classes c extending the Throwable interface, the program counter pc of the corresponding exception handler. If no handler exists, then $\text{ExcptTable}(pp, c) = \bot$. 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 an 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)

d) Thread Rules: Our formalism covers the core methods of the Java Thread API [18]: 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 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.

C. Global Reduction

a) Notation: Table IX 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 life-cycle (see [31]). Then we have two kinds of *frames*, modeling 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

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

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

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

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

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

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

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

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

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 $\alpha_A = \texttt{AbNormal}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle :: \alpha')$.

TABLE VIII
SMALL STEP SEMANTICS OF EXTENDED μ-DALVIK$_A$ - EXCERPT

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 IX
SEMANTIC DOMAINS FOR GLOBAL REDUCTION

b) 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 life-cycle 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' 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 1 *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 are taken from [6], extended with a few simple rules used, e.g., to manage the thread pool. The main new rules are given in Table X and the full set can be found in Appendix **??**. 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.

IV. 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 [28] 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.

(A-ThreadStart)

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

(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)

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

TABLE X
New Global Reduction Rules - Excerpt

A. 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 XI 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 XI
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 $\hat{i}$ 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 $\hat{i}$. 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 $\bot$, representing the lack of a flow-sensitive abstraction of the memory blocks created at pp.

There is just one syntactic element in Table XI 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 IV-C.

B. Ingredients of the Analysis

a) 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, $\hat{h}(pp) = \bot$ 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.

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
$\tau \le \tau'$	Subtyping fact

TABLE XII
ANALYSIS FACTS

For space reasons, we just present selected excerpts of the analysis in the remaining of this section: the full analysis specification is given in Appendix **??**.

b) Analysis Facts: The syntax of the analysis *facts* f is defined in Table XII. 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.

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 of 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 $\tau \le \tau'$.

c) Horn Clauses: We define *Horn clauses* as logical formulas of the form $\forall x_1, \dots, \forall x_m.\mathsf{f}_1 \wedge \dots \wedge \mathsf{f}_n \implies \mathsf{f}$ 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 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 $\forall x_1, \dots, \forall x_m.\mathsf{f}_1 \wedge \dots \wedge \mathsf{f}_n \implies \mathsf{f}'_1 \wedge \dots \wedge \mathsf{f}'_k$ for the set $\{\forall x_1, \dots, \forall x_m.\mathsf{f}_1 \wedge \dots \wedge \mathsf{f}_n \implies \mathsf{f}'_i \mid i \in [1, k]\}$.

d) 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.

C. 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.

a) 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 XIII 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 $\hat{k} \sqcup \hat{k}'$, computing the point-wise maximum between $\hat{k}$ and $\hat{k}'$.

b) 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:

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

c) 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:

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

$$\text{Reach}(\widehat{prim}; \hat{h}; 0^*) \qquad \text{Reach}(\text{NFS}(\lambda); \hat{h}; 0^*) \qquad \text{Reach}(\text{FS}(\text{pp}); \hat{h}; 0^*[\text{pp} \mapsto 1]) \qquad \text{Reach}(\hat{u}; \hat{h}; \hat{k}) \wedge \hat{u} \sqsubseteq \hat{v} \implies \text{Reach}(\hat{v}; \hat{h}; \hat{k})$$

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

TABLE XIII
HORN CLAUSES USED TO DERIVE THE PREDICATE Reach$(\hat{v}; \hat{h}; \hat{k})$

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

$$\text{lift}(\text{NFS}(\lambda); \hat{k}) = \text{NFS}(\lambda) \qquad \text{lift}(\widehat{prim}; \hat{k}) = \widehat{prim}$$

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

TABLE XIV
AXIOMS REQUIRED ON THE FUNCTION lift$(\hat{v}^*; \hat{k})$

for lifted allocation sites. Formally, we require the lift function to satisfy the axioms in Table XIV;

2) the current abstract flow-sensitive heap $\hat{h}$. This is done by the function $\text{hlift}(\hat{h}; \hat{k})$, which replaces all the entries of the form $pp \mapsto \hat{b}$ in $\hat{h}$ with $pp \mapsto \bot$ if $\hat{k}(pp) = 1$, thus invalidating their flow-sensitive abstraction. If $\hat{k}(pp) = 0$, instead, the function calls $\text{lift}(\hat{v}; \hat{k})$ 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 $\hat{k} \sqcup \hat{k}'$, 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 $\text{FS}(pp) \sqsubseteq \hat{v}$.

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

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

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:

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

D. Abstracting Local Reduction

a) 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 $\text{GetBlk}_i(\hat{v}^*; \hat{h}; \hat{\lambda}; \hat{b})$ 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 $\hat{\lambda}$:

$$\text{FS}(\lambda) \sqsubseteq \hat{v}_i \wedge \hat{h}(\lambda) = \hat{b} \implies \text{GetBlk}_i(\hat{v}^*; \hat{h}; \text{FS}(\lambda); \hat{b})$$
$$\text{NFS}(\lambda) \sqsubseteq \hat{v}_i \wedge \text{H}(\lambda, \hat{b}) \implies \text{GetBlk}_i(\hat{v}^*; \hat{h}; \text{NFS}(\lambda); \hat{b})$$

b) 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:

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

c) 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 XV, presenting the abstract semantics of some statements considered in the original HornDroid paper [6]. We focus in particular on the main additions needed to generalize their abstraction to implement a flow-sensitive heap analysis:

- $(\!|\text{new } r_d \ c'|\!)_{pp}$: 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 IV-C. 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 $\text{FS}(pp)$ to enable a flow-sensitive analysis of the new most-recently-allocated object;

- $(\!|\text{move } r_o.f \ rhs|\!)_{pp}$: 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

- $(\!|\text{new } r_d\ c'|\!)_{c,m,pc} =$
  $\{\text{LState}_{c,m,pc}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \text{Reach}(\text{FS}(c, m, pc); \hat{h}; \hat{k}')$
  $\implies \text{LiftHeap}(\hat{h}; \hat{k}') \wedge \text{LState}_{c,m,pc+1}(\_; \text{lift}(\hat{v}^*; \hat{k}')[d \mapsto \text{FS}(c, m, pc)]; \text{hlift}(\hat{h}; \hat{k}')[c, m, pc \mapsto \{|c'; (f \mapsto \hat{\mathbf{0}}_\tau)^*|\}]; \hat{k} \sqcup \hat{k}')\}$
- $(\!|\text{move } r_o.f\ rhs|\!)_{c,m,pc} =$
  $\langle\!\langle rhs \rangle\!\rangle_{c,m,pc} \cup \{\text{RHS}_{c,m,pc}(\hat{v}'') \wedge \text{LState}_{c,m,pc}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \text{GetBlk}_o(\hat{v}^*; \hat{h}; \text{FS}(\lambda); \{|c'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'|\}) \implies$
  $\text{LState}_{c,m,pc+1}(\_; \hat{v}^*; \hat{h}[\lambda \mapsto \{|c'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}''|\}; \hat{k})\} \cup$
  $\{\text{RHS}_{c,m,pc}(\hat{v}'') \wedge \text{LState}_{c,m,pc}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \text{GetBlk}_o(\hat{v}^*; \hat{h}; \text{NFS}(\lambda); \{|c'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'|\}) \wedge \text{Reach}(\hat{v}''; \hat{h}; \hat{k}') \implies$
  $\text{H}(\lambda, \{|c'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}''|\}) \wedge \text{LiftHeap}(\hat{h}; \hat{k}') \wedge \text{LState}_{c,m,pc+1}(\_; \text{lift}(\hat{v}^*; \hat{k}'); \text{hlift}(\hat{h}; \hat{k}'); \hat{k} \sqcup \hat{k}')\}$
- $(\!|\text{return}|\!)_{c,m,pc} = \{\text{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{v}^*_{call}); \hat{v}^*; \hat{h}; \hat{k}) \implies \text{Res}_{c,m}((\hat{\lambda}_t, \hat{v}^*_{call}); \hat{v}_{res}; \hat{h}; \hat{k})\}$
- $(\!|\text{invoke } r_o\ m'\ (r_{i_j})^{j \leq n}|\!)_{c,m,pc} =$
  $\{\text{LState}_{c,m,pc}((\hat{\lambda}_t, \_); \hat{v}^*; \hat{h}; \hat{k}) \wedge \text{GetBlk}_o(\hat{v}^*; \hat{h}; \_; \{|c'; (f \mapsto \hat{u})^*|\}) \wedge c' \leq c'' \implies$
  $\text{LState}_{c'',m',0}((\hat{\lambda}_t, (\hat{v}_{i_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, (\hat{v}_{i_j})^{j \leq n}; \hat{h}; 0^*) \mid c'' \in \widehat{lookup}(m') \wedge sign(c'', m') = (\tau_j)^{j \leq n} \xrightarrow{loc} \tau\} \cup$ (**1**)
  $\{\text{LState}_{c,m,pc}((\hat{\lambda}_t, \_); \hat{v}^*; \hat{h}; \hat{k}) \wedge \text{GetBlk}_o(\hat{v}^*; \hat{h}; \_; \{|c'; (f \mapsto \hat{u})^*|\}) \wedge c' \leq c'' \wedge \text{Res}_{c'',m'}((\hat{\lambda}'_t, \hat{w}^*); \hat{v}'_{res}; \hat{h}_{res}; \hat{k}_{res})$
  $\wedge \hat{\lambda}_t = \hat{\lambda}'_t \wedge \left(\bigwedge_{j \leq n} \hat{v}_{i_j} \sqcap \hat{w}_j \not\sqsubseteq \bot\right) \implies \text{LState}_{c,m,pc+1}((\hat{\lambda}_t, \_); \text{lift}(\hat{v}^*; \hat{k}_{res})[res \mapsto \hat{v}'_{res}]; \hat{h}_{res}; \hat{k} \sqcup \hat{k}_{res}) \mid c'' \in \widehat{lookup}(m')\} \cup$ (**2**)
  $\{\text{LState}_{c,m,pc}((\hat{\lambda}_t, \_); \hat{v}^*; \hat{h}; \hat{k}) \wedge \text{GetBlk}_o(\hat{v}^*; \hat{h}; \_; \{|c'; (f \mapsto \hat{u})^*|\}) \wedge c' \leq c'' \wedge \text{Uncaught}_{c'',m'}((\hat{\lambda}'_t, \hat{w}^*)); \hat{v}'_{excpt}; \hat{h}_{res}; \hat{k}_{res})$
  $\wedge \hat{\lambda}_t = \hat{\lambda}'_t \wedge \left(\bigwedge_{j \leq n} \hat{v}_{i_j} \sqcap \hat{w}_j \not\sqsubseteq \bot\right) \implies \text{AState}_{c,m,pc}((\hat{\lambda}_t, \_); \text{lift}(\hat{v}^*; \hat{k}_{res})[excpt \mapsto \hat{v}'_{excpt}]; \hat{h}_{res}; \hat{k} \sqcup \hat{k}_{res}) \mid c'' \in \widehat{lookup}(m')\}$ (**3**)
- $(\!|\text{throw } r_i|\!)_{c,m,pc} = \{\text{LState}_{c,m,pc}(\_; \hat{v}^*; \hat{h}; \hat{k}) \implies \text{AState}_{c,m,pc}(\_; \hat{v}^*[excpt \mapsto \hat{v}_i]; \hat{h}; \hat{k})\}$
- $(\!|\text{start-thread } r_i|\!)_{c,m,pc} =$
  $\{\text{LState}_{c,m,pc}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \text{GetBlk}_i(\hat{v}^*; \hat{h}; \text{NFS}(\lambda); \{|c'; (f \mapsto \hat{u})^*|\}) \wedge c' \leq \text{Thread}$
  $\implies \text{T}(\lambda, \{|c'; (f \mapsto \hat{u})^*|\}) \wedge \text{LState}_{c,m,pc+1}(\_; \hat{v}^*; \hat{h}; \hat{k})\} \cup$
  $\{\text{LState}_{c,m,pc}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \text{GetBlk}_i(\hat{v}^*; \hat{h}; \text{FS}(\lambda); \{|c'; (f \mapsto \hat{u})^*|\}) \wedge c' \leq \text{Thread} \wedge \text{Reach}(\text{FS}(\lambda); \hat{h}; \hat{k}')$
  $\implies \text{T}(\lambda, \{|c'; (f \mapsto \hat{u})^*|\}) \wedge \text{LiftHeap}(\hat{h}; \hat{k}') \wedge \text{LState}_{c,m,pc+1}(\_; \text{lift}(\hat{v}^*; \hat{k}'); \text{hlift}(\hat{h}; \hat{k}'); \hat{k} \sqcup \hat{k}')\}$
- $(\!|\text{join } r_i|\!)_{c,m,pc} =$
  $\{\text{LState}_{c,m,pc}((\text{NFS}(\lambda_t), \_); \hat{v}^*; \hat{h}; \hat{k}) \wedge \text{H}(\lambda_t, \{|c'; (f \mapsto \hat{u})^*, \text{inte} \mapsto \hat{v}'|\}) \wedge \widehat{false} \sqsubseteq \hat{v}' \implies \text{LState}_{c,m,pc+1}((\text{NFS}(\lambda_t), \_); \hat{v}^*; \hat{h}; \hat{k})\} \cup$
  $\{\text{LState}_{c,m,pc}((\text{NFS}(\lambda_t), \_); \hat{v}^*; \hat{h}; \hat{k}) \wedge \text{H}(\lambda_t, \{|c'; (f \mapsto \hat{u})^*, \text{inte} \mapsto \hat{v}'|\}) \wedge \widehat{true} \sqsubseteq \hat{v}' \implies$
  $\text{H}(c, m, pc; \{|\text{IntExcpt}; |\}) \wedge \text{AState}_{c,m,pc}((\text{NFS}(\lambda_t), \_); \hat{v}^*[excpt \mapsto \text{NFS}(c, m, pc)]; \hat{h}; \hat{k}) \wedge \text{H}(\lambda_t, \{|c'; (f \mapsto \hat{u})^*, \text{inte} \mapsto \widehat{false}|\})\}$

TABLE XV
ABSTRACT SEMANTICS OF STATEMENTS - EXCERPT

the content of the register r_o. If GetBlk_o returns an abstract flow-sensitive location $\text{FS}(\lambda)$, 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 $\text{NFS}(\lambda)$, we use λ to get an abstract heap fact $\text{H}(\lambda, \{|c'; (f' \mapsto \hat{u}')^*, f \mapsto \hat{v}'|\})$ 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;

- $(\!|\text{return}|\!)_{pp}$: The callee generates a return fact Res containing the calling context $(\hat{\lambda}_t, \hat{v}^*_{call})$, the abstract value $\hat{v}_{res}$ 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;
- $(\!|\text{invoke } r_o\ m'\ (r_{i_j})^{j \leq n}|\!)_{pp}$: 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 $\bot$. This prevents a callee from returning to a caller that could not have invoked it, in case (i) because caller and callee are being executed by different threads, 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 $\hat{h}_{res}$, lifting the abstraction of the caller registers, joining the caller abstract filter $\hat{k}$ with the callee abstract filter $\hat{k}_{res}$, and storing the abstraction of the returned value $\hat{v}'_{res}$ 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 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 Uncaught fact with the abstract local state of the caller.

d) Exceptions and Threads: The bottom part of Table XV 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 $\textsf{FS}(\lambda)$, 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 $\widehat{true}$, in which case an abstract abnormal local state throwing an IntExcpt exception is generated, or $\widehat{false}$, in which case the abstract local state is propagated to the next program point.

e) Example: We show in Table XVI a (simplified) byte-code program corresponding to the code snippet in Table I. 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 $\textsf{FS}(7)$ 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:

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

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

$$7 \mapsto \{[\texttt{Storage}; \texttt{s} \mapsto ""]\}, 9 \mapsto \textsf{hlift}(\hat{l}_2; 7 \mapsto bb'_1, 9 \mapsto bb'_2).$$

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.

E. 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 life-cycle. We extended the original rules of HornDroid [6] with some new rules needed to support our richer concrete semantics including threads and exceptions. Table XVII shows two of these rules to exemplify, the other rules are in Appendix **??**. 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 ($\mathbf{A}$) 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 ($\mathbf{B}$) is triggered if no such entry exists, and generates an abstract uncaught exception fact, 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 $\textsf{GetBlk}_i$, plus the Horn clauses abstracting system events and the activity life-cycle. 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.

F. Formal Results

The soundness of the analysis is proved by using *representation functions* [29]: 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 Appendix **??**.

Theorem 1 (Global Preservation) *If* $\Psi \Rightarrow^* \Psi'$ *under a given program P, then for any* $\Delta_1 \in \beta_{Cnf}(\Psi)$ *and* $\Delta_2 :> \Delta_1$ *there exist* $\Delta'_1 \in \beta_{Cnf}(\Psi')$ *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:

Taint	t	$::=$	public $\mid$ secret
Values	u, v	$::=$	$prim^t \mid \ell$

The set of taints is a two-valued lattice, and we use $\sqsubseteq^t$ and $\sqcup^t$ to denote respectively the standard ordering on taints (where public $\sqsubseteq^t$ 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.

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:

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

Generated Horn Clauses for Line 18:

- $\mathsf{LState}_{18}((\hat{\lambda}_t, \_); r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto bb_1, 9 \mapsto bb_2) \wedge$
  $\mathsf{GetBlk}_1(r_0 \mapsto \hat{u}, r_1 \mapsto \hat{v}, \mathtt{ret} \mapsto \hat{w}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; \_; \{\!|c'; \_|\!\}) \wedge c' \le \mathtt{Leaky} \implies$
  $\mathsf{LState}_0((\hat{\lambda}_t, \hat{v}); r_0 \mapsto \hat{v}; 7 \mapsto \hat{l}_1, 9 \mapsto \hat{l}_2; 7 \mapsto 0, 9 \mapsto 0)$

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

Generated Horn Clauses for Line 20:

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

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

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

TABLE XVI

EXAMPLE OF DALVIK BYTECODE AND EXCERPT OF THE CORRESPONDING HORN CLAUSES

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

TABLE XVII

GLOBAL RULES OF THE ABSTRACT SEMANTICS - EXCERPT

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 **??** 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 predicate need to perform a fix-point computation in the abstract heap.

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.

$$\mathsf{Taint}(\widehat{prim^t}, \hat{h}, t) \qquad \mathsf{Taint}(\hat{u}, \hat{h}, \hat{t}) \wedge \hat{u} \sqsubseteq \hat{v} \implies \mathsf{Taint}(\hat{v}, \hat{h}, \hat{t}) \qquad \mathsf{Taint}(\hat{v}, \hat{h}, \hat{t}) \wedge \mathsf{Taint}(\hat{v}, \hat{h}, \hat{t}') \implies \mathsf{Taint}(\hat{v}, \hat{h}, \hat{t} \sqcup^{\mathsf{t}} \hat{t}')$$

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

TABLE XVIII
HORN CLAUSES RULES USED TO DERIVE $\mathsf{Taint}(\hat{v}, \hat{h}, \hat{t})$.

Definition 2 *A program P leaks* starting from a configuration Ψ *if there exists* $(c, m) \in Sinks$ *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) = $ 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 **??**.

Lemma 1 *If for all sinks* $(c, m) \in Sinks$, $\Delta \in \beta_{Cnf}(\Psi)$:

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

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

V. EXPERIMENTS

We implemented a prototype of our flow-sensitive analysis as an extension of an existing taint tracker, HornDroid [6]. 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 [28] 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 [33]; if no query is satisfiable according to Z3, no information leak may occur by the soundness results of our analysis.

A. Testing on DroidBench

We tested our flow-sensitive extension of HornDroid (called fsHornDroid) against DroidBench [3], 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 [3], AmanDroid [40], DroidSafe [14] and the original version of HornDroid [6]. 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 XVIII.

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 contains also 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.

B. 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 [39]. 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, which we plan to manually investigate in the future. 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 the 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 constantly updated experimental evaluation is available online, along with the web version of the tool and its sources [1]. Our results demonstrate that fsHornDroid scales to real applications, despite the increased performance overhead with respect to the original HornDroid.

C. 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 [14] and we think their research may be very helpful

---

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

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-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 XIX
VALIDITY MEASURES AND ANALYSIS TIMES ON DROIDBENCH

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.

VI. RELATED WORK

There are several static information flow analysers for Android applications (see, e.g., [41], [42], [27], [13], [22], [3], [40], [14], [6]). We thoroughly compared with the current state of the art in the rest of the paper, so we focus here on other related works.

a) Sound Analysis of Android Applications: The first paper proposing a formally sound static analysis of Android applications is a seminal work by Chaudhuri [7]. 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 [12]. Unfortunately, SCanDroid is in an early prototype phase and it cannot analyse the applications in DroidBench [3].

Sound type systems for Android applications have also been proposed in [25] to prove non-interference and in [5] 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 [25] 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 [30]. 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.

b) 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 [20], [37], [36]. The most prominent techniques in pointer analysis are variants of the classical Andersen algorithm [2], including flow-insensitive analyses [9], [32], [16], [21] and flow-sensitive analyses [8], [10], [19], [23]; light-weight analyses in the flavor of the unification-based Steensgaard analysis [38], which are flow-insensitive and very efficient; and shape analysis techniques [35], 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 [34], 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 [34] 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.* [15] 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.

VII. CONCLUSION

We presented 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. Designing a sound yet precise analysis in this setting is particularly challenging, due to the complexity of the control flow of Android applications. In this work, we adapted ideas from *recency abstraction* [4] to hit a sweet spot in the analysis design space: our proposal is sound, precise, and efficient in practice. We substantiated these claims by

implementing the analysis in HornDroid [6], a state-of-the-art static information flow analyser for Android applications, and by performing an experimental evaluation of our extension. Our work takes HornDroid one step further towards the sound information flow analysis of real Android applications.

REFERENCES

[1] secpriv.tuwien.ac.at/tools/horndroid, website of fsHornDroid

[2] Andersen, L.O.: Program analysis and specialization for the C programming language. Tech. rep., University of Copenhagen (1994)

[3] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.: FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: PLDI. pp. 259–269. ACM (2014)

[4] Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: SAS. pp. 221–239. Springer-Verlag (2006)

[5] Bugliesi, M., Calzavara, S., Spanò, A.: Lintent: Towards security type-checking of Android applications. In: FMOODS/FORTE. pp. 289–304 (2013)

[6] Calzavara, S., Grishchenko, I., Maffei, M.: HornDroid: Practical and sound static analysis of Android applications by SMT solving. In: EuroS&P. IEEE (2016)

[7] Chaudhuri, A.: Language-based security on Android. In: PLAS. pp. 1–7. ACM (2009)

[8] Choi, J.D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: POPL. pp. 232–245. ACM (1993)

[9] Das, M.: Unification-based pointer analysis with directional assignments. SIGPLAN Not. 35(5), 35–46 (May 2000)

[10] Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. SIGPLAN Not. 29(6), 242–256 (Jun 1994)

[11] Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: USENIX Security Symposium (2011)

[12] Fuchs, A.P., Chaudhuri, A., , Foster, J.S.: Scandroid: Automated security certification of Android applications. Tech. rep., University of Maryland (2009)

[13] Gibler, C., Crussell, J., Erickson, J., Chen, H.: Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In: TRUST. pp. 291–307. Springer-Verlag (2012)

[14] Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of Android applications in DroidSafe. In: NDSS. IEEE (2015)

[15] Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI. pp. 266–277. ACM (2007)

[16] Hardekopf, B., Lin, C.: The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. SIGPLAN Not. 42(6), 290–299 (Jun 2007)

[17] Java 8 Documentation on Object. https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html, last accessed on February 2017

[18] Java 8 Documentation on Thread. https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html, last accessed on February 2017

[19] Kahlon, V.: Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. SIGPLAN Not. 43(6), 249–259 (Jun 2008)

[20] Kanvar, V., Khedker, U.P.: Heap abstractions for static analysis. CoRR abs/1403.4910 (2014), http://arxiv.org/abs/1403.4910

[21] Kastrinis, G., Smaragdakis, Y.: Hybrid context-sensitivity for points-to analysis. SIGPLAN Not. 48(6), 423–434 (Jun 2013)

[22] Kim, J., Yoon, Y., Yi, K., Shin, J., Center, S.: Scandal: Static analyzer for detecting privacy leaks in Android applications. In: MoST (2012)

[23] Lhoták, O., Chung, K.C.A.: Points-to analysis with efficient strong updates. SIGPLAN Not. 46(1), 3–16 (Jan 2011)

[24] Lochbihler, A.: Making the java memory model safe. ACM Trans. Program. Lang. Syst. 35(4), 12:1–12:65 (Jan 2014), http://doi.acm.org/10.1145/2518191

[25] Lortz, S., Mantel, H., Starostin, A., Bähr, T., Schneider, D., Weber, A.: Cassandra: Towards a certifying app store for Android. In: SPSM@CCS. pp. 93–104. ACM (2014)

[26] Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In: CCS. pp. 229–240. ACM (2012)

[27] Mann, C., Starostin, A.: A framework for static detection of privacy leaks in Android applications. In: SAC. pp. 1457–1462. ACM (2012)

[28] de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340. Springer-Verlag (2008)

[29] Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer-Verlag (1999)

[30] Payet, É., Spoto, F.: Static analysis of Android programs. Information & Software Technology 54(11), 1192–1201 (2012)

[31] Payet, É., Spoto, F.: An operational semantics for Android activities. In: PEPM. pp. 121–132. ACM (2014)

[32] Pereira, F.M.Q., Berlin, D.: Wave propagation and deep propagation for pointer analysis. In: GCO. pp. 126–135 (2009)

[33] Rasthofer, S., Arzt, S., Bodden, E.: A machine-learning approach for classifying and categorizing Android sources and sinks. In: NDSS (2014)

[34] Rugina, R., Rinard, M.: Pointer analysis for multithreaded programs. SIGPLAN Not. 34(5), 77–90 (May 1999)

[35] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL. pp. 105–118. ACM (1999)

[36] Smaragdakis, Y., Balatsouras, G.: Pointer analysis. Found. Trends Program. Lang. 2(1), 1–69 (Apr 2015)

[37] Sridharan, M., Chandra, S., Dolby, J., Fink, S.J., Yahav, E.: Alias analysis for object-oriented programs. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming, pp. 196–232. Springer-Verlag, Berlin, Heidelberg (2013), http://dl.acm.org/citation.cfm?id=2554511.2554523

[38] Steensgaard, B.: Points-to analysis in almost linear time. In: POPL. pp. 32–41. ACM (1996)

[39] The Collection of Android Apps and Metadata. https://archive.org/details/android_apps&tab=about, last accessed on February 2017

[40] Wei, F., Roy, S., Ou, X., Robby: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In: CCS. pp. 1329–1341. ACM (2014)

[41] Yang, Z., Yang, M.: Leakminer: Detect information leakage on Android with static taint analysis. In: WCSE. pp. 101–104. IEEE (2012)

[42] Zhao, Z., Osorio, F.C.C.: Trustdroid: Preventing the use of smartphones for information leaking in corporate networks through the use of static analysis taint tracking. In: MALWARE. pp. 135–143. IEEE (2012)

a) Appendix outline:: In Section **??** 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 **??** we give the full abstract semantics; in Section **??** we give the soundness proof.

APPENDIX A
CONCRETE SEMANTICS

As in [6], we require that Dalvik programs are *well-formed*.

Definition 3 (Well-formed Program [6]) *A program P is* well-formed *iff all its class names are pairwise distinct and, for each of its classes, all the field names and the method names are pairwise distinct.*

From now on, we always consider a fixed well-formed program $P = cls^*$. We give in Table **??** the syntax and an informal explanation of the Dalvik statements that were omitted in the body. The extensions with respect to [6] are in bold.

A. Extensions : Waiting Sets and Monitors

In order to give a full account of Java concurrency we extended our model to include waiting sets and monitors [17], 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).

$$
\begin{array}{llll}
\text{Waiting states} & \omega & ::= & \mathsf{waiting}(\ell, j) \\
\text{Local state lists} & L^\# & ::= & \varepsilon \mid L :: L^\# \mid \omega :: L^\#
\end{array}
$$

a) 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 the monitor is not held by another thread. If a thread

`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 XX
SYNTAX AND INFORMAL SEMANTICS OF ADDITIONAL STATEMENTS

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.

B. Local Reduction Relation

1) 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 $type_H(v)$ that retrieve from the heap H the type of the memory block v is pointing to.

Definition 4 *Given a heap H, we let the partial function $type_H(v)$ be defined as follows:*

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

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

Given a class name c, we let $super(c) = c'$ if there exists a class cls_i such that $cls_i = \mathtt{cls}\ c \leq c'\ \mathtt{imp}\ c^*\ \{fld^*; mtd^*\}$, and $inter(c) = \{c^*\}$ iff there exists a class cls_i such that $cls_i = \mathtt{cls}\ c \leq c'\ \mathtt{imp}\ c^*\ \{fld^*; mtd^*\}$. The subtyping relation is quite simple: a class c is a subclass of its super class $super(c)$ and of the interfaces $inter(c)$ 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 run-time to avoid errors). The typing rules are summarized below.

$$
\text{(SUB-REFL)} \quad \frac{}{\tau \leq \tau}
\qquad
\text{(SUB-TRANS)} \quad \frac{\tau \leq \tau' \quad \tau' \leq \tau''}{\tau \leq \tau''}
\qquad
\text{(SUB-EXT)} \quad \frac{}{c \leq super(c)}
\qquad
\text{(SUB-IMPL)} \quad \frac{c' \in inter(c)}{c \leq c'}
\qquad
\text{(SUB-ARRAY)} \quad \frac{\tau \leq \tau'}{array[\tau] \leq array[\tau']}
$$

2) 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 **??** the relation $\Sigma[\![rhs]\!]$ that evaluates a right-hand side expression in a given local configuration Σ.

$$
\text{(RHS-REGISTER)} \quad \frac{}{\Sigma[\![r]\!] = R(r)}
\qquad
\text{(RHS-ARRAY)} \quad \frac{\ell = \Sigma[\![r_a]\!] \quad a = H(\ell) \quad j = \Sigma[\![r_{idx}]\!]}{\Sigma[\![r_a[r_{idx}]]\!] = a[j]}
\qquad
\text{(RHS-OBJECT)} \quad \frac{\ell = \Sigma[\![r_o]\!] \quad o = H(\ell)}{\Sigma[\![r_o.f]\!] = o.f}
\qquad
\text{(RHS-STATIC)} \quad \frac{}{\Sigma[\![c.f]\!] = S(c.f)}
\qquad
\text{(RHS-PRIM)} \quad \frac{}{\Sigma[\![prim]\!] = prim}
$$

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 $\alpha_c = \mathtt{AbNormal}(\langle pp \cdot \_ \cdot st^* \cdot R \rangle :: \alpha')$.

TABLE XXI
EVALUATION OF RIGHT-HAND SIDES ($\Sigma[\![rhs]\!] = v$)

3) 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:

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

We are finally ready to give the semantics of the Dalvik bytecode relation: the standard operation are in Table **??**, while the new operations are given in Table **??**

(R-GOTO)

$$\overline{\Sigma, \texttt{goto } pc' \Downarrow \Sigma[pc \mapsto pc']}$$

(R-TRUE)

$$\frac{\Sigma[\![r_1]\!] \oslash \Sigma[\![r_2]\!]}{\Sigma, \texttt{if}_\oslash\ r_1\ r_2 \texttt{ then } pc' \Downarrow \Sigma[pc \mapsto pc']}$$

(R-FALSE)

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

(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)

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

(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)

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

(R-NEWARR)

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

(R-CAST)

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

(R-INSTOFTRUE)

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

(R-INSTOFFALSE)

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

(R-RETURN)

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

(R-SCALL)

$$\frac{\begin{array}{c} lookup(c', m') = (c', st^*) \qquad sign(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau \\ R' = ((r_j \mapsto \mathbf{0})^{j \leq loc}, (r_{loc+k} \mapsto \Sigma[\![r_k']\!])^{k \leq n}) \\ \alpha'' = \langle c', m', 0 \cdot (\Sigma[\![r_k']\!])^{k \leq n} \cdot st^* \cdot R' \rangle :: \alpha \end{array}}{\Sigma, \texttt{sinvoke } c'\ m'\ r_1', \ldots, r_n' \Downarrow \Sigma[\alpha \mapsto \alpha'']}$$

(R-CALL)

$$\frac{\begin{array}{ccc} \ell = \Sigma[\![r_o]\!] & lookup(type_H(\ell), m') = (c', st^*) & sign(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau \\ R' = ((r_j \mapsto \mathbf{0})^{j \leq loc}, r_{loc+1} \mapsto \ell, (r_{loc+1+k} \mapsto \Sigma[\![r_k']\!])^{k \leq n}) & \alpha'' = \langle c', m', 0 \cdot (\Sigma[\![r_k']\!])^{k \leq n} \cdot st^* \cdot R' \rangle :: \alpha \end{array}}{\Sigma, \texttt{invoke } r_o\ m'\ r_1', \ldots, r_n' \Downarrow \Sigma[\alpha \mapsto \alpha'']}$$

(R-NEWINTENT)

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

(R-PUTEXTRA)

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

(R-GETEXTRA)

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

(R-STARTACT)

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

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 XXII

SMALL STEP SEMANTICS OF μ-DALVIK$_A$ - STANDARD STATEMENTS

Exception Rules

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

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

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

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

Thread Rules

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

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

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

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

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

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

Monitor and Wait Rules

(R-MONITORENTER1)
$$\frac{\ell = \Sigma[\![r_i]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \mathsf{acquired} \mapsto \_, \mathsf{m\text{-}cnt} \mapsto 0|\!\}}{o' = \{\!|c'; (f \mapsto v)^*, \mathsf{acquired} \mapsto \ell_r, \mathsf{m\text{-}cnt} \mapsto 1|\!\}}{\Sigma, \texttt{monitor-enter } r_i \Downarrow \Sigma^+[H \mapsto H[\ell \mapsto o']]}$$

(R-MONITORENTER2)
$$\frac{\ell = \Sigma[\![r_i]\!] \qquad H(\ell) = \{\!|c'; (f \mapsto v)^*, \mathsf{acquired} \mapsto \ell_r, \mathsf{m\text{-}cnt} \mapsto j|\!\}}{o' = \{\!|c'; (f \mapsto v)^*, \mathsf{acquired} \mapsto \ell_r, \mathsf{m\text{-}cnt} \mapsto j+1|\!\} \qquad j > 0}{\Sigma, \texttt{monitor-enter } r_i \Downarrow \Sigma^+[H \mapsto H[\ell \mapsto o']]}$$

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

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

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

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

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 XXIII

SMALL STEP SEMANTICS OF μ-DALVIK$_A$ - NEW STATEMENTS

C. Global Rules Descriptions

1) 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 exchanged 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 model serialization using a set of derivation rules for fact of the form $\Gamma \vdash ser_{Val}^H(v) = (v', H', \Gamma')$ and $\Gamma \vdash ser_{Blk}^H(b) = (b', H', \Gamma')$, where Γ and Γ' are serialization context consisting a of list of key-value bindings of locations of the form $(p_\lambda \mapsto p'_\lambda)$ (notice that both location have the same annotation). Serialization 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 $ser_{Val}^H(v) = (v', H', \Gamma')$ (resp. $\Gamma \vdash ser_{Blk}^H(b) = (b', H', \Gamma')$) 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 **??** for the formal statement of the serialization rules.

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

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

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

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

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

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

TABLE XXIV

SERIALIZATION RULES

2) Threads and Activities: Before giving the global reduction relation, we need some definitions. We start by formally define what is a thread class and an activity class.

Definition 5 *A class cls is a* thread class *if and only if* $cls = $ `cls` $c \leq c'$ `imp` c^* $\{fld^*; mtd^*\}$ *for some* $c' \leq$ 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 6 *A class cls is an* activity class *if and only if* $cls = $ `cls` $c \leq c'$ `imp` c^* $\{fld^*; mtd^*\}$ *for some* $c' \leq$ 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 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 $handlers(c) = \{m_1, \ldots, m_n\}$ be the set of callback method names of c. We model the activity life-cycle (see [31]) 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 an new activity frame, created upon a callback method invocation:

Definition 7 *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) = $*

$\tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ 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 $\tau_1, \ldots, \tau_n$.

3) Global Reduction Relation: We are now ready to give the global reduction relation. First we will describe two new rules which were not given in the body and can be found in Table **??**: 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.

(T-REDUCE)
$$\dfrac{\ell_t \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S \rightsquigarrow \ell_t \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S'}{\Omega \cdot \Xi :: \langle\!\langle \ell, \ell_t, \pi, \gamma, \alpha \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega \cdot \Xi :: \langle\!\langle \ell, \ell_t, \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H' \cdot S'}$$

(T-KILL)
$$\dfrac{H(\ell') = \{\!| c; (f \mapsto v)^*, \mathsf{finished} \mapsto \_ |\!\} \qquad H' = H[\ell' \mapsto \{\!| c; (f \mapsto v)^*, \mathsf{finished} \mapsto true |\!\}]}{\Omega \cdot \Xi :: \langle\!\langle \ell, \ell', \varepsilon, \varepsilon, \overline{\alpha} \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega \cdot \Xi :: \Xi' \cdot H' \cdot S}$$

(T-INTENT)
$$\dfrac{(\varphi, \varphi') \in \{(\langle \ell, s, \pi, \gamma, \alpha \rangle, \langle \ell, s, i :: \pi, \gamma, \alpha \rangle), (\underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}, \underline{\langle \ell, s, i :: \pi, \gamma, \alpha \rangle})\}}{\Omega :: \varphi :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', i :: \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega :: \varphi' :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S}$$

(T-THREAD)
$$\dfrac{(\varphi, \varphi') \in \{(\langle \ell, s, \pi, \gamma, \alpha \rangle, \langle \ell, s, \pi, \ell_t :: \gamma, \alpha \rangle), (\underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}, \underline{\langle \ell, s, \pi, \ell_t :: \gamma, \alpha \rangle})\}}{\Omega :: \varphi :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \gamma' :: \ell_t :: \gamma'', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega :: \varphi' :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \gamma' :: \gamma'', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S}$$

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

TABLE XXV
NEW GLOBAL REDUCTION RULES

Table **??** recalls the rules introduced by [6] to model the activity life-cycle 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 life-cycle, and executes a callback method from this new activity state, provided some side conditions related to the pending activity stack and the 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 (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 activity. 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)
$$\frac{\pi \neq \varepsilon \Rightarrow (s, s') = (\textit{running}, \textit{onPause}) \qquad H(\ell).\text{finished} = \textit{true} \Rightarrow (s, s') \in \{(\textit{running}, \textit{onPause}), (\textit{onPause}, \textit{onStop}), (\textit{onStop}, \textit{onDestroy})\}}{\langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \underline{\langle \ell, s', \pi, \gamma, \alpha_{\ell.s'} \rangle} :: \Omega \cdot \Xi \cdot H \cdot S}$$

with $(s, s') \in \textit{Lifecycle}$ above.

(A-DESTROY)
$$\frac{H(\ell).\text{finished} = \textit{true}}{\Omega :: \langle \ell, \textit{onDestroy}, \pi, \gamma, \overline{\alpha} \rangle :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \Omega' \cdot \Xi \cdot H \cdot S}$$

(A-BACK)
$$\frac{H' = H[\ell \mapsto H(\ell)[\text{finished} \mapsto \textit{true}]]}{\langle \ell, \textit{running}, \varepsilon, \gamma, \overline{\alpha} \rangle :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \langle \ell, \textit{running}, \varepsilon, \gamma, \overline{\alpha} \rangle :: \Omega \cdot \Xi \cdot H' \cdot S}$$

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

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

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

(A-SWAP)
$$\frac{\varphi' = \langle \ell', \textit{onPause}, \varepsilon, \gamma', \overline{\alpha}' \rangle \qquad H(\ell').\text{finished} = \textit{true} \qquad \varphi = \langle \ell, s, i :: \pi, \gamma, \overline{\alpha} \rangle \qquad s \in \{\textit{onPause}, \textit{onStop}\} \qquad H(\ell').\text{parent} = \ell}{\varphi' :: \varphi :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \varphi :: \varphi' :: \Omega \cdot \Xi \cdot H \cdot S}$$

(A-RESULT)
$$\frac{\varphi' = \langle \ell', \textit{onPause}, \varepsilon, \gamma', \overline{\alpha}' \rangle \qquad H(\ell').\text{finished} = \textit{true} \qquad \varphi = \langle \ell, s, \varepsilon, \gamma, \overline{\alpha} \rangle \qquad s \in \{\textit{onPause}, \textit{onStop}\} \qquad H(\ell').\text{parent} = \ell \qquad \emptyset \vdash ser_{Val}^H(H(\ell').\text{result}) = (w', H') \qquad H'' = (H, H')[\ell \mapsto H(\ell)[\text{result} \mapsto w']]}{\varphi' :: \varphi :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \underline{\langle \ell, s, \varepsilon, \gamma, \alpha_{\ell.\textit{onActivityResult}} \rangle} :: \varphi' :: \Omega \cdot \Xi \cdot H'' \cdot S}$$

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

TABLE XXVI

REDUCTION RULES FOR CONFIGURATIONS $(\Omega \cdot \Xi \cdot H \cdot S \Rightarrow \Omega' \cdot \Xi' \cdot H' \cdot S')$

APPENDIX B

ABSTRACT SEMANTICS

1) Lifting functions: We first give the formal definition of the hlift$(;)$ and $\hat{\sqcup}$ functions, that we informally described in the body of the paper.

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

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

2) Right-Hand Side: We can now present the rules for the abstract evaluation of right-hand sides (a formal description is given in Table **??**): 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 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* $\mathsf{lift}(\hat{v}; 1^*)$ [4]. *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.*

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

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

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

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

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

TABLE XXVII
ABSTRACT EVALUATION OF RIGHT-HAND SIDES

3) Activity Abstraction: We will now describe the rules abstracting the activity life-cycle and thread management mechanisms, which are given in Table **??**. 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 finished field of an activity object by $\top_{\mathsf{bool}}$. The rule (REP) restarts abstract activity objects at any time, by re-setting their fields to their default initial abstract value $\hat{0}_\tau$ (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 $\mathsf{I}_{c'}(\{\!|@in(c); \hat{v}^*|\!\})$ 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:

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

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 an 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.

$$
\begin{aligned}
\textit{Tstart} \quad = \quad & \{\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}\} \\
\textit{Cbk} \quad = \quad & \{\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 sign(\mathsf{c}', m) = \tau_1, \dots, \tau_n \xrightarrow{loc} \tau\} \\
\textit{Fin} \quad = \quad & \{\mathsf{H}(c, \{\!|c; (f \mapsto \_)^*, \mathsf{finished} \mapsto \_|\!\}) \implies \mathsf{H}(c, \{\!|c; (f \mapsto \_)^*, \mathsf{finished} \mapsto \top_{\mathsf{bool}}|\!\})\} \\
\textit{Rep} \quad = \quad & \{\mathsf{H}(c, \{\!|c; (f_\tau \mapsto \_)^*|\!\}) \implies \mathsf{H}(c, \{\!|c; (f_\tau \mapsto \hat{\mathbf{0}}_\tau)^*|\!\})\} \\
\textit{Act} \quad = \quad & \{\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{false}, \mathsf{parent} \mapsto \mathsf{c}', \mathsf{intent} \mapsto in(c)|\!\})\} \\
\textit{Res} \quad = \quad & \{\mathsf{H}(\mathsf{c}', \{\!|\mathsf{c}'; (f' \mapsto \_)^*, \mathsf{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}|\!\})\} \\
\textit{Sub} \quad = \quad & \{\tau \leq \tau' \mid \tau \leq \tau' \text{ is a valid subtyping judgment}\} \\
\textit{Po} \quad = \quad & \{\hat{v} \sqsubseteq \hat{v}' \mid \hat{v} \sqsubseteq \hat{v}' \text{ is a valid partial ordering}\}
\end{aligned}
$$

TABLE XXVIII
ABSTRACT SEMANTICS OF μ-DALVIK$_A$ - ACTIVITY RULES

4) Statement Abstraction: Before giving the abstract rule for Dalvik statements, we need to define the abstract counterpart of the $type_H(b)$ function:

Definition 8 *Given an abstract memory block $\hat{b}$, we define a function $\widehat{get\text{-}type}(\hat{b})$ as follows:*

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

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 the main body of this paper, and the full set of rules is given in Table **??**, Table **??** and Table **??**. A few points are worth mentioning:

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

$$(\!|\texttt{goto } pc'|\!)_{pp} \;=\; \{\mathsf{LState}_{pp}(\_;\hat{v}^*;\_;\_) \implies \mathsf{LState}_{c,m,pc'}(\_;\hat{v}^*;\_;\_)\}$$

$$(\!|\texttt{if}_{\oslash}\ r_i\ r_j\ \texttt{then } pc'|\!)_{pp} \;=\; \{\mathsf{LState}_{pp}(\_;\hat{v}^*;\_;\_) \wedge \hat{v}_i\ \hat{\oslash}\ \hat{v}_j \implies \mathsf{LState}_{c,m,pc'}(\_;\hat{v}^*;\_;\_)\}\cup$$
$$\{\mathsf{LState}_{pp}(\_;\hat{v}^*;\_;\_) \wedge \hat{v}_i\ \hat{\cancel{\oslash}}\ \hat{v}_j \implies \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*;\_;\_)\}$$

$$(\!|\texttt{binop}_{\oplus}\ r_d\ r_i\ r_j|\!)_{pp} \;=\; \{\mathsf{LState}_{pp}(\_;\hat{v}^*;\_;\_) \implies \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*[d \mapsto \hat{v}_i\ \hat{\oplus}\ \hat{v}_j];\_;\_)\}$$

$$(\!|\texttt{unop}_{\odot}\ r_d\ r_i|\!)_{pp} \;=\; \{\mathsf{LState}_{pp}(\_;\hat{v}^*;\_;\_) \implies \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*[d \mapsto \hat{\odot}\hat{v}_i];\_;\_)\}$$

$$(\!|\texttt{move } r_d\ rhs|\!)_{pp} \;=\; \langle\!\langle rhs \rangle\!\rangle_{pp} \cup \{\mathsf{RHS}_{pp}(\hat{v}') \wedge \mathsf{LState}_{pp}(\_;\hat{v}^*;\_;\_) \implies \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*[d \mapsto \hat{v}'];\_;\_)\}$$

$$(\!|\texttt{move } r_a[r_{idx}]\ rhs|\!)_{pp} \;=\; \langle\!\langle rhs \rangle\!\rangle_{pp} \cup \{\mathsf{RHS}_{pp}(\hat{v}'') \wedge \mathsf{LState}_{pp}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_a(\hat{v}^*;\hat{h};\mathsf{NFS}(\lambda);\tau[\hat{v}']) \wedge \mathsf{Reach}(\hat{v}'';\hat{h};\hat{k}')$$
$$\implies \mathsf{H}(\lambda,\tau[\hat{v}'\sqcup\hat{v}'']) \wedge \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{c,m,pc+1}(\_;\mathsf{lift}(\hat{v}^*;\hat{k}');\mathsf{hlift}(\hat{h};\hat{k}');\hat{k}\,\hat{\sqcup}\,\hat{k}')\}\cup$$
$$\{\mathsf{RHS}_{pp}(\hat{v}'') \wedge \mathsf{LState}_{pp}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_a(\hat{v}^*;\hat{h};\mathsf{FS}(\lambda);\tau[\hat{v}'])$$
$$\implies \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*;\hat{h}[\lambda\mapsto\tau[\hat{v}'\sqcup\hat{v}''];\hat{k})\}$$

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

$$(\!|\texttt{move } c'.f\ rhs|\!)_{pp} \;=\; \langle\!\langle rhs \rangle\!\rangle_{pp} \cup \{\mathsf{RHS}_{pp}(\hat{v}') \wedge \mathsf{LState}_{pp}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{Reach}(\hat{v}';\hat{h};\hat{k}')$$
$$\implies \mathsf{S}_{c',f}(\hat{v}') \wedge \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{c,m,pc+1}(\_;\mathsf{lift}(\hat{v}^*;\hat{k}');\mathsf{hlift}(\hat{h};\hat{k}');\hat{k}\,\hat{\sqcup}\,\hat{k}')\}$$

$$(\!|\texttt{instof } r_d\ r_s\ \tau|\!)_{pp} \;=\; \{\mathsf{LState}_{pp}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_s(\hat{v}^*;\hat{h};\_;\hat{b}) \wedge \widehat{\mathit{get\text{-}type}}(\hat{b}) \leq \tau$$
$$\implies \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*[d\mapsto\widehat{true}];\hat{h};\hat{k})\} \cup$$
$$\{\mathsf{LState}_{pp}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_s(\hat{v}^*;\hat{h};\_;\hat{b}) \wedge \widehat{\mathit{get\text{-}type}}(\hat{b}) \not\leq \tau$$
$$\implies \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*[d\mapsto\widehat{false}];\hat{h};\hat{k})\}$$

$$(\!|\texttt{checkcast } r_s\ \tau|\!)_{pp} \;=\; \{\mathsf{LState}_{pp}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_s(\hat{v}^*;\hat{h};\_;\hat{b}) \wedge \widehat{\mathit{get\text{-}type}}(\hat{b}) \leq \tau \implies \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*;\hat{h};\hat{k})\}$$

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

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

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

$$(\!|\texttt{start-act } r_i|\!)_{pp} \;=\; \{\mathsf{LState}_{pp}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{NFS}(\lambda);\{\!|@c';\hat{u}|\!\})$$
$$\implies \mathsf{I}_c(\{\!|@c';\hat{u}|\!\}) \wedge \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*;\hat{h};\hat{k})\} \cup$$
$$\{\mathsf{LState}_{pp}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{FS}(\lambda);\{\!|@c';\hat{u}|\!\}) \wedge \mathsf{Reach}(\mathsf{FS}(\lambda);\hat{h};\hat{k}')$$
$$\implies \mathsf{I}_c(\{\!|@c';\hat{u}|\!\}) \wedge \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{c,m,pc+1}(\_;\mathsf{lift}(\hat{v}^*;\hat{k}');\mathsf{hlift}(\hat{h};\hat{k}');\hat{k}\,\hat{\sqcup}\,\hat{k}')\}$$

$$(\!|\texttt{put-extra } r_i\ r_k\ r_j|\!)_{pp} \;=\; \{\mathsf{LState}_{pp}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{NFS}(\lambda);\{\!|@c';\hat{v}'|\!\}) \wedge \mathsf{Reach}(\hat{v}_j;\hat{h};\hat{k}') \implies$$
$$\mathsf{H}(\lambda,\{\!|@c';\hat{v}'\sqcup\hat{v}_j|\!\}) \wedge \mathsf{LiftHeap}(\hat{h};\hat{k}') \wedge \mathsf{LState}_{c,m,pc+1}(\_;\mathsf{lift}(\hat{v}^*;\hat{k}');\mathsf{hlift}(\hat{h};\hat{k}');\hat{k}\,\hat{\sqcup}\,\hat{k}')\}\cup$$
$$\{\mathsf{LState}_{pp}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\mathsf{FS}(\lambda);\{\!|@c';\hat{v}'|\!\})$$
$$\implies \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*;\hat{h}[\lambda\mapsto\{\!|@c';\hat{v}'\sqcup\hat{v}_j|\!\}];\hat{k})\}$$

$$(\!|\texttt{get-extra } r_i\ r_k\ \tau|\!)_{pp} \;=\; \{\mathsf{LState}_{pp}(\_;\hat{v}^*;\hat{h};\hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*;\hat{h};\_;\{\!|@c';\hat{v}'|\!\}) \implies \mathsf{LState}_{c,m,pc+1}(\_;\hat{v}^*[\mathsf{res}\mapsto\hat{v}'];\hat{h};\hat{k})\}$$

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

Conventions: $pp = c, m, pc$

TABLE XXIX
ABSTRACT SEMANTICS OF μ-DALVIK$_A$ - STANDARD STATEMENTS

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

Conventions: $pp = c, m, pc$

TABLE XXX
ABSTRACT SEMANTICS OF μ-DALVIK$_A$ - INVOKE STATEMENTS

Statement Abstractions:

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

$(\!|\texttt{interrupt } r_i|\!)_{pp}$ $=$ $\{\mathsf{LState}_{pp}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{NFS}(\lambda); \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \_|\!\})$
$\implies \mathsf{H}(\lambda, \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \widehat{true}|\!\} \wedge \mathsf{LState}_{c,m,pc+1}(\_; \hat{v}^*; \hat{h}; \hat{k})\} \cup$
$\{\mathsf{LState}_{pp}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{FS}(\lambda); \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \_|\!\})$
$\implies \mathsf{LState}_{c,m,pc+1}(\_; \hat{v}^*; \hat{h}[\lambda \mapsto \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \widehat{true}|\!\}]; \hat{k})\}$

$(\!|\texttt{interrupted } r_i|\!)_{pp}$ $=$ $\{\mathsf{LState}_{pp}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{NFS}(\lambda); \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \hat{v}'|\!\})$
$\implies \mathsf{H}(\lambda, \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \widehat{false}|\!\} \wedge \mathsf{LState}_{c,m,pc+1}(\_; \hat{v}^*[\mathsf{res} \mapsto \hat{v}']; \hat{h}; \hat{k})\} \cup$
$\{\mathsf{LState}_{pp}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \mathsf{FS}(\lambda); \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \hat{v}'|\!\})$
$\implies \mathsf{LState}_{c,m,pc+1}(\_; \hat{v}^*[\mathsf{res} \mapsto \hat{v}']; \hat{h}[\lambda \mapsto \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \widehat{false}|\!\}]; \hat{k})\}$

$(\!|\texttt{is-interrupted } r_i|\!)_{pp}$ $=$ $\{\mathsf{LState}_{pp}(\_; \hat{v}^*; \hat{h}; \hat{k}) \wedge \mathsf{GetBlk}_i(\hat{v}^*; \hat{h}; \_; \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \hat{v}'|\!\})$
$\implies \mathsf{LState}_{c,m,pc+1}(\_; \hat{v}^*[\mathsf{res} \mapsto \hat{v}']; \hat{h}; \hat{k})\}$

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

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

$(\!|\texttt{monitor-enter } r_i|\!)_{pp}$ $=$ $\{\mathsf{LState}_{pp}(\_; \hat{v}^*; \hat{h}; \hat{k}) \implies \mathsf{LState}_{c,m,pc+1}(\_; \hat{v}^*; \hat{h}; \hat{k})\}$

$(\!|\texttt{monitor-exit } r_i|\!)_{pp}$ $=$ $\{\mathsf{LState}_{pp}(\_; \hat{v}^*; \hat{h}; \hat{k}) \implies \mathsf{LState}_{c,m,pc+1}(\_; \hat{v}^*; \hat{h}; \hat{k})\}$

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

$(\!|\texttt{move-except } r_d|\!)_{pp}$ $=$ $\{\mathsf{LState}_{c,m,pc}(\_; \hat{v}^*; \hat{h}; \hat{k}) \implies \mathsf{LState}_{c,m,pc+1}(\_; \hat{v}^*[d \mapsto \hat{v}_{\mathsf{except}}]; \hat{h}; \hat{k})\}$

Global Abstractions:

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

Conventions: $pp = c, m, pc$

TABLE XXXI

ABSTRACT SEMANTICS OF μ-DALVIK$_A$ - RULES FOR NEW STATEMENTS

APPENDIX C
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 who 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 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).*

A. 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.

a) 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 $\perp$. 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 $\perp$.

b) 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 $\perp$ such that there is *exactly* one key-value binding for all pp. Formally we have:

Definition 9 *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 an memory block storing a location to an element of G.

Definition 10 $H \rightarrow_{\mathsf{ref}} G$ *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 11 $(G, (K_i)_{i \leq n})$ *is a heap decomposition of* $H \cdot S$ *if and only if:*

- $H = G \cup \bigcup_{i \leq n} K_i$
- $\forall i.dom(G) \cap dom(K_i) = \emptyset$
- $\forall i \neq j.dom(K_i) \cap dom(K_j) = \emptyset$
- $\forall i.G \cup S \not\rightarrow_{\text{ref}} K_i$ *and* $\forall j \neq i.K_i \not\rightarrow_{\text{ref}} K_j$
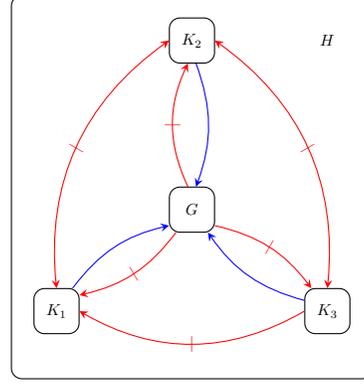
Example: a local heap decomposition with three local heaps.

B. 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 12 *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 13 *For all* $i \in \mathbb{N} \cup \{+\infty\}$*,* $\Gamma^i(K_a, (\mathsf{lk}^j)_j)$ *is the function defined as follows: let* $\mathsf{lk} = \mathsf{lk}^1 \sqcup^{\mathsf{loc}} \ldots \sqcup^{\mathsf{loc}} \mathsf{lk}^{i-1}$*, 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 **??**.

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

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

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

	pp_1	pp_2	pp_3	pp_4	pp_5	pp_6	pp_7
lk_5	ℓ_{17}		ℓ_{18}				
lk_4		ℓ_{15}			ℓ_{16}		
							$\Gamma^4(K_a, (lk_i)_{i\leq 5})$
lk_3				ℓ_{12}	ℓ_{13}	ℓ_{14}	
lk_2	ℓ_{11}						
							$\Gamma^2(K_a, (lk_i)_{i\leq 5})$
lk_1		ℓ_8	ℓ_9			ℓ_{10}	

K_a	ℓ_1	ℓ_2	ℓ_3	ℓ_4	ℓ_5	ℓ_6	ℓ_7

Convention: Each line of the table represents one local filter, by having a pointer ℓ in position (lk_i, pp_j) if and only if there exists p such that $\ell = p_{pp}$ and $lk_i(\ell) = 1$. 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 $\Gamma^2(K_a(lk_i)_{i\leq 5})$ (resp. $\Gamma^4(K_a, (lk_i)_{i\leq 5})$).
TABLE XXXII
GRAPHICAL REPRESENTATION OF THE $\Gamma^j(K_a, (lk_i)_{i\leq n})$ FUNCTIONS

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

$$\Gamma^i(K_a, (lk_j)_{1\leq j\leq n}) = \Gamma^i(K'_a, (lk_a \sqcup^{\text{loc}} lk_1) :: (lk_j)_{2\leq j\leq n})$$

We can now define when $(K, (lk^j)_j)$ is a filter history of a call-stack α. Equation (**??**) 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 $L_j :: \ldots L_n$. Equation (**??**) 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 14 $(K, (lk^j)_j)$ *is a filter history of* $\alpha = L_1 :: \cdots :: L_n$ *if and only if for all* $1 \leq i < l \leq n$ *and for all* pp *we have:*

$$\Gamma^i(K, (lk^j)_j)(pp) \neq \Gamma^l(K, (lk^j)_j)(pp) \implies \Gamma^i(K, (lk^j)_j)(pp) \notin dom(L_l :: \ldots :: L_n) \tag{1}$$

$$\forall i, \forall p_{pp}, \big((i = 0 \wedge p_{pp} \in dom(K)) \vee lk^i(p_{pp}) = 1\big) \implies \forall j \neq i, lk^j(p_{pp}) = 0 \tag{2}$$

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

Lemma 2 *Let* $(K, (lk^j)_j)$ *be a filter history of* $\alpha = L_1 :: \alpha_t$. *Let* $\alpha' = L'_1 :: \alpha_t$, *and* $(K'_a, (lk'^j)_j)$ *be such that* $(lk'^j)_j = lk'^1 :: (lk^j)_{j>1}$, *and let* n *be the length of* α'. *If the four following conditions holds:*

$$\forall i > 1, \forall pp, \Gamma^i(K, (lk^j)_j)(pp) \quad = \quad \Gamma^i(K', (lk'^j)_j)(pp) \tag{3}$$

$$(dom(K')\backslash dom(K)) \cap dom(\alpha_t) \quad = \quad \emptyset \tag{4}$$

$$(dom(K')\backslash dom(K)) \cap \{\ell \mid \exists j, lk^j(\ell) = 1\} \quad = \quad \emptyset \tag{5}$$

$$\{\ell \mid lk'^1(\ell) = 1 \wedge lk'^1(\ell) \neq lk^1(\ell)\} \quad \subseteq \quad dom(K)\backslash dom(K') \tag{6}$$

then $(K', (lk'^j)_j)$ *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:

$$\Gamma^i(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \implies \Gamma^i(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \notin dom(\alpha'_l :: \ldots :: \alpha'_n) \tag{7}$$

  - For $1 < i < l \leq n$, using Equation (**??**) 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 (**??**) 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 (**??**) 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 (**??**) we get that $\Gamma^1(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \notin dom(\alpha_t)$, which shows that Equation (**??**) holds.

- Now we are going to show that:

$$\forall i, \forall p_{\mathsf{pp}}, \left((i = 0 \wedge p_{\mathsf{pp}} \in dom(K')) \vee \mathsf{lk}'^i(p_{\mathsf{pp}}) = 1\right) \implies \forall j \neq i, \mathsf{lk}'^j(p_{\mathsf{pp}}) = 0$$

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 (**??**) 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 (**??**) we know that $\forall j > 2, \mathsf{lk}'^j(\ell) = \mathsf{lk}^j(\ell)$. Besides by Equation (**??**) 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 (**??**).
  - $i = 1$. 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 (**??**) 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.

■

C. 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 15 $(G, (K_i)_i, K, (\mathsf{lk}^j)_j)$ *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* $K \in (K_i)_i$
- $dom(\alpha) \subseteq dom(G) \cup dom(K)$
- $(K, (\mathsf{lk}^j)_j)$ *is a filter history of* α
- $\forall i \in \pi, \exists p_\lambda, (p_\lambda \mapsto i) \in G$
- $\forall \ell \in \gamma, \ell \in dom(G)$
- $\ell \in dom(G)$

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

Definition 16 *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* $\phi_i \in \{\langle \ell, s, \pi, \gamma, \alpha \rangle, \overline{\langle \ell, s, \pi, \gamma, \alpha \rangle}\}$ *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.

D. 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 1 (Exception Table Correction) *If* ExcptTable(c, m, pc, c') *is defined (i.e is equal to some pc' or to $\bot$) then $c' \leq$ Throwable.*

Assumption 2 (Type Soundness Guarrantee)
- *If $\Sigma, $ `throw` $r_e \Downarrow \Sigma'$ and $H(\Sigma[\![r_e]\!]) = \{\!|c'; (f \mapsto v)^*|\!\}$ then $c' \leq$ Throwable.*
- *If $\Sigma, st \Downarrow \Sigma'$ where $st \in \{$`start-thread` $r_t,$ `interrupt` $r_t,$ `join` $r_t\}$ and $H(\Sigma[\![r_t]\!]) = \{\!|c'; (f \mapsto v)^*|\!\}$ then $c' \leq$ Thread.*

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

Definition 17 *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 = $ `AbNormal`$(L_1 :: \ldots :: L_n)$, we have:*
- *For all i, $L_i = $ **waiting**$(\_,\_)$ implies that $i = 1$ and $\alpha = $ `AbNormal`$(L_1 :: \ldots :: L_n)$.*
- *If $L_1 = $ **waiting**$(\ell_o, \_)$ then $L_2 = \langle c, m, pc \cdot \_ \cdot st^* \cdot \_ \rangle$ with $st_{pc} = $ `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 $c' \leq$ 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} = $ `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} = $ `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 3 (Preserving Local Well-formation) *If Σ is well-formed and $\Sigma \leadsto^* \Sigma'$, 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 18 *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 3 (Java Type Soundness)
If $\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S \leadsto \ell' \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S'$, 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 19 *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 4 (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 **??** and Assumption **??** 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 **??**.

E. 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.

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

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 location. 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.

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

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

2) 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 $\beta_{LstInv}^\ell(L, n_0, c', K_a, (\mathsf{lk}^n)_n)$ where n_0 is the position is the call-stack and c' is the class of the object that L invoked a method upon.

$$\beta_{LstInv}^\ell(\langle \mathsf{pp} \cdot u^* \cdot st^* \cdot R \rangle, n_0, c', K_a, (\mathsf{lk}^n)_n) = \Big\{ \mathsf{Inv}_{\mathsf{pp}}^{c'}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{k}) \mid \hat{k} = \beta_{Filter}(\mathsf{lk}^{n_0})$$

$$\wedge \, \forall j : \hat{u}_j = \beta_{LocVal}(u_j, K_a, (\mathsf{lk}^n)_{\mathbf{n} \leq \mathbf{n_0}}) \wedge \hat{\lambda}_t = \beta_{Val}(\ell) \wedge \forall k : \hat{v}_k = \beta_{LocVal}(R(r_k), K_a, (\mathsf{lk}^n)_{\mathbf{n} < \mathbf{n_0}}) \Big\}$$

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

$$\beta_{Lst}^\ell(\langle \mathsf{pp} \cdot u^* \cdot st^* \cdot R \rangle, K_a, (\mathsf{lk}^n)_n) = \Big\{ \mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \mid \hat{k} = \beta_{Filter}(\mathsf{lk}^1)$$

$$\wedge \, \forall j : \hat{u}_j = \beta_{LocVal}(u_j, K_a, (\mathsf{lk}^n)_{\mathbf{n} \leq \mathbf{1}}) \wedge \hat{\lambda}_t = \beta_{Val}(\ell) \wedge \forall k : \hat{v}_k = \beta_{LocVal}(R(r_k), K_a, (\mathsf{lk}^n)_{\mathbf{n} < \mathbf{1}}) \wedge \hat{h} = \beta_{LHeap}(K_a) \Big\}$$

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

$$\beta_{ALst}^{\ell}(\langle \mathsf{pp} \cdot u^* \cdot st^* \cdot R \rangle, K_a, (\mathsf{lk}^n)_n) = \Big\{ \mathsf{AState}_{\mathsf{pp}}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \mid \hat{k} = \beta_{Filter}(\mathsf{lk}^1)$$

$$\wedge \, \forall j : \hat{u}_j = \beta_{LocVal}(u_j, K_a, (\mathsf{lk}^n)_{\mathbf{n} \leq \mathbf{1}}) \wedge \hat{\lambda}_t = \beta_{Val}(\ell) \wedge \forall k : \hat{v}_k = \beta_{LocVal}(R(r_k), K_a, (\mathsf{lk}^n)_{\mathbf{n} < \mathbf{1}}) \wedge \hat{h} = \beta_{LHeap}(K_a) \Big\}$$

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:

$$\begin{aligned}
\beta_{Call}^{\ell}(\mathsf{waiting}(\_,\_) :: \alpha, K_a, (\mathsf{lk}^n)_n) &= \beta_{Call}^{\ell}(\alpha, K_a, (\mathsf{lk}^n)_n) \\
&= \beta_{Lst}^{\ell}(L_1, K_a, (\mathsf{lk}^n)_n) \cup \bigcup_{i \in [2,n]} \beta_{LstInv}^{\ell}(L_i, i, c_{i-1}, K_a, (\mathsf{lk}^n)_n) \\
\beta_{Call}^{\ell}(\mathtt{AbNormal}(\alpha), K_a, (\mathsf{lk}^n)_n) &= \beta_{ALst}^{\ell}(L_1, K_a, (\mathsf{lk}^n)_n) \cup \bigcup_{i \in [2,n]} \beta_{LstInv}^{\ell}(L_i, i, c_{i-1}, K_a, (\mathsf{lk}^n)_n) \\
\beta_{Call}^{\ell}(\varepsilon, K_a, (\mathsf{lk}^n)_n) &= \beta_{Call}^{\ell}(\mathtt{AbNormal}(\varepsilon), K_a, (\mathsf{lk}^n)_n) = \emptyset
\end{aligned}$$

We can now define the following representation functions:

$$\begin{aligned}
\beta_{Heap}^{G}(H) &= \Big\{ \mathsf{H}(\lambda, \hat{b}) \mid H(\ell') = b \wedge \lambda = \beta_{Lab}(\ell') \wedge \hat{b} = \beta_{Blk}(b) \wedge \ell' \in dom(G) \Big\} \\
\beta_{Stat}(S) &= \{ \mathsf{S}(c, f, \hat{v}) \mid S = S', c.f \mapsto v \wedge \hat{v} = \beta_{Val}(v) \} \\
\beta_{Pact}^{\ell}(\pi) &= \Big\{ \mathsf{I}_c(\hat{b}) \mid c = \beta_{Lab}(\ell) \wedge \pi = \pi_0 :: i :: \pi_1 \wedge \hat{b} = \beta_{Blk}(i) \Big\} \\
\beta_{Pthr}^{G}(\gamma) &= \Big\{ \mathsf{T}(\lambda, \hat{b}) \mid \gamma = \gamma_0 :: \ell :: \gamma_1 \wedge \lambda = \beta_{Lab}(\ell) \wedge (\ell \mapsto b) \in G \wedge \hat{b} = \beta_{Blk}(b) \Big\} \\
\beta_{Frm}^{G}(\langle \ell, s, \pi, \gamma, \alpha \rangle, K_a, (\mathsf{lk}^j)_j) &= \beta_{Frm}(\langle \ell, s, \pi, \gamma, \alpha \rangle, K_a, (\mathsf{lk}^j)_j) \\
&= \beta_{Frm}(\langle\!\langle \ell, \ell', \pi, \gamma, \alpha \rangle\!\rangle, K_a, (\mathsf{lk}^j)_j) \\
&= \beta_{Call}^{\ell}(\alpha, K_a, (\mathsf{lk}^j)_j) \cup \beta_{Pact}^{\ell}(\pi) \cup \beta_{Pthr}^{G}(\gamma)
\end{aligned}$$

Let $\Omega = \varphi_1 :: \ldots :: \varphi_n$ and $\Xi = \psi_1 :: \ldots :: \psi_m$. We then define the representation function β_{Stk}^{G} abstracting the activity stack and the thread pool as follows:

$$\beta_{Stk}^{G}(\Omega, \Xi, (K_i, (\mathsf{lk}^{i,j})_j)_i) = \left(\bigcup_{i \in [1,n]} \beta_{Frm}^{G}(\varphi_i, K_i, (\mathsf{lk}^{i,j})_j) \right) \cup \left(\bigcup_{l \in [1,m]} \beta_{Frm}^{G}(\psi_l, K_{n+l}, (\mathsf{lk}^{n+l,j})_j) \right)$$

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:

$$\beta_{Lcnf}(\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S) = \Big\{ \beta_{Call}^{\ell}(\alpha, K_a, (\mathsf{lk}^j)_j) \cup \beta_{Pact}^{\ell}(\pi) \cup \beta_{Pthr}^{G}(\gamma) \cup \beta_{Heap}^{G}(H) \cup \beta_{Stat}(S)$$

$$\mid (G, (K_i)_i, K_a, (\mathsf{lk}^j)_j) \text{ is a local configuration decomposition of } \ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S \Big\}$$

The representation function β_{Cnf} abstracts a configuration Ψ into a set of sets of abstract facts, one for each configuration decomposition of Ψ:

$$\beta_{Cnf}(\Omega \cdot \Xi \cdot H \cdot S) = \Big\{ \beta_{Stk}^{G}(\Omega, (K_i, (\mathsf{lk}^{i,j})_j)_i) \cup \beta_{Heap}^{G}(H) \cup \beta_{Stat}(S)$$

$$\mid (G, (K_i, (\mathsf{lk}^{i,j})_j)_i) \text{ is a configuration decomposition of } \Omega \cdot \Xi \cdot H \cdot S \Big\}$$

Remark 5 *The predicates* $\mathsf{Inv}_{\mathsf{pp}}^{c'}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{k})$ *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.

F. 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.

1) 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*

$$\hat{b} \sqsubseteq^{nfs}_{Blk} \hat{b}' \wedge \hat{u} \sqsubseteq^{nfs} \hat{v} \implies \hat{b}[f \mapsto \hat{u}] \sqsubseteq^{nfs}_{Blk} \hat{b}'[f \mapsto \hat{v}]$$
$$\hat{b} \sqsubseteq_{Blk} \hat{b}' \wedge \hat{u} \sqsubseteq \hat{v} \implies \hat{b}[f \mapsto \hat{u}] \sqsubseteq_{Blk} \hat{b}'[f \mapsto \hat{v}]$$

2) 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:

$$\mathsf{Call}^{\Delta}_{r_o, c'', m'}(\hat{v}^*; \hat{h}) = \exists \mathsf{pp}', c', \Big((\mathsf{NFS}(\mathsf{pp}') \sqsubseteq \hat{v}_o \wedge \mathsf{H}(\mathsf{pp}', \{\!| c'; \_ |\!\}) \in \Delta) \vee \big(\mathsf{FS}(\mathsf{pp}') \sqsubseteq \hat{v}_o \wedge \hat{h}(\mathsf{pp}') = \{\!| c'; \_ |\!\} \big) \Big)$$
$$\wedge\, c' \leq c'' \wedge c'' \in \widehat{lookup}(m')$$

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 $\sqsubseteq^{\Delta'}_{Inv}$ as follows:

- $\mathsf{LState}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{call}); \hat{u}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}')$ iff
  - $\hat{\lambda}^1_t = \hat{\lambda}^2_t$ and $\hat{u}^*_{call} \sqsubseteq_{Seq} \hat{v}^*_{call}$
  - $\hat{u}^* \sqsubseteq_{Seq} \hat{v}^*$
  - $\hat{k} \sqsubseteq_{Filter} \hat{k}'$
  - $\forall \mathsf{pp}, \hat{h}(\mathsf{pp}) \neq \bot \implies \hat{h}(\mathsf{pp}) \sqsubseteq_{Blk} \hat{h}'(\mathsf{pp})$
- $\mathsf{AState}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{call}); \hat{u}^*; \hat{h}; \hat{k}) \sqsubseteq_A \mathsf{AState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}')$ iff :

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{call}); \hat{u}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}')$$

- $\mathsf{Inv}^{c''}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{call}); \hat{u}^*; \hat{k}) \sqsubseteq^{\Delta}_{Inv} \mathsf{LState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}')$ iff:
  - $\hat{\lambda}^1_t = \hat{\lambda}^2_t$ and $\hat{u}^*_{call} \sqsubseteq_{Seq} \hat{v}^*_{call}$

- $\hat{u}^* \sqsubseteq_{Seq} \hat{v}^*$
- $\hat{k} \sqsubseteq_{Filter} \hat{k}'$
- $lookup(c, m) = (\_, st^*)$, $st_{pc} = \texttt{invoke } r_o\ m'\ \_$ 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:

- $\forall \mathsf{LState}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{call}); \hat{u}^*; \hat{h}; \hat{k}) \in \Delta$, $\exists \mathsf{LState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}') \in \Delta'$ s.t.

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{call}); \hat{u}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}')$$

- $\forall \mathsf{AState}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{call}); \hat{u}^*; \hat{h}; \hat{k}) \in \Delta$, $\exists \mathsf{AState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}') \in \Delta'$ s.t.

$$\mathsf{AState}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{call}); \hat{u}^*; \hat{h}; \hat{k}) \sqsubseteq_A \mathsf{AState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}')$$

- $\forall \mathsf{Inv}^{c''}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{call}); \hat{u}^*; \hat{k}) \in \Delta$, $\exists \mathsf{LState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}') \in \Delta'$ s.t.

$$\mathsf{Inv}^{c''}_{c,m,pc}((\hat{\lambda}^1_t, \hat{u}^*_{call}); \hat{u}^*; \hat{k}) \sqsubseteq^{\Delta'}_{Inv} \mathsf{LState}_{c,m,pc}((\hat{\lambda}^2_t, \hat{v}^*_{call}); \hat{v}^*; \hat{h}'; \hat{k}')$$

- $\forall \mathsf{H}(\lambda, \hat{b}) \in \Delta$, $\exists \mathsf{H}(\lambda, \hat{b}') \in \Delta'$ 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 $\hat{u} \sqsubseteq^{\mathsf{nfs}} \hat{v}$
- $\forall \mathsf{I_c}(\hat{b}) \in \Delta$, $\exists \mathsf{I_c}(\hat{b}') \in \Delta'$ 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 $\hat{b} \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b}'$

G. Preliminary Lemmas

1) Pre-orders:

Lemma 5 *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 6 *If $\Delta \subseteq \Delta'$, and $\Delta' <: \Delta''$ then $\Delta <: \Delta''$.*

Lemma 7 *If $\Delta_1 <: \Delta_2$ and $\Delta_3 <: \Delta_4$, then $\Delta_1 \cup \Delta_3 <: \Delta_2 \cup \Delta_4$.*

Lemma 8 *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:

$$\left(\overbrace{(\mathsf{NFS}(\mathsf{pp}') \sqsubseteq \hat{v}'_{r_o} \wedge \mathsf{H}(\mathsf{pp}', \{|c'; \_|\}) \in \Delta')}^{A} \vee \overbrace{(\mathsf{FS}(\mathsf{pp}') \sqsubseteq \hat{v}'_{r_o} \wedge \hat{h}'(\mathsf{pp}') = \{|c'; \_|\})}^{B} \right) \wedge c' \leq c'' \wedge c'' \in \widehat{lookup}(m')$$

  - 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.

∎

2) Representation Function:

Proposition 6 *For all filter history* $K, (\mathsf{lk}^j)_j$ *we have:*

- *For any block* b, $\beta_{LocBlk}(b, K) \sqsubseteq^{nfs}_{Blk} \beta_{Blk}(b)$ *and* $\beta_{Blk}(b) \sqsubseteq^{nfs}_{Blk} \beta_{LocBlk}(b, K)$.
- *For any value* v, $\beta_{LocVal}(v, K, (\mathsf{lk}^j)_j) \sqsubseteq^{nfs} \beta_{Val}(v)$ *and* $\beta_{Val}(v) \sqsubseteq^{nfs} \beta_{LocVal}(v, K, (\mathsf{lk}^j)_j)$.

Proof: This is following from the fact that the pre-orders $\sqsubseteq^{nfs}_{Blk}$ and $\sqsubseteq^{nfs}$ ignore the flow-sensitive and flow-insensitive annotations of the abstract labels. ∎

Assumption 4 (Soundness of the Abstract Operations) $\hat{\oslash}, \hat{\odot}$ *and* $\hat{\oplus}$ *are monotonous operators, and soundly over-approximate the concrete operators* $\oslash, \odot$ *and* $\oplus$*: for all local heap* K, *we have:*

- $u \oslash v$ *implies that* $\beta_{LocVal}(u, K) \hat{\oslash} \beta_{LocVal}(v, K)$
- $\beta_{LocVal}(\odot v, K) \sqsubseteq \hat{\odot}\beta_{LocVal}(v, K)$
- $\beta_{LocVal}(u \oplus v, K) \sqsubseteq \beta_{LocVal}(u, K) \hat{\oplus} \beta_{LocVal}(v, K)$

This carry over to all the representation functions $\beta_{LocVal}(\cdot, K, (\mathsf{lk}^i)_i)$ (with order $\sqsubseteq$) and $\beta_{Val}(\cdot)$ (with order $\sqsubseteq^{nfs}$):

Proposition 7 *For all concrete values* u *and* v, *and for all filter history* $K, (\mathsf{lk}^i)_i$ *we have:*

- $u \oslash v$ *implies that* $\beta_{LocVal}(u, K, (\mathsf{lk}^i)_i) \hat{\oslash} \beta_{LocVal}(v, K, (\mathsf{lk}^i)_i)$ *and that* $\beta_{Val}(u) \hat{\oslash} \beta_{Val}(v)$
- $\beta_{LocVal}(\odot v, K, (\mathsf{lk}^i)_i) \sqsubseteq \hat{\odot}\beta_{LocVal}(v, K, (\mathsf{lk}^i)_i)$ *and* $\beta_{Val}(\odot v) \sqsubseteq^{nfs} \hat{\odot}\beta_{Val}(v)$
- $\beta_{LocVal}(u \oplus v, K, (\mathsf{lk}^i)_i) \sqsubseteq \beta_{LocVal}(u, K, (\mathsf{lk}^i)_i) \hat{\oplus} \beta_{LocVal}(v, K, (\mathsf{lk}^i)_i)$ *and* $\beta_{Val}(u \oplus v) \sqsubseteq^{nfs} \beta_{Val}(u) \hat{\oplus} \beta_{Val}(v)$

Proof: Observe that for all filter history $K, (\mathsf{lk}^i)_i$, we have that for all concrete value u:

$$\beta_{LocVal}(u, K, (\mathsf{lk}^i)_i) = \beta_{LocVal}\left(u, \left(\mathsf{pp} \mapsto \Gamma^\infty(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})\right)^*\right)$$

This together with Assumption **??** shows the first point of each item bullet.

The second point of each item bullet follows from the fact that if $\sqsubseteq^{nfs}$ 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:

$$\begin{array}{ll}
\beta_{LocVal}(\odot v, K) \sqsubseteq \hat{\odot}\beta_{LocVal}(v, K) & \text{by Assumption } \textbf{??} \\
\beta_{LocVal}(\odot v, K) \sqsubseteq^{nfs} \hat{\odot}\beta_{LocVal}(v, K) & \text{by Proposition } \textbf{??} \\
\beta_{Val}(\odot v) \sqsubseteq^{nfs} \beta_{LocVal}(\odot v, K) \sqsubseteq^{nfs} \hat{\odot}\beta_{LocVal}(v, K) & \text{by Proposition } \textbf{??}
\end{array}$$

By Proposition **??** we know that $\beta_{LocVal}(v, K) \sqsubseteq^{nfs} \beta_{Val}(v)$, therefore by monotonicity of $\hat{\odot}$ we get that $\hat{\odot}\beta_{LocVal}(v, K) \sqsubseteq^{nfs} \hat{\odot}\beta_{Val}(v)$. This concludes the $\hat{\odot}$ case by showing that:

$$\beta_{Val}(\odot v) \sqsubseteq^{nfs} \beta_{LocVal}(\odot v, K) \sqsubseteq^{nfs} \hat{\odot}\beta_{LocVal}(v, K) \sqsubseteq^{nfs} \hat{\odot}\beta_{Val}(v)$$

∎

Assumption 5 (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 $\mathsf{f} \in \Delta'$.

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* $\hat{u} \sqsubseteq^{nfs} \hat{v}$ *then* $\mathsf{lift}(\hat{u}; \hat{k}) \sqsubseteq^{nfs} \mathsf{lift}(\hat{v}; \hat{k})$.
- *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:

$$\hat{u} = \hat{v}_1 \sqsubseteq_{Loc} \hat{v}'_1 \sqsubseteq \hat{v}_2 \sqsubseteq_{Loc} \hat{v}'_2 \ldots \hat{v}'_{n-1} \sqsubseteq \hat{v}_n \sqsubseteq_{Loc} \hat{v}'_n = \hat{v}$$

By Proposition **??**.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 **??**.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:

$$\mathsf{lift}(\hat{u};1^*) = \mathsf{lift}(\hat{v}_1;1^*) \sqsubseteq \mathsf{lift}(\hat{v}'_1;1^*) \sqsubseteq \mathsf{lift}(\hat{v}_2;1^*) \ldots \mathsf{lift}(\hat{v}_n;1^*) \sqsubseteq \mathsf{lift}(\hat{v}'_n;1^*) = \mathsf{lift}(\hat{v};1^*)$$

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:*

$$\beta_{Frm}(\Omega_i, K, (\mathsf{lk}'^j)_j) = \beta_{Frm}(\Omega_i, K, (\mathsf{lk}^j)_j)$$

3) Technical lemmas:

Lemma 9 (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 $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{pp}(\hat{v})$.
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:

$$\beta^\ell_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, K, (\mathsf{lk}^j)_j) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \tag{8}$$

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 **??**, $\ell \in \bigcup_i dom(K_i)$ implies that $\ell \in dom(K)$. Therefore we are in one of the two following cases:

- $\ell \in dom(G)$: from Equation **??** 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 $\mathsf{H}(\lambda, \{\!|c; (f_i \mapsto \hat{u}_i)^*, f \mapsto \hat{v}_f|\!\}) \in \Delta$ such that $\beta_{Val}(v) \sqsubseteq^{\mathsf{nfs}} \hat{v}_f$. Let $\hat{v} = \mathsf{lift}(\hat{v}_f;1^*)$, then we have $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{pp}(\hat{v})$ by applying the rule:

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{NFS}(\lambda) \sqsubseteq \hat{v}'_i \wedge \mathsf{H}(\lambda, \{\!|c; (f_i \mapsto \hat{u}_i)^*, f \mapsto \hat{v}_f|\!\}) \implies \mathsf{RHS}_{pp}(\mathsf{lift}(\hat{v}_f;1^*))$$

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 **??** 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 **??** we get that $\hat{v}_i = \beta_{LocVal}(\ell, K) = \mathsf{FS}(\lambda)$. Moreover since:

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \tag{9}$$

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 **??** 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 $\Delta \cup \langle\!\langle rhs\rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{pp}(\hat{v}'_f)$ by applying the following rule, which is in $\langle\!\langle r_i.f\rangle\!\rangle_{pp}$:

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{FS}(\lambda) \sqsubseteq \hat{v}'_i \wedge \hat{h}'(\lambda) = \{\!|c; (f_i \mapsto \hat{u}''_i)^*, f \mapsto \hat{v}'_f|\!\} \implies \mathsf{RHS}_{pp}(\hat{v}'_f)$$

∎

Lemma 10 (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 $\hat{h}(\lambda) = \{\!|c; (f_i \mapsto \hat{v}_i)_i|\!\}$
- $Reach^{n+1}_\lambda = Reach^n_\lambda \cup Reach^n_{\hat{v}}$ if $\hat{h}(\lambda) = \tau[\hat{v}]$
- $Reach^{n+1}_\lambda = Reach^n_\lambda \cup Reach^n_{\hat{v}}$ if $\hat{h}(\lambda) = \{\!|@\tau; \hat{v}|\!\}$

For all λ (resp. $\hat{v}$), $(Reach^n_\lambda)_{n \geq 0}$ (resp. $(Reach^n_{\hat{v}})_{n \geq 0}$) is an 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 $Reach_\lambda = \bigcup_{n \geq 0} Reach^n_\lambda$ (resp. $Reach_{\hat{v}} = \bigcup_{n \geq 0} Reach^n_{\hat{v}}$). Moreover since $\hat{h}$ is finite, this limit is reached in a finite number of steps. Therefore there exists N such that $Reach_\lambda = \bigcup_{n \leq N} Reach^n_\lambda$ and $Reach_{\hat{v}} = \bigcup_{n \leq N} Reach^n_{\hat{v}}$.

   We define I^λ_n to be the indicator function of $Reach^n_\lambda$, and $I^{\hat{v}}_n$ to be the indicator function of $Reach^n_{\hat{v}}$. We will see I^λ_n and $I^{\hat{v}}_n$ 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 $\vdash \mathsf{Reach}(\mathsf{FS}(\lambda); \hat{h}; I^\lambda_n)$ and $\vdash \mathsf{Reach}(\hat{v}; \hat{h}; I^\lambda_n)$ (observe that the second point uses the fact that there is a finite number of λ). Therefore we have $\vdash \mathsf{Reach}(\hat{u}; \hat{h}; I^{\hat{u}}_N)$, where $I^{\hat{u}}_N$ is the indicator function of $Reach^N_{\hat{u}} = Reach_{\hat{u}}$. ∎

Lemma 11 (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:

$$\beta_{Loc}(p_\lambda, K') = \mathsf{NFS}(\lambda) = \mathsf{lift}(\mathsf{NFS}(\lambda); \hat{k}_a) = \mathsf{lift}(\beta_{Loc}(p_\lambda, K); \hat{k}_a) = \mathsf{lift}(\hat{v}; \hat{k}_a)$$

∎

Lemma 12 (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:*

$$\beta^{\ell_r}_{Lst}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R[r_d \mapsto w] \rangle, K'[p_{\lambda'} \mapsto b], (\mathsf{lk}_a \sqcup_f \mathsf{lk}^1) :: (\mathsf{lk}^n)_{n>1}))$$
$$= \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a)[d \mapsto \beta_{LocVal}(w, K')]; \mathsf{hlift}(\hat{h}; \hat{k}_a)[\lambda' \mapsto \beta_{LocBlk}(b, K')]; \hat{k}_a \hat{\sqcup} \hat{k})$$

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 **??**.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 **??** 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 $p_{pp} \in dom(K')$ such that $\mathsf{hlift}(\hat{h}; \hat{k}_a)(pp) = \beta_{LocBlk}(K'(p_{pp}), K')$. Since K' is a local heap, we know that there exists $\ell = p_{pp} \in dom(K')$. 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 $\hat{k}_a(pp) = 1$, therefore we have:

$$\beta_{LocBlk}(K'(\ell), K') = \beta_{LocBlk}(\bot, K') = \bot = \mathsf{hlift}(\hat{h}; \hat{k}_a)(pp)$$

  - $\ell \notin S$. Then by hypothesis we know that $K'(\ell) = K(\ell)$. Assume that $K(\ell) = \{\!|c; (f_i \mapsto u_i)_{i \leq n}|\!\}$ (the array and intent cases are similar). Then we have:

$$\beta_{LocBlk}(K'(\ell), K') = \{\!|c; (f_i \mapsto \beta_{LocVal}(u_i, K'))_{i \leq n}|\!\}$$

Since S is fresh in Σ we know that for all $i \leq n$, $u_i \notin S$. Therefore by Lemma **??**, for all $i \leq n$, we have $\beta_{LocVal}(u_i, K')_{i \leq n} = \mathsf{lift}(\beta_{LocVal}(u_i, K); \hat{k}_a)$. Moreover since $\ell \in dom(K') \backslash S$, we know that $\hat{k}_a(\lambda) = 0$. Therefore:

$$\{\!|c; (f_i \mapsto \beta_{LocVal}(u_i, K'))_{i \leq n}|\!\} = \{\!|c; (f_i \mapsto \mathsf{lift}(\beta_{LocVal}(u_i, K); \hat{k}_a))_{i \leq n}|\!\} = \mathsf{hlift}(\hat{h}; \hat{k}_a)(\lambda)$$

- $\hat{k}_a \hat{\sqcup} \hat{k} = \beta_{Filter}(\mathsf{lk}_a \sqcup_f \mathsf{lk}^1)$: this is trivial.

∎

We can now state the local preservation lemma, which shows that our abstraction soundly over-approximates the concrete reduction $\rightsquigarrow^*$ between local reduction.

Lemma 13 (Local Preservation) *If $\Sigma \leadsto^* \Sigma'$ under a given program P, then for any $X \in \beta_{Lcnf}(\Sigma)$ with local configuration decomposition $(G, (K_i)_{i \leq n}, K, (\mathsf{lk}^j)_j)$, for any $\Delta :> X$ there exists Δ' and $X' \in \beta_{Lcnf}(\Sigma')$ with local configuration decomposition $(G', (K'_i)_{i \leq n}, K', (\mathsf{lk}'^j)_j)$ 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 **??**.

H. 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 you serialize 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 14 *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 15 *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.

- $\dfrac{p_\lambda \notin dom(\Gamma) \quad p'_\lambda \text{ fresh pointer} \quad \Gamma, p_\lambda \mapsto p'_\lambda \vdash ser^H_{Blk}(H(p_\lambda)) = (b, H'', \Gamma') \quad H' = H'', p'_\lambda \mapsto b}{\Gamma \vdash ser^H_{Val}(p_\lambda) = (p'_\lambda, H', \Gamma')}$:
  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 $image(\Gamma') \cap dom(H) = \emptyset$.

- $\dfrac{\Gamma_0 = \Gamma \quad \forall i \in [1, n] : \Gamma_{i-1} \vdash ser^H_{Val}(v_i) = (u_i, H_i, \Gamma_i) \quad H' = H_1, \ldots, H_n}{\Gamma \vdash ser^H_{Blk}(\{|c'; (f_i \mapsto v_i)^{i \leq n}|\}) = (\{|c'; (f_i \mapsto u_i)^{i \leq n}|\}, H', \Gamma_n)}$:
  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 16 *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 $(\_ \mapsto b') \not\mapsto_{\mathsf{ref}} H$.*

Proof: Simple proof by case analysis on the last (or two last) derivation rule(s) applied. ∎

Lemma 17 *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 $\beta^{G \cup H'}_{Heap}(H') = \emptyset$

- $$\frac{(p_\lambda \mapsto p'_\lambda) \in \Gamma}{\Gamma, \vdash ser^H_{Val}(p_\lambda) = (p'_\lambda, \cdot, \Gamma)}$$: idem here we have $\beta^{G \cup H'}_{Heap}(H') = \emptyset$

- $$\frac{p_\lambda \notin dom(\Gamma) \quad p'_\lambda \text{ fresh pointer} \quad \Gamma, p_\lambda \mapsto p'_\lambda \vdash ser^H_{Blk}(H(p_\lambda)) = (b, H'', \Gamma') \quad H' = H'', p'_\lambda \mapsto b}{\Gamma \vdash ser^H_{Val}(p_\lambda) = (p'_\lambda, H', \Gamma')}$$:

  Since $p_\lambda \in dom(G)$ we know that $(p_\lambda \mapsto H(p_\lambda)) \in G$. Therefore by induction we know that $\Delta \mathrel{>:} \beta^{G \cup H''}_{Heap}(H'')$. 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 $\Delta \mathrel{:>} \beta^{G \cup H'}_{Heap}(H')$ we just need to show that:

  $$
  \begin{aligned}
  \Delta \quad &\mathrel{:>} \quad \beta^{G \cup H'}_{Heap}(p'_\lambda \mapsto b) \\
  &= \quad \{\mathsf{H}(\lambda, \beta_{Blk}(b))\} \\
  &= \quad \{\mathsf{H}(\lambda, \beta_{Blk}(H(p_\lambda)))\} \quad \text{by Lemma \textbf{??}} \\
  &= \quad \beta^G_{Heap}(p_\lambda \mapsto H(p_\lambda)) \quad \text{since } p_\lambda \in dom(G)
  \end{aligned}
  $$

  The last point is implied by the fact that $\Delta \mathrel{:>} \beta^G_{Heap}(H)$.

  Moreover by induction we know that $G \cup H'' \cdot (K_i)_i$ is a heap decomposition of $H \cup H''$. By Lemma **??** we know that $(\_ \mapsto b) \not\rightarrow_{\mathsf{ref}} H$. 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 $H \cup H'$.

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

  By applying repeatedly Lemma **??** 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 $(p_\lambda \mapsto \{\!|c'; (f_i \mapsto v_i)^{i \le n}|\!\})) \in G$. 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 $i \in [1, n]$ $\Delta \mathrel{:>} \beta^{G \cup H_i}_{Heap}(H_i)$, which implies that :

  $$\Delta \mathrel{:>} \bigcup_{1 \le i \le n} \beta^{G \cup H_i}_{Heap}(H_i) = \beta^{G \cup (\bigcup_{1 \le i \le n} H_i)}_{Heap}\left(\bigcup_{1 \le i \le n} H_i\right)$$

  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 $G \cup \left(\bigcup_{1 \le i \le n} H_i\right) \cdot (K_i)_i$ is a heap decomposition of $H\left(\bigcup_{1 \le i \le n} H_i\right)$.

- Block serialization of arrays and intents works exactly like the object case.

$\blacksquare$

I. Proof of Theorem 1

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 6 *For all configuration decomposition $(G, (K_i, (\mathsf{lk}^{i,j})_j)_i)$, for all location ℓ pointing to an activity object, for all life-cycle 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 2 (Global Preservation) *If $\Psi \Rightarrow^* \Psi'$ under a given program P, then for any $X \in \beta_{Cnf}(\Psi)$, for any $\Delta \mathrel{:>} X$ there exists Δ' and $X' \in \beta_{Cnf}(\Psi')$ such that $\Delta' \mathrel{:>} X'$ and $(\!|P|\!) \cup \Delta \vdash \Delta'$.*

The proof can be found in Section **??**.

J. Application to Taint Tracking

Lemma 18 (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:*

$$\beta_{Lst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, K_n, (\mathsf{lk}^{n,j})_j)$$
$$= \quad \mathsf{LState}_{c,m,pc}((\hat\lambda_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \qquad\qquad\qquad \in \Delta$$
$$\sqsubseteq_R \quad \mathsf{LState}_{c,m,pc}((\hat\lambda'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \qquad\qquad\qquad \in \Delta'$$

and there exists $\hat{t}$ *such that* $\mathsf{taint}_\Psi(R(r_k)) \sqsubseteq^t \hat{t}$ *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 $\mathsf{taint}_\Psi(R(r_k)) \sqsubseteq^t \hat{t}$ and :

$$(\!|P|\!) \cup \Delta' \vdash \mathsf{Taint}(\hat{v}'_i, \hat{h}', \hat{t})$$

We let:

$$\mathsf{taint}_\Psi^0(u) = \begin{cases} t & \text{if } u = prim^t \\ \mathsf{public} & \text{otherwise} \end{cases}$$

For all n we define the following functions:

$$\mathsf{taint}_\Psi^{n+1}(u) = \begin{cases} \sqcup^t_i \, \mathsf{taint}_\Psi^n(v_i) & \text{if } u = \ell \wedge H(\ell) = \{\!|c; (f_i \mapsto v_i)^*|\!\} \\ \sqcup^t_i \, \mathsf{taint}_\Psi^n(v_i) & \text{if } u = \ell \wedge H(\ell) = \tau[v^*] \\ \sqcup^t_i \, \mathsf{taint}_\Psi^n(v_i) & \text{if } u = \ell \wedge H(\ell) = \{\!|@c; (k_i \mapsto v_i)^*|\!\} \\ t & \text{if } u = prim^t \end{cases}$$

We know that $\mathsf{taint}_\Psi(v) = \lim_{n \in \mathbb{N}} \mathsf{taint}_\Psi^n(v)$ and that this limit is reached in a finite number of step (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 $\mathsf{taint}_\Psi^n(u) \sqsubseteq^t \hat{t}$ 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 19 *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 2, 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 1, 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 **??** 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:

$$\beta_{Lst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, K_n, (\mathsf{lk}^{n,j})_j)$$
$$= \quad \mathsf{LState}_{c,m,pc}((\hat\lambda_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \qquad\qquad\qquad \in \Delta'$$
$$\sqsubseteq_R \quad \mathsf{LState}_{c,m,pc}((\hat\lambda'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \qquad\qquad\qquad \in \Delta''$$

and there exists $\hat{t}$ such that $\mathsf{taint}_{\Psi'}(R(r_k)) \sqsubseteq^t \hat{t}$ 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:

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

∎

K. Proof of Lemma **??**

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 $\Delta :> X$.

a) 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 $L_1' = \langle c', m', pc' \cdot u'^* \cdot st'^* \cdot R' \rangle$.

b) 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:
   - $\beta_{Call}^{\ell_r}(\alpha', K', (\mathsf{lk}'^j)_j) \backslash \beta_{Call}^{\ell_r}(\alpha, K, (\mathsf{lk}^j)_j) \subseteq D_{Call}$
   - $\beta_{Heap}^{G'}(H') \backslash \beta_{Heap}^{G}(H) \subseteq D_{Heap}$
   - $\beta_{Stat}(S') \backslash \beta_{Stat}(S) \subseteq D_{Stat}$
   - $\beta_{Pact}^{\ell_r}(\pi') \backslash \beta_{Pact}^{\ell_r}(\pi) \subseteq D_{Pact}$
   - $\beta_{Pthr}^{G'}(\gamma') \backslash \beta_{Pthr}^{G}(\gamma) \subseteq D_{Pthr}$

3) Define $\Delta_{Call}, \Delta_{Heap}, \Delta_{Stat}, \Delta_{Pact}$ 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 $X' = \beta_{Call}^{\ell_r}(\alpha', K', (\mathsf{lk}'^j)_j) \cup \beta_{Heap}^{G'}(H') \cup \beta_{Stat}(S') \cup \beta_{Pact}^{\ell_r}(\pi') \cup \beta_{Pthr}^{G'}(\gamma')$ 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 **??** 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 **??** we have $X' <: X \cup D_{Call} \cup D_{Heap} \cup D_{Stat} \cup D_{Pact} \cup D_{Pthr}$, and by applying Lemma **??** 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 **??** we know that for all $i \geq 2$ we have:

$$\beta_{LstInv}^{\ell_r}(\alpha_i, i, \_, K, (\mathsf{lk}^n)_n) = \beta_{LstInv}^{\ell_r}(\alpha_i, i, \_, K', (\mathsf{lk}'^n)_n)$$

  Hence $D_{Call} = \beta_{Lst}^{\ell_r}(\langle c, m, pc' \cdot u^* \cdot st^* \cdot R \rangle, K', (\mathsf{lk}'^n)_n)$ satisfies the wanted properties.

  3) We know that $\beta_{Lst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$ 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:

$$\beta_{LSt}^{\ell_r}(\langle c, m, pc' \cdot u^* \cdot st^* \cdot R\rangle, K', (\mathsf{lk}'^n)_n) = \mathsf{LState}_{c,m,pc'}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$$

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 `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:

$$\beta_{LSt}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta \quad (10)$$

Let $\Sigma[\![r_o]\!] = \ell''$, we know by Proposition **??** we know that either $\ell'' \in G$ or $\ell'' \in K$.

Case 1: $\ell'' \in G$

By Lemma **??** we know that $\beta_{LocVal}(\Sigma[\![r_o]\!], K) \sqsubseteq \hat{v}'_{r_o}$. Moreover by applying Lemma **??** 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 **??** 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:

$$\begin{aligned}
&dom(K')\setminus \{p_{\mathsf{pp}} \in dom(K') \mid \exists p', \mathsf{lk}_a(p'_{\mathsf{pp}}) = 1\} \\
=\ &dom(K')\setminus \{p_{\mathsf{pp}} \in dom(K') \mid \exists p', p'_{\mathsf{pp}} \in dom(Reach_a)\} \\
=\ &dom(K')\setminus dom(M) \\
\subseteq\ &dom(K)
\end{aligned}$$

Therefore by Proposition **??**.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 **??** 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 **??** we know that for all $j \geq 2$:

$$\beta_{LStInv}^{\ell_r}(L_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta_{LStInv}^{\ell_r}(L_j, j, \_, K', (\mathsf{lk}'^i)_i)$$

One can then show that the following definitions of D_{Call} and D_{Heap} satisfy the wanted properties:

* $D_{Call} = \beta_{LSt}^{\ell_r}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R\rangle, K', (\mathsf{lk}'^i)_i)$
* $D_{Heap} = \{\mathsf{H}(\lambda, \hat{b}) \mid H(\ell') = b \wedge \lambda = \beta_{Lab}(\ell') \wedge \hat{b} = \beta_{Blk}(b) \wedge \ell' \in dom(Reach_a)\}$
  $\cup \{\mathsf{H}(\lambda, \hat{b}) \mid \lambda = \beta_{Lab}(\ell'') \wedge \hat{b} = \beta_{Blk}(H(\ell'')[f \mapsto \beta_{Val}(\Sigma[\![rhs]\!])])\}$

3) * $\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}_a \sqcup \hat{k}')$.
   * 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 $\beta_{Heap}^G(H) <: \Delta$ and $H(\ell'') \neq \bot$ we know that there exists $\mathsf{H}(\lambda_o, \hat{b}_o) \in \Delta$ such that $\beta_{Blk}(H(\ell'')) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}_o$ and $\lambda_o = \beta_{Lab}(\ell'')$. Then we add $\mathsf{H}(\lambda_o, \hat{b}_o[f \mapsto \hat{v}''])$ to Δ_{Heap}.

4) We are going to show that:

* $D_{Call} <: \Delta \cup \Delta_{Call}$: by applying Lemma **??**.1 we know that:

$$\beta_{Lst}^{\ell_r}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R \rangle, K', (\mathsf{lk}'^n)_n)) = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a); \mathsf{hlift}(\hat{h}; \hat{k}_a); \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k})$$

Therefore we just have to prove that:

$$\mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a); \mathsf{hlift}(\hat{h}; \hat{k}_a); \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k})$$
$$\sqsubseteq_R \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t', \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}') \quad (11)$$

From Equation (**??**) 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 (**??**) 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 **??** 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 **??** 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:

$$\beta_{Lst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$$

Therefore by definitions of $\beta_{Lst}^{\ell_r}$ 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 **??** we know that $\beta_{Blk}(b) \sqsubseteq_{Blk}^{\mathsf{nfs}} \beta_{LocBlk}(b, K)$. In summary:

$$\hat{b} = \beta_{Blk}(b) \sqsubseteq_{Blk}^{\mathsf{nfs}} \beta_{LocBlk}(b, K) = \hat{h}(\lambda) \quad (12)$$

By Equation (**??**) 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 **??** we get that $\hat{h}(\mathsf{pp}) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{h}'(\mathsf{pp})$. Putting Equation (**??**) together with this we get that:

$$\hat{b} \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{h}(\lambda) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{h}'(\lambda)$$

We know that $\hat{k}_a(\lambda) = 1$. Besides $\hat{h}(\lambda) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{h}'(\lambda)$ 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:

$$\{\mathsf{H}(\lambda, \hat{b}) \mid \lambda = \beta_{Lab}(\ell'') \wedge \hat{b} = \beta_{Blk}(H(\ell'')[f \mapsto \Sigma[\![rhs]\!]])\} <: \Delta_{Heap}$$

Recall that $\beta_{LocVal}(\Sigma[\![rhs]\!], K) \sqsubseteq \hat{v}''$, $\mathsf{H}(\lambda_o, \hat{b}_o) \in \Delta$, $\beta_{Blk}(H(\ell'')) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}_o$, $\lambda_o = \beta_{Lab}(\ell'')$ and $H(\lambda_o, \hat{b}_o[f \mapsto \hat{v}'']) \in \Delta_{Heap}$.

By Proposition **??** we have $\beta_{LocVal}(\Sigma[\![rhs]\!], K) \sqsubseteq^{\mathsf{nfs}} \hat{v}''$, and by Proposition **??** 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 **??** to $\beta_{Blk}(H(\ell'')) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}_o$ and $\beta_{Val}(\Sigma[\![rhs]\!]) \sqsubseteq^{\mathsf{nfs}} \hat{v}''$ we get that :

$$\beta_{Blk}(H(\ell''))[f \mapsto \beta_{Val}(\Sigma[\![rhs]\!])] \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}_o[f \mapsto \hat{v}'']$$

Which proves that :

$$\mathsf{H}(\lambda_o, \beta_{Blk}(\mathsf{H}(\ell'')[f \mapsto \Sigma[\![rhs]\!]])) <: \mathsf{H}(\lambda_o, \hat{b}_o[f \mapsto \hat{v}'']) <: \Delta_{Heap}$$

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:

$$\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}_a \sqcup \hat{k}')$$

We proved at the beginning of this case that $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{pp}(\hat{v}'')$ 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 **??** applied to ℓ'' and $\mathsf{LState}_{pp}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$ 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:

$$\mathsf{NFS}(\lambda_o) \sqsubseteq \hat{v}'_o \wedge \mathsf{H}(\lambda_o, \hat{b}_o) \implies \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda_o); \hat{b}_o)$$

Finally we apply the following rule:

$$\mathsf{RHS}_{pp}(\hat{v}'') \wedge \mathsf{LState}_{pp}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda_o); \hat{b}_o) \wedge \mathsf{Reach}(\hat{v}''; \hat{h}'; \hat{k}_a)$$
$$\implies \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}'); \mathsf{hlift}(\hat{h}; \hat{k}'); \hat{k}_a \sqcup \hat{k}')$$

This concludes this case.

$*$ $(\!|P|\!) \cup \Delta \vdash \Delta_{Heap}$: $(\!|P|\!)$ contains the two following rules:

$$\mathsf{RHS}_{pp}(\hat{v}'') \wedge \mathsf{LState}_{pp}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda_o); \hat{b}_o) \wedge \mathsf{Reach}(\hat{v}''; \hat{h}'; \hat{k}_a)$$
$$\wedge \mathsf{H}(\lambda_o, \{|c'; (f' \mapsto \hat{u}'')^*, f \mapsto \_|\}) \implies \mathsf{H}(\lambda_o, \{|c'; (f' \mapsto \hat{u}'')^*, f \mapsto \hat{v}'')|\}) \tag{13}$$
$$\mathsf{RHS}_{pp}(\hat{v}'') \wedge \mathsf{LState}_{pp}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda_o); \hat{b}_o)$$
$$\wedge \mathsf{Reach}(\hat{v}''; \hat{h}'; \hat{k}_a) \wedge \mathsf{Reach}(\hat{v}''; \hat{h}'; \hat{k}_a) \implies \mathsf{LiftHeap}(\hat{h}'; \hat{k}_a) \tag{14}$$

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 (**??**) 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 (**??**).

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 (**??**) 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 (**??**) 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 **??**.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 **??** 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 **??** 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 **??** we know that there exists $\hat{v}''$ such that $\beta_{LocVal}(\Sigma[\![rhs]\!], K) \sqsubseteq \hat{v}''$ and $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{pp}(\hat{v}'')$. 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:

$$\beta_{LSt}^{\ell_r}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R \rangle, K, (\mathsf{lk}^n)_n)$$
$$= \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \underbrace{\hat{h}[\lambda_o \mapsto \hat{h}(\lambda_o)[f \mapsto \beta_{LocVal}(\Sigma[\![rhs]\!], K)]]}_{\hat{h}_1}; \hat{k})$$

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:

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \tag{15}$$

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 **??** 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 **??** 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 $\Delta \cup \langle\!\langle rhs \rangle\!\rangle_{pp} \vdash \mathsf{RHS}_{pp}(\hat{v}'')$.
   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:

$$\mathsf{FS}(\lambda_o) \sqsubseteq \hat{v}'_o \wedge \hat{b}_o = \hat{h}'(\lambda_o) \implies \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}; \mathsf{FS}(\lambda_o); \hat{b}_o)$$
$$\mathsf{RHS}_{pp}(\hat{v}'') \wedge \mathsf{LState}_{pp}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}; \mathsf{FS}(\lambda_o); \hat{b}_o)$$
$$\implies \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}[\lambda \mapsto \hat{b}_o[f \mapsto \hat{v}'']]; \hat{k}')$$

Which conclude this case.

- (R-CALL)
  Since Σ reduces to Σ' by applying the rule $\texttt{invoke } r_o \; m' \; (r_{i_k})^{k \leq n}$ we know that $\Sigma[\![r_o]\!] = \ell$ and that

$$lookup(type_H(\ell), m') = (c', st'^*) \qquad\qquad sign(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$$

$$R' = ((r_j \mapsto \mathbf{0})^{j \leq loc}, r_{loc+1} \mapsto \ell, (r_{loc+1+k} \mapsto \Sigma[\![r_{i_k}]\!])^{k \leq n}) \qquad \alpha' = \langle c', m', 0 \cdot (\Sigma[\![r_{i_k}]\!])^{k \leq n} \cdot st'^* \cdot R' \rangle :: \alpha$$

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 **??**.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:

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

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:

$$\Gamma^{i-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^{l-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \text{ implies } \Gamma^{i-1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \notin dom(\alpha_{|\geq l-1})$$

   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 **??** 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:

$$D_{Call} = \{\beta^{\ell_r}_{Lst}(\langle c', m', 0 \cdot (\Sigma[\![r_{i_k}]\!])^{k \leq n} \cdot st'^* \cdot R' \rangle, K', (\mathsf{lk}'^j)_j)\} \cup \{\beta^{\ell_r}_{LstInv}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, 2, c', K', (\mathsf{lk}'^j)_j)\}$$

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

$$\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \quad (16)$$

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:

$$\beta^{\ell_r}_{Lst}(\langle c', m', 0 \cdot (\mathbf{0}_k)^{k \leq loc}, (\Sigma[\![r_{i_k}]\!])^{k \leq n} \cdot st'^* \cdot R' \rangle, K', (\mathsf{lk}'^j)_j) = \mathsf{LState}_{c',m',0}((\hat{\lambda}_t, \hat{u}^*_{call}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \hat{u}^*_{call}; \hat{h}; 0^*) \quad (17)$$

$$\beta^{\ell_r}_{LstInv}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, 2, c', K', (\mathsf{lk}'^j)_j) \qquad\qquad = \mathsf{Inv}^{c'}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{k}) \quad (18)$$

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:

$$\mathsf{Inv}^{c'}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{k}) \qquad\qquad \sqsubseteq^{\Delta}_{Inv} \quad \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \quad (19)$$

$$\mathsf{LState}_{c',m',0}((\hat{\lambda}_t, \hat{u}^*_{call}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \hat{u}^*_{call}; \hat{h}; 0^*) \quad \sqsubseteq_R \quad \mathsf{LState}_{c',m',0}((\hat{\lambda}'_t, \hat{u}'^*_{call}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \hat{u}'^*_{call}; \hat{h}'; 0^*) \quad (20)$$

Eq. (??): All conditions are trivial consequences of Equation (**??**), 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 **??** 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 $H(\lambda_o, \{\!|c''; \_|\!\}) \in X$, 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 $H(\lambda_o, \{\!|c''; \_|\!\}) \in X$, then there exists $\hat{b}$ such that $H(\lambda_o, \{\!|c''; \hat{b}|\!\}) \in \Delta$. 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. (??): The fact that $0^* \sqsubseteq_{Filter} 0^*$ is trivial. From Equation (**??**) 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 $H(\lambda_o, \{\!|c''; \hat{b}|\!\}) \in \Delta$ then we can apply the rule:

$$\mathsf{NFS}(\lambda_o) \sqsubseteq \hat{v}'_o \wedge H(\lambda_o, \{\!|c''; \hat{b}|\!\}) \implies \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda_o); \{\!|c''; \hat{b}|\!\})$$

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 $(\![P]\!)$:

$$\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_o(\hat{v}'^*; \hat{h}'; \_; \{\!|c''; \hat{b}|\!\}) \wedge c'' \leq c' \implies$$
$$\mathsf{LState}_{c',m',0}((\hat{\lambda}'_t, \hat{u}'_{call}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \hat{u}'_{call}; \hat{h}'; 0^*)$$

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 $\Sigma \leadsto \Sigma'$ 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 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:

$$\Gamma^i(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \implies \Gamma^i(K', (\mathsf{lk}'^j)_j)(\mathsf{pp}) \notin dom(\alpha'_{|\geq l})$$

- If $1 < i < l \leq n$, then for all pp we have:

$$\Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \text{ iff } \Gamma^{i+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^{l+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})$$

Moreover since $(K_a, (\mathsf{lk}^j)_j)$ is a filter history of α we know that:

$$\Gamma^{i+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^{l+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \text{ implies } \Gamma^{i+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \notin dom(\alpha_{|\geq l+1})$$

$\alpha_{|\geq l+1} = \alpha'_{|\geq l}$, and $\Gamma^{i+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) = \Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp})$, hence:

$$\Gamma^{i+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \notin dom(\alpha_{|>l+1}) \implies \Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \notin dom(\alpha'_{|\geq l})$$

Therefore we have:

$$\Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \implies \Gamma^i(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \notin dom(\alpha'_{|\geq l})$$

- If $i = 1$ and $1 < l \leq n$. For all pp we have:

$$\Gamma^1(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \neq \Gamma^l(K'_a, (\mathsf{lk}'^j)_j)(\mathsf{pp}) \text{ iff } \Gamma^1(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp}) \neq \Gamma^{l+1}(K_a, (\mathsf{lk}^j)_j)(\mathsf{pp})$$

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 **??** 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:

$$\beta^{\ell_r}_{Lst}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, K, (\mathsf{lk}^j)_j) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}_1^*); \hat{v}_1^*; \hat{h}_1; \hat{k}_1) \qquad (21)$$
$$\sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{w}'_1, \hat{u}_1^*); \hat{v}_1'^*; \hat{h}'_1; \hat{k}'_1) \in \Delta$$

$$\beta^{\ell_r}_{LstInv}(\langle c', m', pc' \cdot u'^* \cdot st'^* \cdot R' \rangle, 2, c, K, (\mathsf{lk}^j)_j) = \mathsf{Inv}^c_{c',m',pc'}((\hat{\lambda}_t, \hat{u}_2^*); \hat{v}_2^*; \hat{k}_2) \qquad (22)$$
$$\sqsubseteq^\Delta_{Inv} \mathsf{LState}_{c',m',pc'}((\hat{w}'_2, \hat{u}_2'^*); \hat{v}_2'^*; \hat{h}'_2; \hat{k}'_2) \in \Delta$$

Let $\Delta_{Call} = \{\mathsf{LState}_{c',m',pc'+1}((\hat{w}'_2, \hat{u}_2'^*); \mathsf{lift}(\hat{v}_2'^*; \hat{k}'_1)[\mathsf{res} \mapsto (\hat{v}_1'^*)_{\mathsf{res}}]; \hat{h}'_1; \hat{k}'_1 \mathbin{\hat\sqcup} \hat{k}'_2)\}$.

4) By Proposition **??**.1 and Proposition **??**.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

$$\beta_{LocVal}((u_2^*)_k, K, \mathsf{lk}^1 :: \mathsf{lk}^2) = \beta_{LocVal}((u_2^*)_k, K, \mathsf{lk}^1 \sqcup^{\mathsf{loc}} \mathsf{lk}^2) \qquad (23)$$

Let r_d be a register different from r_{res}, we want to show that:

$$\beta_{LocVal}(R'(r_d), K) = \mathsf{lift}(\beta_{LocVal}(R'(r_d), K, \mathsf{lk}^1); \hat{k}_1) \qquad (24)$$

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 **??** and Equation **??** 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 (**??**).
- $\forall i, \mathsf{lift}(\hat{v}_2^*; \hat{k}_1)[\mathsf{res} \mapsto (\hat{v}_1^*)_{\mathsf{res}}] \sqsubseteq \mathsf{lift}(\hat{v}_2'^*; \hat{k}'_1)[\mathsf{res} \mapsto (\hat{v}_1'^*)_{\mathsf{res}}]$: the case where $i = r_{\mathsf{res}}$ is a trivial consequence of Equation (**??**).
  Assume $i \neq r_{\mathsf{res}}$: from Equation (**??**) 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 (**??**) that $\hat{v}_2 \sqsubseteq_{Seq} \hat{v}_2'^*$, therefore by applying Proposition **??** 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 (**??**), Equation (**??**) and $\beta^{\ell_r}_{Lst}$ 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 **??** 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) 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 $sign(c', m') = (\tau_i)_{i \leq n} \xrightarrow{loc} \tau$, $st'_{pc'} = \mathtt{invoke}\ r_o\ m\ (r_{j_i})_{i \leq n}$ and $u^* = (R'(r_{j_i}))_{i \leq n}$. Moreover from Equation (**??**) 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 (**??**) 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 **??** we get that $(\hat{v}_2'^*)_{j_i} \sqcap (\hat{u}_1'^*)_i \neq \bot$.
Similarly from Equation (**??**) we get that $\hat{\lambda}_t = \beta_{Val}(\ell_r) = \hat{w}'_1$, and from Equation (**??**) we get that $\hat{\lambda}_t = \beta_{Val}(\ell_r) = \hat{w}'_2$, hence we have $\hat{w}'_1 = \hat{w}'_2$.
From Equation (**??**) 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:

$$\left(\overbrace{(\mathsf{NFS}(\lambda_o) \sqsubseteq (\hat{v}_2'^*)_o \wedge \mathsf{H}(\lambda_o, \{\!| c''; \_ |\!\}) \in \Delta)}^{A} \vee \overbrace{\left(\mathsf{FS}(\lambda_o) \sqsubseteq (\hat{v}_2'^*)_o \wedge \hat{h}'_2(\lambda_o) = \{\!| c''; \_ |\!\} \right)}^{B} \right) \wedge c'' \leq c' \wedge c' \in \widehat{lookup}(m')$$

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 $(\!|P|\!)$:

$$\mathsf{LState}_{c',m',pc'}((\hat{w}_2', \hat{u}_2'^*); \hat{v}_2'^*; \hat{h}_2'; \hat{k}_2') \wedge \mathsf{GetBlk}_o(\hat{v}_2'^*; \hat{h}_2'; \_; \{\!|c''; \_|\!\}) \wedge c'' \leq c'$$
$$\wedge\, \mathsf{Res}_{c,m}((\hat{w}_1', \hat{u}_1'^*); (\hat{v}_1'^*)_{\mathsf{res}}; \hat{h}_1'; \hat{k}_1') \wedge \hat{w}_1' = \hat{w}_2' \wedge \left(\bigwedge_{j \leq n} (\hat{v}_2'^*)_{i_j} \sqcap (\hat{u}_1'^*)_j \neq \bot\right)$$
$$\implies \mathsf{LState}_{c',m',pc'+1}((\hat{w}_2', \hat{u}_2'); \mathsf{lift}(\hat{v}_2'^*; \hat{k}_1')[\mathsf{res} \mapsto (\hat{v}_1'^*)_{\mathsf{res}}]; \hat{h}_1'; \hat{k}_1' \sqcup \hat{k}_2')$$

This shows that $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$.

- **(R-NEWOBJ)**

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

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:

$$\beta_{Lst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$$
$$\sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t', \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta \qquad (25)$$

By Lemma **??** 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:

$$K' = \left((K)_{|dom(K) \setminus dom(Reach_a)} \cup M\right)[\ell \mapsto o]$$

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:

$$dom(K') \setminus \{p_{\mathsf{pp}} \in dom(K') \mid \exists p', \mathsf{lk}_a(p_{\mathsf{pp}}') = 1\}$$
$$= dom(K') \setminus \{p_{\mathsf{pp}} \in dom(K') \mid \exists p', p_{\mathsf{pp}}' \in dom(Reach_a)\}$$
$$= dom(K') \setminus (dom(M) \cup \{\ell\})$$
$$\subseteq dom(K)$$

Hence by Proposition **??**.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 **??**, 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 **??** we know that for all $j \geq 2$,

$$\beta_{LstInv}^{\ell_r}(L_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta_{LstInv}^{\ell_r}(L_j, j, \_, K', (\mathsf{lk}'^i)_i)$$

One can then show that the following definitions satisfy the wanted property:

- $D_{Call} = \beta_{Lst}^{\ell_r}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R[r_d \mapsto \ell] \rangle, K', (\mathsf{lk}'^i)_i)$
- $D_{Heap} = \{\mathsf{H}(\lambda, \hat{b}) \mid H(\ell') = b \wedge \lambda = \beta_{Lab}(\ell') \wedge \hat{b} = \beta_{Blk}(b) \wedge \ell' \in dom(Reach_a)\}$

3)
- $\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})]; \mathsf{hlift}(\hat{h}'; \hat{k}_a)[\mathsf{pp} \mapsto \{\!|c'; (f \mapsto \hat{\mathbf{0}}_\tau)^*|\!\}]; \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}')$
- 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 **??**.2 we get that:

$$\beta_{Lst}^{\ell_r}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R[r_d \mapsto \ell] \rangle, K', (\mathsf{lk}'^n)_n))$$
$$= \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})]; \mathsf{hlift}(\hat{h}; \hat{k}_a)[\mathsf{pp} \mapsto \{\!|c'; (f \mapsto \hat{\mathbf{0}}_\tau)^*|\!\}]; \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k})$$

Therefore we just have to prove that:

$$\mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*); \mathsf{lift}(\hat{v}^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})]; \overbrace{\mathsf{hlift}(\hat{h}; \hat{k}_a)[\mathsf{pp} \mapsto \{\!|c'; (f \mapsto \hat{\mathbf{0}}_\tau)^*|\!\}]}^{\hat{h}_1}; \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}) \quad (26)$$
$$\sqsubseteq_R \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})]; \underbrace{\mathsf{hlift}(\hat{h}'; \hat{k}_a)[\mathsf{pp} \mapsto \{\!|c'; (f \mapsto \hat{\mathbf{0}}_\tau)^*|\!\}]}_{\hat{h}'_1}; \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}')$$

From Equation **(??)** 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 **(??)** 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 **??** 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 $\hat{k}_a \mathbin{\hat{\sqcup}} \hat{k} \sqsubseteq_{Filter} \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}'$.
  * 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 **??** 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:

$$\beta_{Lst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k})$$

Therefore by definitions of $\beta_{Lst}^{\ell_r}$ 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 **??** we know that $\beta_{Blk}(b) \sqsubseteq_{Blk}^{\mathsf{nfs}} \beta_{LocBlk}(b, K)$. In summary:

$$\hat{b} = \beta_{Blk}(b) \sqsubseteq_{Blk}^{\mathsf{nfs}} \beta_{LocBlk}(b, K) = \hat{h}(\lambda) \quad (27)$$

By Equation **(??)** 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 **(??)** together with this we get that $\hat{b} \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{h}(\lambda) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{h}'(\lambda)$.

We know that $\hat{k}_a(\lambda) = 1$. Besides $\hat{h}(\lambda) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{h}'(\lambda)$ 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

$$\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})]; \mathsf{hlift}(\hat{h}'; \hat{k}_a)[\mathsf{pp} \mapsto \{\!|c'; (f \mapsto \hat{\mathbf{0}}_\tau)^*|\!\}]; \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}')$$

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 $(\!|P|\!)$:

$$\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{Reach}(\mathsf{FS}(\mathsf{pp}); \hat{h}'; \hat{k}_a)$$
$$\implies \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \mathsf{lift}(\hat{v}'^*; \hat{k}_a)[d \mapsto \mathsf{FS}(\mathsf{pp})]; \mathsf{hlift}(\hat{h}'; \hat{k}_a)[\mathsf{pp} \mapsto \{\!| c'; (f \mapsto \hat{\mathbf{0}}_\tau)^* |\!\}]; \hat{k}_a \sqcup \hat{k}')$$

This concludes this case.

- $(\!|P|\!) \cup \Delta \vdash \Delta_{Heap}$: we can apply the following rule, which is included in $(\!|P|\!)$:

$$\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{Reach}(\mathsf{FS}(\mathsf{pp}); \hat{h}'; \hat{k}_a) \implies \mathsf{LiftHeap}(\hat{h}'; \hat{k}_a) \tag{28}$$

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 (**??**) plus the above rule yield $(\!|P|\!) \cup \Delta \vdash \mathsf{H}(\mathsf{pp}, \hat{h}'(\mathsf{pp}))$.

- (R-STARTTHREAD)

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

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:

$$\beta_{Lst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R \rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta \tag{29}$$

Let $\ell = \Sigma[\![r_i]\!]$, $H(\ell) = b = \{\!| c'; (f \mapsto w)^* |\!\}$. By Assumption **??** 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 (**??**).
    * $D_{Pthr} <: \Delta_{Pthr}$: this case is trivial since $\hat{b} \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}'$.
5) We know by Lemma **??** 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 $\beta_{Blk}(b) = \beta_{Blk}(\{\!| c'; (f \mapsto w)^* |\!\}) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}'$ we know that $\hat{b}' = \{\!| c'; (f \mapsto \hat{w}) |\!\}$. Moreover we know that $(\!|P|\!)$ contains the two following rules:

$$\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_i(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda); \{\!| c'; (f \mapsto \hat{w}) |\!\}) \wedge c' \leq \mathsf{Thread}$$
$$\implies \mathsf{T}(\lambda, \{\!| c'; (f \mapsto \hat{w})^* |\!\})$$
$$\mathsf{LState}_{\mathsf{pp}}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge \mathsf{GetBlk}_i(\hat{v}'^*; \hat{h}'; \mathsf{NFS}(\lambda); \{\!| c'; (f \mapsto \hat{w}) |\!\}) \wedge c' \leq \mathsf{Thread}$$
$$\implies \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}')$$

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 **??** 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 $K' = \left((K)_{|dom(K) \setminus dom(Reach_a)} \cup M \right)$ 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 **??**.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 **??**, 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 **??** 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:

* $\Delta_{Call} = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*)); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}')$
* We define Δ_{Heap} as follows: for all pp, if $\hat{k}_a(\mathsf{pp}) = 1 \wedge \hat{h}'(\mathsf{pp}) \neq \bot$ then $\mathsf{H}(\mathsf{pp}, \hat{h}'(\mathsf{pp})) \in \Delta_{Heap}$.
* $\ell \in dom(K)$, therefore we know that $\hat{h}(\lambda) = \beta_{LocBlk}(b, K) \neq \bot$. From (**??**) 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 **??**.1 we get that:

$$\beta^{\ell_r}_{Lst}(\langle c, m, pc + 1 \cdot u^* \cdot st^* \cdot R \rangle, K', (\mathsf{lk}'^n)_n)) = \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*)); \mathsf{lift}(\hat{v}^*; \hat{k}_a); \mathsf{hlift}(\hat{h}; \hat{k}_a); \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k})$$

Therefore we just have to prove that:

$$\mathsf{LState}_{c,m,pc+1}((\hat{\lambda}_t, \hat{u}^*)); \mathsf{lift}(\hat{v}^*; \hat{k}_a); \mathsf{hlift}(\hat{h}; \hat{k}_a); \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k} \qquad (30)$$
$$\sqsubseteq_R \mathsf{LState}_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*)); \mathsf{lift}(\hat{v}'^*; \hat{k}_a); \mathsf{hlift}(\hat{h}'; \hat{k}_a); \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}'$$

From Equation (**??**) 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 (**??**) 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 **??** 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 $\hat{k}_a \mathbin{\hat{\sqcup}} \hat{k} \sqsubseteq_{Filter} \hat{k}_a \mathbin{\hat{\sqcup}} \hat{k}'$.
· For all pp, by applying Proposition **??** 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 **??** we know that $\beta_{Blk}(b'') \sqsubseteq_{Blk}^{nfs} \beta_{LocBlk}(b'', K)$. In summary:

$$\hat{b}'' = \beta_{Blk}(b'') \sqsubseteq_{Blk}^{nfs} \beta_{LocBlk}(b'', K) = \hat{h}(\lambda'') \tag{31}$$

From Equation (**??**) we get that $\forall pp, \hat{h}(pp) \neq \bot \implies \hat{h}(pp) \sqsubseteq_{Blk} \hat{h}'(pp)$. 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 (**??**) together with this we get that $\hat{b}'' \sqsubseteq_{Blk}^{nfs} \hat{h}(\lambda'') \sqsubseteq_{Blk}^{nfs} \hat{h}'(\lambda'')$.

We know that $\hat{k}_a(\lambda'') = 1$. Besides $\hat{h}(\lambda'') \sqsubseteq_{Blk}^{nfs} \hat{h}'(\lambda'')$ and $\hat{h}(\lambda'') \neq \bot$ implies that $\hat{h}'(\lambda'') \neq \bot$. Therefore $H(\lambda'', \hat{h}'(\lambda'')) \in \Delta_{Heap}$, which concludes this case.

∗ $\ell \in dom(K)$, therefore $\hat{h}(\lambda) = \beta_{LocBlk}(b, K) \neq \bot$. Hence by Equation (**??**) we know that $\hat{h}(\lambda) \sqsubseteq_{Blk} \hat{h}'(\lambda)$. By Proposition **??** we know that $\hat{b} = \beta_{Blk}(b) \sqsubseteq_{Blk}^{nfs} \beta_{LocBlk}(b, K) = \hat{h}(\lambda)$, and by Proposition **??** we get that $\hat{h}(\lambda) \sqsubseteq_{Blk}^{nfs} \hat{h}'(\lambda)$. Therefore $\hat{b} \sqsubseteq_{Blk}^{nfs} \hat{h}'(\lambda)$, which shows that $D_{Pthr} <: \Delta_{Pthr}$.

5) We are going to show that:

∗ $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$: recall that $LState_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta$ and that:

$$\Delta_{Call} = LState_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); lift(\hat{v}'^*; \hat{k}_a); hlift(\hat{h}'; \hat{k}_a); \hat{k}_a \sqcup \hat{k}')$$

We know by Lemma **??** that $\beta_{LocVal}(\Sigma[\![r_i]\!], K) \sqsubseteq \hat{v}'_i$. Moreover since $\Sigma[\![r_i]\!] = \ell \in dom(K)$ we have $FS(\lambda) = \beta_{LocVal}(\Sigma[\![r_i]\!], K)$. We saw previously that $\beta_{Blk}(b) \sqsubseteq_{Blk}^{nfs} \hat{h}'(\lambda)$, 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:

$$\vdash GetBlk_i(\hat{v}'^*; \hat{h}'; FS(\lambda); \{\!|c'; (f \mapsto \hat{w})^*|\!\})$$

Finally $c' \leq Thread$ and $\vdash Reach(FS(\lambda); \hat{h}'; \hat{k}_a)$, which allows us to apply the following rule, which is included in $(\!|P|\!)$:

$$LState_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge GetBlk_i(\hat{v}'^*; \hat{h}'; FS(\lambda); \{\!|c'; (f \mapsto \hat{w})^*|\!\}) \wedge Reach(FS(\lambda); \hat{h}'; \hat{k}_a)$$
$$\wedge c' \leq Thread \implies LState_{c,m,pc+1}((\hat{\lambda}'_t, \hat{u}'^*); lift(\hat{v}'^*; \hat{k}_a); hlift(\hat{h}'; \hat{k}_a); \hat{k}' \sqcup \hat{k}_a)$$

This concludes this case.

∗ $(\!|P|\!) \cup \Delta \vdash \Delta_{Heap}$: We can apply the following rule, which is in $(\!|P|\!)$:

$$LState_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge GetBlk_i(\hat{v}'^*; \hat{h}'; FS(\lambda); \{\!|c'; (f \mapsto \hat{w})^*|\!\}) \wedge Reach(FS(\lambda); \hat{h}'; \hat{k}_a)$$
$$\wedge c' \leq Thread \implies LiftHeap(\hat{h}'; \hat{k}_a) \quad (32)$$

Δ_{Heap} is the set defined by: for all pp, if $\hat{k}_a(pp) = 1 \wedge \hat{h}'(pp) \neq \bot$ then $H(pp, \hat{h}'(pp)) \in \Delta_{Heap}$. Let pp satisfying those conditions. $(\!|P|\!)$ contains the following rule:

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

Rule Equation (**??**) plus the above rule yield $(\!|P|\!) \cup \Delta \vdash H(pp, \hat{h}'(pp))$.

∗ $(\!|P|\!) \cup \Delta \vdash \Delta_{Pthr}$: directly obtained by applying:

$$LState_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge GetBlk_i(\hat{v}'^*; \hat{h}'; FS(\lambda); \{\!|c'; (f \mapsto \hat{w})^*|\!\}) \wedge c' \leq Thread$$
$$\implies T(\lambda, \{\!|c'; (f \mapsto \hat{w})^*|\!\})$$

- **(R-INTERRUPTWAIT)**

$$\begin{array}{c} \text{(R-INTERRUPTWAIT)} \\[4pt] H(\ell_r) = \{\!|\lambda_r; (f_r \mapsto u_r)^*, inte \mapsto true|\!\} \\ p_{c,m,pc} \notin dom(H) \qquad o = \{\!|c_r; (f_r \mapsto u_r)^*, inte \mapsto false|\!\} \\ \alpha = waiting(\_,\_) :: \alpha_0 \qquad o_e = \{\!|IntExcpt; |\!\} \\ \hline \Sigma \Downarrow \Sigma[\alpha \mapsto \texttt{AbNormal}(\alpha_0[r_{\text{excpt}} \mapsto \ell_e]), H \mapsto H[p_{c,m,pc} \mapsto o_e, \ell_r \mapsto o]] \end{array}$$

1) Let $pp = c, m, pc$. Let $G' = G[\ell_r \mapsto o] \cup \{(p_{c,m,pc} \mapsto o_e)\}$ and $((K'_i)_{i \leq n}, K', (lk'^j)_j) = ((K_i)_{i \leq n}, K, (lk^j)_j)$. Since $(G, (K_i)_i, K, (lk^j)_j)$ 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 $(G', (K'_i)_i, K', (lk'^j)_j)$ 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 **??**.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 **??** we know that for all $j \geq 2$:

$$\beta_{LstInv}^{\ell_r}(L_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta_{LstInv}^{\ell_r}(L_j, j, \_, K', (\mathsf{lk}'^i)_i)$$

One can then show that the following definitions satisfy the wanted property:

– $D_{Call} = \beta_{ALst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R[r_{\mathsf{except}} \mapsto p_{c,m,pc}]\rangle, K', (\mathsf{lk}'^n)_n))$

– $D_{Heap} = \{H(\beta_{Lab}(\ell_r), \beta_{Blk}(o))\} \cup \{H(\beta_{Lab}(p_{c,m,pc}), \beta_{Blk}(o_e))\}$

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:

$$\beta_{Lst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^n)_n) = \mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta \tag{33}$$

We define:

– $\Delta_{Call} = \mathsf{AState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*[\mathsf{except} \mapsto \mathsf{pp}]; \hat{h}'; \hat{k}')$

– Since $X <: \Delta$ and $\ell_r \in dom(G)$ we know that there exists $H(\lambda_r, \hat{b}) \in \Delta$ such that $H(\ell_r) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}$. This implies that $\hat{b} = \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}$ and that $(\beta_{Val}(u_r))^* \sqsubseteq_{Seq}^{\mathsf{nfs}} \hat{v}_r^*$ and $\beta_{Val}(true) \sqsubseteq^{\mathsf{nfs}} \hat{v}_i$. We define :

$$\Delta_{Heap} = \{H(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \widehat{false}|\!\})\} \cup \{H(\mathsf{pp}; \{\!|\mathsf{IntExcpt}; |\!\})\}$$

4) Show that:

– $D_{Call} <: \Delta_{Call}$: one can check that:

$$\beta_{ALst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R[r_{\mathsf{except}} \mapsto p_{c,m,pc}]\rangle, K', (\mathsf{lk}'^n)_n)) = \mathsf{AState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*[\mathsf{except} \mapsto \mathsf{pp}]; \hat{h}; \hat{k}) \tag{34}$$

From Equation (**??**) 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:

$$\mathsf{LState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*[\mathsf{except} \mapsto \mathsf{pp}]; \hat{h}; \hat{k}) \sqsubseteq_R \mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*[\mathsf{except} \mapsto \mathsf{pp}]; \hat{h}'; \hat{k}')$$

Hence by definition of $\sqsubseteq_A$ we have:

$$\mathsf{AState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}^*); \hat{v}^*[\mathsf{except} \mapsto \mathsf{pp}]; \hat{h}; \hat{k}) \sqsubseteq_A \mathsf{AState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*[\mathsf{except} \mapsto \mathsf{pp}]; \hat{h}'; \hat{k}')$$

Equation (**??**) and the above relation shows that $D_{Call} <: \Delta_{Call}$.

– $D_{Heap} <: \Delta_{Heap}$: we know that $(\beta_{Val}(u_r))^* \sqsubseteq_{Seq}^{\mathsf{nfs}} \hat{u}_r^*$. Besides $\beta_{Val}(false) \sqsubseteq^{\mathsf{nfs}} \widehat{false}$, therefore we have $\beta_{Blk}(o) \sqsubseteq_{Blk}^{\mathsf{nfs}} \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \widehat{false}|\!\})$, which in turn implies that :

$$\{H(\beta_{Lab}(\ell_r), \beta_{Blk}(o))\} <: \{H(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*|\!\})\} \subseteq \Delta_{Heap}$$

The fact that $\{H(\beta_{Lab}(\ell_r), \beta_{Blk}(o_e))\} <: \{H(\mathsf{pp}; \{\!|\mathsf{IntExcpt}; |\!\})\} \subseteq \Delta_{Heap}$ is trivial.

5) By definition of β_{Lst}, we get from Equation (**??**) that $\hat{\lambda}_t = \beta_{Val}(\ell_r) = \mathsf{NFS}(\lambda_r)$, and that $\hat{\lambda}_t = \hat{\lambda}'_t$. Besides we know that $H(\lambda_r, \hat{b}) \in \Delta$, where $\hat{b} = \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}$ and $\beta_{Val}(true) = \widehat{true} \sqsubseteq^{\mathsf{nfs}} \hat{v}_i$, which implies that $\widehat{true} \sqsubseteq \hat{v}_i$. Moreover Equation (**??**) gives us that $\mathsf{LState}_{c,m,pc}((\hat{\lambda}'_t, \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \in \Delta$, therefore we have :

$$\Delta \vdash \mathsf{LState}_{c,m,pc}((\mathsf{NFS}(\lambda_r), \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge H(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}) \wedge \widehat{true} \sqsubseteq \hat{v}_i \tag{35}$$

Since Σ is well-formed, and since $L_1 = \mathsf{waiting}(\_,\_)$ we know that $st_{pc} = \mathtt{wait}\ \_$. Therefore $(\!|P|\!)$ contains the following rules:

$$\mathsf{LState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_r), \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge H(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}) \wedge \widehat{true} \sqsubseteq \hat{v}_i$$
$$\implies \mathsf{AState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_r), \hat{u}'^*); \hat{v}'^*[\mathsf{except} \mapsto \mathsf{pp}]; \hat{h}'; \hat{k}') \tag{36}$$

$$\mathsf{LState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_r), \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge H(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}) \wedge \widehat{true} \sqsubseteq \hat{v}_i$$
$$\implies H(\lambda_r, \{\!|c'; (f \mapsto \hat{u})^*, \mathsf{inte} \mapsto \widehat{false}|\!\}) \tag{37}$$

$$\mathsf{LState}_{\mathsf{pp}}((\mathsf{NFS}(\lambda_r), \hat{u}'^*); \hat{v}'^*; \hat{h}'; \hat{k}') \wedge H(\lambda_r, \{\!|c_r; (f_r \mapsto \hat{u}_r)^*, \mathsf{inte} \mapsto \hat{v}_i|\!\}) \wedge \widehat{true} \sqsubseteq \hat{v}_i$$
$$\implies H(\mathsf{pp}; \{\!|\mathsf{IntExcpt}; |\!\}) \tag{38}$$

– $(\!|P|\!)\cup\Delta\vdash\Delta_{Call}$: this is trivially implied by Equation (**??**) and Equation (**??**).
– $(\!|P|\!)\cup\Delta\vdash\Delta_{Heap}$: Equation (**??**) and Equation (**??**) gives us that $(\!|P|\!)\cup\Delta\vdash\mathsf{H}(\lambda_r,\{\!|c';(f\mapsto\hat{u})^*,\mathsf{inte}\mapsto\widehat{\mathit{false}}|\!\})$, and abstract fact $\mathsf{H}(\mathsf{pp};\{\!|\mathsf{IntExcpt};|\!\})$ is obtained by Equation (**??**).

- **(R-Caught)**

<div align="center">

(R-Caught)

$$\dfrac{\begin{array}{cc}\ell=\Sigma[\![r_{\mathsf{excpt}}]\!] & H(\ell)=\{\!|c';(f\mapsto v)^*|\!\} \\ \mathsf{ExcptTable}(c,m,pc,c')=pc' & \alpha'=\langle c,m,pc'\cdot\_\cdot\_\cdot R\rangle::\alpha_0\end{array}}{\Sigma\Downarrow\Sigma[\alpha\mapsto\alpha']}$$

</div>

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 **??**.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 **??** 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:
$$\beta^{\ell_r}_{ALst}(\langle c,m,pc\cdot u^*\cdot st^*\cdot R\rangle,K,(\mathsf{lk}^n)_n)=\mathsf{AState}_{c,m,pc}((\hat{\lambda}_t,\hat{u}^*);\hat{v}^*;\hat{h};\hat{k})\sqsubseteq_A\mathsf{AState}_{c,m,pc}((\hat{\lambda}'_t,\hat{u}'^*);\hat{v}'^*;\hat{h}';\hat{k}')\in\Delta \tag{39}$$
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) 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 **??**. We know by Lemma **??** 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 (**??**) 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 $(\!|P|\!)$:
  $$\mathsf{AState}_{c,m,pc}(\hat{u}'^*;\hat{v}'^*;\hat{h}';\hat{k}')\wedge\mathsf{GetBlk}_{\mathsf{excpt}}(\hat{v}'^*;\hat{h}';\_;\{\!|c';(f\mapsto\hat{w})^*|\!\})\wedge c'\leq\mathsf{Throwable}$$
  $$\implies\mathsf{LState}_{c,m,pc'}(\hat{u}'^*;\hat{v}'^*;\hat{h}';\hat{k}')$$

  This concludes this case.

- **(R-Uncaught)**

<div align="center">

(R-Uncaught)

$$\dfrac{\begin{array}{cc} & \ell=\Sigma[\![r_{\mathsf{excpt}}]\!] \\ H(\ell)=\{\!|c_e;(f\mapsto v)^*|\!\} & \mathsf{ExcptTable}(c,m,pc,c_e)=\bot\end{array}}{\Sigma\Downarrow\Sigma[\alpha\mapsto\texttt{AbNormal}(\alpha_0[r_{\mathsf{excpt}}\mapsto\ell])]}$$

</div>

Here the call-stack is abnormal $\alpha=\texttt{AbNormal}(\langle c,m,pc\cdot u^*\cdot st^*\cdot R\rangle::\alpha_0)$. If α_0 is the empty list, then this case is easy. Hence we assume that :

$$\begin{aligned}\alpha &= \texttt{AbNormal}(\langle c,m,pc\cdot v^*\cdot st^*\cdot R\rangle::\langle c',m',pc'\cdot u'^*\cdot st'^*\cdot R'\rangle::\alpha_1) \\ \alpha' &= \texttt{AbNormal}(\langle c',m',pc'\cdot u'^*\cdot st'^*\cdot R'[r_{\mathsf{excpt}}\mapsto\ell]\rangle::\alpha_1)\end{aligned}$$

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 **??** we get for all $j \geq 1$:

$$\beta_{LstInv}^{\ell_r}(\alpha_j, j, \_, K, (\mathsf{lk}^i)_i) = \beta_{LstInv}^{\ell_r}(\alpha_j, j, \_, K', (\mathsf{lk}'^i)_i)$$

One can then check that:

$$D_{Call} = \beta_{ALst}^{\ell_r}(\langle c', m', pc' \cdot u'^* \cdot st'^* \cdot R'[r_{\mathsf{excpt}} \mapsto \ell]\rangle, K', (\mathsf{lk}'^j)_j)$$

3) We know that:

$$\begin{aligned}
\beta_{ALst}^{\ell_r}(\langle c, m, pc \cdot u^* \cdot st^* \cdot R\rangle, K, (\mathsf{lk}^j)_j) &= \mathsf{AState}_{c,m,pc}((\hat{\lambda}_t, \hat{u}_1^*); \hat{v}_1^*; \hat{h}_1; \hat{k}_1) \quad (40) \\
&\sqsubseteq_R \mathsf{AState}_{c,m,pc}((\hat{w}_1', \hat{u}_1'^*); \hat{v}_1'^*; \hat{h}_1'; \hat{k}_1') \in \Delta
\end{aligned}$$

$$\begin{aligned}
\beta_{LstInv}^{\ell_r}(\langle c', m', pc' \cdot u'^* \cdot st'^* \cdot R'\rangle, 2, c, K, (\mathsf{lk}^j)_j) &= \mathsf{Inv}_{c',m',pc'}^c((\hat{\lambda}_t, \hat{u}_2^*); \hat{v}_2^*; \hat{k}_2) \quad (41) \\
&\sqsubseteq_{Inv}^{\Delta} \mathsf{LState}_{c',m',pc'}((\hat{w}_2', \hat{u}_2'^*); \hat{v}_2'^*; \hat{h}_2'; \hat{k}_2') \in \Delta
\end{aligned}$$

Let $\Delta_{Call} = \mathsf{AState}_{c',m',pc'}((\hat{w}_2', \hat{u}_2'^*); \mathsf{lift}(\hat{v}_2'^*; \hat{k}_1')[\mathsf{except} \mapsto (\hat{v}_1'^*)_{\mathsf{excpt}}]; \hat{h}_1'; \hat{k}_1' \sqcup \hat{k}_2')$.

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 **??**. Therefore we have the following rule in $(\!|P|\!)$:

$$\begin{aligned}
\mathsf{AState}_{c,m,pc}((\hat{w}_1', \hat{u}_1'^*); \hat{v}_1'^*; \hat{h}_1'; \hat{k}_1') \wedge \mathsf{GetBlk}_{\mathsf{except}}(\hat{v}_1'^*; \hat{h}_1'; \_; \{\!|c_e; \_|\!\}) \wedge c_e \leq \mathsf{Throwable} \\
\implies \mathsf{Uncaught}_{c,m}((\hat{w}_1', \hat{u}_1'^*); (\hat{v}_1'^*)_{\mathsf{excpt}}; \hat{h}_1'; \hat{k}_1')
\end{aligned}$$

As it was done in (R-CAUGHT), one can show that:

$$\Delta \vdash \mathsf{GetBlk}_{\mathsf{except}}(\hat{v}_1'^*; \hat{h}_1'; \_; \{\!|c_e; \_|\!\}) \wedge c_e \leq \mathsf{Throwable}$$

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 $sign(c', m') = (\tau_i)_{i \leq n} \xrightarrow{loc} \tau$, $st'_{pc'} = \texttt{invoke } r_o \ m \ (r_{j_i})_{i \leq n}$ and $u^* = (R'(r_{j_i}))_{i \leq n}$. By using the same reasoning that we did in (R-RETURN) we can show that:

$$\Delta \vdash \mathsf{GetBlk}_o(\hat{v}_2'^*; \hat{h}_2'; \_; \{\!|c''; \_|\!\}) \wedge c'' \leq c' \wedge \hat{w}_1' = \hat{w}_2' \wedge \left(\bigwedge_{j \leq n} (\hat{v}_2'^*)_{i_j} \sqcap (\hat{u}_1'^*)_j \neq \bot\right)$$

Hence we can apply the following rule, which is included in $(\!|P|\!)$:

$$\begin{aligned}
\mathsf{LState}_{c',m',pc'}((\hat{w}_2', \hat{u}_2'^*); \hat{v}_2'^*; \hat{h}_2'; \hat{k}_2') \wedge \mathsf{GetBlk}_o(\hat{v}_2'^*; \hat{h}_2'; \_; \{\!|c''; \_|\!\}) \wedge c'' \leq c' \\
\wedge \mathsf{Uncaught}_{c,m}((\hat{w}_1', \hat{u}_1'^*); (\hat{v}_1'^*)_{\mathsf{excpt}}; \hat{h}_1'; \hat{k}_1') \wedge \hat{w}_1' = \hat{w}_2' \wedge \left(\bigwedge_{j \leq n} (\hat{v}_2'^*)_{i_j} \sqcap (\hat{u}_1'^*)_j \neq \bot\right) \\
\implies \mathsf{LState}_{c',m',pc'}((\hat{w}_2', \hat{u}_2'); \mathsf{lift}(\hat{v}_2'^*; \hat{k}_1')[\mathsf{excpt} \mapsto (\hat{v}_1'^*)_{\mathsf{excpt}}]; \hat{h}_1'; \hat{k}_1' \sqcup \hat{k}_2')
\end{aligned}$$

This shows that $(\!|P|\!) \cup \Delta \vdash \Delta_{Call}$.

- **Remaining cases** The remaining cases are straightforward or very similar to cases we already analyzed. 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

∎

L. Proof of Lemma **??**

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):

$$
\begin{array}{c}
\text{(A-ACTIVE)} \\[4pt]
\dfrac{\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S \leadsto \ell \cdot \alpha' \cdot \pi' \cdot \gamma' \cdot H' \cdot S'}{\Omega :: \underline{\langle \ell, s, \pi, \gamma, \alpha \rangle} :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \underline{\langle \ell, s, \pi', \gamma', \alpha' \rangle} :: \Omega' \cdot \Xi \cdot H' \cdot S'}
\end{array}
$$

We know that:

$$
X = \beta_{Stk}^G(\Omega :: \underline{\langle \ell, s, \pi, \gamma, \alpha \rangle} :: \Omega', \Xi, (K_l, (\mathsf{lk}^{l,j})_j)_l) \cup \beta_{Heap}^G(H) \cup \beta_{Stat}(S)
$$

and that :

$$
\beta_{Frm}^G(\langle \ell, s, \pi, \gamma, \alpha \rangle, K_n, (\mathsf{lk}^{n,j})_j) \subseteq \beta_{Stk}^G(\Omega :: \underline{\langle \ell, s, \pi, \gamma, \alpha \rangle} :: \Omega', \Xi, (K_l, (\mathsf{lk}^{l,j})_j)_l)
$$

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:

$$
\begin{aligned}
X_{loc} &= \beta_{Frm}^G(\langle \ell, s, \pi, \gamma, \alpha \rangle, K_n, (\mathsf{lk}^{n,j})_j) \cup \beta_{Heap}^G(H) \cup \beta_{Stat}(S) \\
&= \beta_{Call}^\ell(\alpha, K_n, (\mathsf{lk}^{n,j})_j) \cup \beta_{Pact}^\ell(\pi) \cup \beta_{Pthr}^G(\gamma) \cup \beta_{Heap}^G(H) \cup \beta_{Stat}(S) \\
&\in \beta_{Lcnf}(\ell \cdot \alpha \cdot \pi \cdot \gamma \cdot H \cdot S)
\end{aligned}
$$

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 **??** we have $X_{loc} <: \Delta$. By Lemma **??** 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:

$$
X' = \beta_{Stk}^{G'}(\Omega :: \underline{\langle \ell, s, \pi', \gamma', \alpha' \rangle} :: \Omega', \Xi, (K'_l, (\mathsf{lk}'^{l,j})_j)_l) \cup \beta_{Heap}^{G'}(H') \cup \beta_{Stat}(S')
$$

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:

$$
\beta_{Stk}^{G'}(\Omega :: \underline{\langle \ell, s, \pi', \gamma', \alpha' \rangle} :: \Omega', \Xi, (K'_l, (\mathsf{lk}'^{l,j})_j)_l) \backslash \beta_{Frm}^{G'}(\underline{\langle \ell, s, \pi', \gamma', \alpha' \rangle}, K'_n, (\mathsf{lk}'^{n,j})_j)
$$

$$
= \left(\bigcup_{l=1}^{n-1} \beta_{Frm}^{G'}(\Omega_l, K'_l, (\mathsf{lk}'^{l,j})_j) \right) \cup \left(\bigcup_{l=1}^{n'} \beta_{Frm}^{G'}(\Omega'_l, K'_{l+n}, (\mathsf{lk}'^{l+n,j})_j) \right) \cup \left(\bigcup_{l=1}^{m} \beta_{Frm}^{G'}(\Xi_l, K'_{l+n+n'}, (\mathsf{lk}'^{l+n+n',j})_j) \right)
$$

which by Proposition **??** is equal to

$$
= \left(\bigcup_{l=1}^{n-1} \beta_{Frm}^{G}(\Omega_l, K_l, (\mathsf{lk}^{l,j})_j) \right) \cup \left(\bigcup_{l=1}^{n'} \beta_{Frm}^{G}(\Omega'_l, K_{l+n}, (\mathsf{lk}^{l+n,j})_j) \right) \cup \left(\bigcup_{l=1}^{m} \beta_{Frm}^{G}(\Xi_l, K_{l+n+n'}, (\mathsf{lk}^{l+n+n',j})_j) \right)
$$

Which implies that:

$$
X' \backslash X \subseteq \beta_{Frm}^{G'}(\langle \ell, s, \pi', \gamma', \alpha' \rangle, K'_n, (\mathsf{lk}'^{n,j})_j) \cup \beta_{Heap}^{G'}(H') \cup \beta_{Stat}(S') = X'_{loc}
$$

We define $\Delta' = \Delta \cup \Delta'_{loc}$. We know that $X <: \Delta$ and $X'_{loc} <: \Delta'_{loc}$, therefore by Lemma **??** we have $X \cup X'_{loc} <: \Delta \cup \Delta'_{loc} = \Delta'$. Moreover $X' \subseteq X \cup X'_{loc}$, therefore by Lemma **??** 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):

$$
\begin{array}{c}
\text{(A-DEACTIVATE)} \\[4pt]
\dfrac{}{\Omega :: \underline{\langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle} :: \Omega' \cdot \Xi \cdot H \cdot S \Rightarrow \Omega :: \langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle :: \Omega' \cdot \Xi \cdot H \cdot S}
\end{array}
$$

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):

(A-STEP)

$$\frac{(s, s') \in \textit{Lifecycle} \qquad \pi \neq \varepsilon \Rightarrow (s, s') = (\textit{running}, \textit{onPause})}{\langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle :: \Omega \cdot \Xi \cdot H \cdot S \Rightarrow \underline{\langle \ell, s', \pi, \gamma, \alpha_{\ell.s'} \rangle} :: \Omega \cdot \Xi \cdot H \cdot S}$$

We have:

$$X = \beta_{Stk}^G(\underline{\langle \ell, s, \pi, \gamma, \overline{\alpha} \rangle} :: \Omega, \Xi, (K_l, (\mathsf{lk}^{l,j})_j)_l) \cup \beta_{Heap}^G(H) \cup \beta_{Stat}(S)$$

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 $(c', st^*) = lookup(c, m)$ 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 $\tau_1, \ldots, \tau_n$. By Assumption **??**, we also have $c \leq c'$.

Given that $\Delta :> X \in \beta_{Cnf}(\Psi)$, we have $\Delta :> \beta_{Heap}^G(H)$. 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 $\beta_{Blk}(\{|c; (f \mapsto u)^*|\}) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}$. 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:

$$(|P|) \cup \Delta \vdash \mathsf{LState}_{c',m,0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*) \quad (42)$$

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 **??** 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 $(\mathsf{lk}^{\prime 1,j})_j = (\{|(\ell \mapsto 0) \mid \ell|\}) :: \varepsilon$. Using Assumption **??**, it is simple to show that $(G', (K_i', (\mathsf{lk}^{\prime i,j})_j)_i)$ is a configuration decomposition of $\underline{\langle \ell, s', \pi, \gamma, \alpha_{\ell.s'} \rangle} :: \Omega \cdot \Xi \cdot H \cdot S$, and that:

$$\Delta' >: \{\mathsf{LState}_{c',m,0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*)\} :> \beta_{Call}^\ell(\alpha_{\ell.s'}, K_1', (\mathsf{lk}^{\prime 1,j})_j) \quad (43)$$

Observe that $\beta_{Pthr}^G(\gamma) = \beta_{Pthr}^{G'}(\gamma)$. Besides $\Delta :> \beta_{Cnf}(\Omega \cdot \Xi \cdot H \cdot S)$ implies that $\beta_{Pact}^G(\pi) \cup \beta_{Pthr}^G(\gamma) <: \Delta$, and we know that since $\Delta \subseteq \Delta'$ we have $\Delta <: \Delta'$. Therefore by transitivity of $<:$ we have :

$$\beta_{Pact}^\ell(\pi) \cup \beta_{Pthr}^{G'}(\gamma) <: \Delta' \quad (44)$$

It is easy to check that $X' \in \beta_{Cnf}(\Psi')$, where X' is the following set of facts:

$$X' = \beta_{Stk}^{G'}(\underline{\langle \ell, s', \pi, \gamma, \alpha_{\ell.s'} \rangle} :: \Omega, \Xi, (K_l', (\mathsf{lk}^{\prime l,j})_j)_l) \cup \beta_{Heap}^G(H) \cup \beta_{Stat}(S)$$

Using Proposition **??**, one can check that:

$$X' \backslash X = \beta_{Call}^\ell(\alpha_{\ell.s'}, K_1', (\mathsf{lk}^{\prime 1,j})_j) \cup \beta_{Pact}^\ell(\pi) \cup \beta_{Pthr}^{G'}(\gamma)$$

Equation **??** and Equation **??** 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)

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

This case is analogous to the case (A-STEP).

- Rule applied is (A-DESTROY):

(A-DESTROY)

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

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 $X' \in \beta_{Cnf}(\Psi')$ where:

$$X' = \beta_{Stk}^{G \cup K_n}(\Omega :: \Omega', \Xi, (K_l, (\mathsf{lk}^{l,j})_j)_{l \neq n}) \cup \beta_{Heap}^G(H) \cup \beta_{Stat}(S) \subseteq X$$

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):

(A-BACK)

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

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 $b' = H'(\ell) = \{\!|c; (f \mapsto u)^*, \mathsf{finished} \mapsto true|\!\}$ according to the reduction rule.

Given that $\Delta :> X \in \beta_{Cnf}(\Psi)$, we have $\Delta :> \beta_{Heap}^G(H)$. 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 $\beta_{Blk}(\{\!|c; (f \mapsto u)^*, \mathsf{finished} \mapsto v|\!\}) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}$. 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:

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

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:

$$\mathsf{H}(\beta_{Lab}(\ell), \beta_{Blk}(b')) \quad \sqsubseteq_{Blk}^{\mathsf{nfs}} \quad \mathsf{H}(\lambda, \{\!|c; (f \mapsto \hat{u})^*, \mathsf{finished} \mapsto \widehat{true}|\!\})$$
$$\sqsubseteq_{Blk}^{\mathsf{nfs}} \quad \mathsf{H}(\lambda, \{\!|c; (f \mapsto \hat{u})^*, \mathsf{finished} \mapsto \top_{\mathtt{bool}}|\!\})$$

Hence $\beta_{Heap}^G(H') <: \Delta'$. It is then easy to conclude this case.

- Rule applied is (A-SWAP):

(A-SWAP)

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

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):

(A-START)

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

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 $\alpha_{p_c.constructor} = \langle c', m, 0 \cdot v^* \cdot st^* \cdot R\rangle :: \varepsilon$, where $(c', st^*) = lookup(c, constructor)$, $sign(c', constructor) = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ 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 $v'_1, \ldots, v'_n$ of the correct type $\tau_1, \ldots, \tau_n$. By Assumption **??**, we also have $c \leq c'$.

Given that $X <: \Delta$, we have $\Delta :> \beta_{Pact}^\ell(i :: \pi)$, which implies that there exists $\mathsf{I}_\lambda(\hat{b}) \in \Delta$ such that $\lambda = \beta_{Lab}(\ell) = c'$ and $\beta_{Blk}(i) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}$. This implies that $\hat{b} = \{\!|@c; \hat{v}|\!\}$ for some $\hat{v}$ such that $\sqcup_i \beta_{Val}(v_i) \sqsubseteq^{\mathsf{nfs}} \hat{v}$. Using the implications *Act* in $(\!|P|\!)$ we get:

$$(\!|P|\!) \cup \Delta \quad \vdash \quad \mathsf{H}(in(c), \{\!|@c; \hat{v}|\!\}) \tag{45}$$

$$(\!|P|\!) \cup \Delta \quad \vdash \quad \mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{false}, \mathsf{parent} \mapsto c', \mathsf{intent} \mapsto in(c)|\!\}) \tag{46}$$

Hence using the implications *Cbk* included in $(\!|P|\!)$ we get that:

$$(\!|P|\!) \cup \{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{\mathit{false}}, \mathsf{parent} \mapsto c', \mathsf{intent} \mapsto in(c)|\!\})\}$$
$$\vdash \mathsf{LState}_{c',m,0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*) \quad (47)$$

We define the set of abstract fact:

$$\Delta' = \Delta \cup \{\mathsf{LState}_{c',m,0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*))\} \cup \{\mathsf{H}(in(c), \{\!|@c; \hat{v}|\!\})\}$$
$$\cup \{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{\mathit{false}}, \mathsf{parent} \mapsto c', \mathsf{intent} \mapsto in(c)|\!\})\}$$

From Equation **??**, Equation **??** and Equation **??** we get that $(\!|P|\!) \cup \Delta \vdash \Delta'$.

a) Configuration Decomposition: Let K_0' be an fresh empty local heap. We take $G' = G \cup H' \cup \{p_c, p'_{in(c)}\}$, $(K_l')_l = K_0' :: (K)_l$ 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 **??** 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, $o \not\rightarrow_{\mathsf{ref}} K_i$. By Lemma **??** we know that for all i, $i \not\rightarrow_{\mathsf{ref}} K_i$. 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 **??**, 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:

$$\beta^{G'}_{Stk}(\underline{\langle p_c, \mathit{constructor}, \varepsilon, \varepsilon, \alpha_{p_c.\mathit{constructor}}\rangle} :: \langle \ell, s, \pi, \gamma, \overline{\alpha}\rangle :: \Omega, \Xi, (K_l', (\mathsf{lk}'^{l,j})_j)_l) \cup \beta^{G'}_{Heap}(H'') \cup \beta_{Stat}(S)$$

We are going to prove that X' is over-approximated by the set of abstract facts Δ'.

b) Heap: We already saw that $\Delta :> \beta^G_{Heap}(H')$, and by applying Lemma **??** we know that $\beta_{Blk}(i) = \beta_{Blk}(i')$. We then observe that:

$$\begin{array}{llll}
\{\mathsf{H}(in(c), \{\!|@c; \hat{v}|\!\})\} & :> & \{\mathsf{H}(in(c), \beta_{Blk}(i))\} & \text{since } \beta_{Blk}(i) \sqsubseteq^{\mathsf{nfs}}_{Blk} \hat{b} = \{\!|@c; \hat{v}|\!\} \\
& = & \{\mathsf{H}(in(c), \beta_{Blk}(i'))\} & \text{since } \beta_{Blk}(i) = \beta_{Blk}(i') \\
& = & \{\mathsf{H}(\beta_{Lab}(p'_{in(c)}), \beta_{Blk}(i'))\} & \text{by definition}
\end{array} \quad (48)$$

Also notice that:

$$\{\mathsf{H}(c, \{\!|c; (f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{\mathit{false}}, \mathsf{parent} \mapsto c', \mathsf{intent} \mapsto in(c)|\!\})\} = \mathsf{H}(\beta_{Lab}(p_c), \beta_{Blk}(o)) \quad (49)$$

Moreover it is simple to see that we have:

$$\beta^{G'}_{Heap}(H'') = \beta^G_{Heap}(H) \cup \beta^{G \cup H'}_{Heap}(H') \cup \{\mathsf{H}(\beta_{Lab}(p_c), \beta_{Blk}(o))\} \cup \{\{\mathsf{H}(\beta_{Lab}(p'_{in(c)}), \beta_{Blk}(i'))\}\}$$

We already saw that $\beta^{G \cup H'}_{Heap}(H') <: \Delta <: \Delta'$. This together with Equation **??** and Equation **??** shows that $\beta^{G'}_{Heap}(H'') <: \Delta'$.

c) Activity Stack: Let n be the length of Ω, and let m be the length of Ξ.

$$\beta^{G'}_{Stk}(\underline{\langle p_c, \mathit{constructor}, \varepsilon, \varepsilon, \alpha_{p_c.\mathit{constructor}}\rangle} :: \langle \ell, s, \pi, \gamma, \overline{\alpha}\rangle :: \Omega, \Xi, (K_l', (\mathsf{lk}'^{l,j})_j)_l)$$
$$= \beta^{G'}_{Frm}(\underline{\langle p_c, \mathit{constructor}, \varepsilon, \varepsilon, \alpha_{p_c.\mathit{constructor}}\rangle}, K_0', (\mathsf{lk}'^{0,j})_j) \cup \beta^{G'}_{Frm}(\langle \ell, s, \pi, \gamma, \overline{\alpha}\rangle, K_1', (\mathsf{lk}'^{1,j})_j)$$
$$\cup \left(\bigcup_{1 \leq l \leq n} \beta^{G'}_{Frm}(\Omega_l, K_{l+1}', (\mathsf{lk}'^{l+1,j})_j) \right) \cup \left(\bigcup_{1 \leq l \leq m} \beta^{G'}_{Frm}(\Xi_l, K_{l+n+1}', (\mathsf{lk}'^{l+n+1,j})_j) \right)$$

By Proposition **??** this is equal to:

$$\beta^{G'}_{Frm}(\langle p_c, \mathit{constructor}, \varepsilon, \varepsilon, \alpha_{p_c.\mathit{constructor}}\rangle, K_0', (\mathsf{lk}'^{0,j})_j) \cup \beta^G_{Frm}(\langle \ell, s, \pi, \gamma, \overline{\alpha}\rangle, K_1, (\mathsf{lk}^{1,j})_j)$$
$$\cup \left(\bigcup_{1 \leq l \leq n} \beta^G_{Frm}(\Omega_l, K_{l+1}, (\mathsf{lk}^{l+1,j})_j) \right) \cup \left(\bigcup_{1 \leq l \leq m} \beta^G_{Frm}(\Xi_l, K_{l+n+1}, (\mathsf{lk}^{l+n+1,j})_j) \right)$$

We then observe that:

$$\Delta' \;:>\; \{\mathsf{LState}_{c',\mathsf{m},0}((\mathsf{NFS}(c),(\top_{\tau_j})^{j\leq n});(\hat{\mathbf{0}}_k)^{k\leq loc}, \mathsf{NFS}(c),(\top_{\tau_j})^{j\leq n};(\bot)^*;0^*)\}$$

$$:>\; \beta^{G'}_{Frm}(\underline{\langle p_c, constructor, \varepsilon, \varepsilon, \alpha_{p_c.constructor}\rangle}, K'_0, (\mathsf{lk}'^{0,j})_{0,j})$$

This proves that the changes to the activity stack are over-approximated by Δ'.

- Rule applied is (A-REPLACE):

  (A-REPLACE)

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

Since we only focus on well-formed configurations, we know that c is an activity class and $\ell = p'_c$ for some pointer p'.

We then observe that $\alpha_{p_c.constructor} = \langle c', m, 0 \cdot v^* \cdot st^* \cdot R\rangle :: \varepsilon$, where $(c', st^*) = lookup(c, constructor)$, $sign(c', constructor) = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ 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 $v'_1, \ldots, v'_n$ of the correct type $\tau_1, \ldots, \tau_n$. By Assumption **??**, we also have $c \leq c'$.

Given that $\Delta \;:>\; X \in \beta_{Cnf}(\Psi)$, we have $\Delta \;:>\; \beta^G_{Heap}(H)$. We know that $\ell = p'_c \in dom(H)$, 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*[5] included in $(\!|P|\!)$ we get that:

$$(\!|P|\!) \cup \Delta \vdash \mathsf{LState}_{c',\mathsf{m},0}((\mathsf{NFS}(c),(\top_{\tau_j})^{j\leq n});(\hat{\mathbf{0}}_k)^{k\leq loc}, \mathsf{NFS}(c),(\top_{\tau_j})^{j\leq n};(\bot)^*;0^*) \tag{50}$$

$$(\!|P|\!) \cup \Delta \vdash \mathsf{H}(c, \{\!|c;(f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{false}|\!\})) \tag{51}$$

We define the set of abstract Δ' by:

$$\Delta' = \Delta \;\;\cup\;\; \Big\{\mathsf{LState}_{c',\mathsf{m},0}((\mathsf{NFS}(c),(\top_{\tau_j})^{j\leq n});(\hat{\mathbf{0}}_k)^{k\leq loc}, \mathsf{NFS}(c),(\top_{\tau_j})^{j\leq n};(\bot)^*;0^*)\Big\}$$

$$\cup\;\; \Big\{\mathsf{H}(c, \{\!|c;(f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{false}|\!\}))\Big\}$$

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 **??**, 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:

$$\beta^\ell_{Call}(\alpha_{p_c.constructor}, K'_1, (\mathsf{lk}'^{1,j})_j) <: \{\mathsf{LState}_{c',\mathsf{m},0}((\mathsf{NFS}(c),(\top_{\tau_j})^{j\leq n});(\hat{\mathbf{0}}_k)^{k\leq loc}, \mathsf{NFS}(c),(\top_{\tau_j})^{j\leq n};(\bot)^*;0^*)\} <: \Delta' \tag{52}$$

Observe that $\beta^\ell_{Pthr}(\gamma) = \beta^{G'}_{Pthr}(\gamma)$. 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 :

$$\beta^\ell_{Pact}(\pi) \cup \beta^{G'}_{Pthr}(\gamma) <: \Delta' \tag{53}$$

Moreover:

$$\begin{aligned}
\beta^{G'}_{Heap}(H') &= \beta^G_{Heap}(H) \;\cup\; \mathsf{H}(\beta_{Lab}(p_c), \beta_{Blk}(o)) \\
&= \beta^G_{Heap}(H) \;\cup\; \mathsf{H}(c, \beta_{Blk}(\{\!|c;(f_\tau \mapsto \mathbf{0}_\tau)^*, \mathsf{finished} \mapsto false|\!\})) \\
&<: \quad \Delta \quad\;\; \cup\; \mathsf{H}(c, \{\!|c;(f \mapsto \hat{\mathbf{0}}_\tau)^*, \mathsf{finished} \mapsto \widehat{false}|\!\})) \\
&<: \quad \Delta' \tag{54}
\end{aligned}$$

It is easy to check that $X' \in \beta_{Cnf}(\Psi')$, where X' is the following set of facts:

$$X' = \beta^{G'}_{Stk}(\underline{\langle \ell, s', \pi, \gamma, \alpha_{p_c.constructor}\rangle} :: \Omega, \Xi, (K'_l, (\mathsf{lk}'^{l,j})_j)_l) \cup \beta^{G'}_{Heap}(H') \cup \beta_{Stat}(S)$$

---

[5]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*.

Using Proposition **??** 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 **??**, Equation **??** and Equation **??** 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)

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

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 $H(\ell') = \{\!| c'; (f' \mapsto \hat{v}')^*, \mathsf{parent} \mapsto \ell, result \mapsto w |\!\}$. We then observe that $\alpha_{p_c.onActivityResult} = \langle c'', m, 0 \cdot v^* \cdot st^* \cdot R \rangle :: \varepsilon$, where $(c'', st^*) = lookup(c, onActivityResult)$, $sign(c'', onActivityResult) = \tau_1, \ldots, \tau_n \xrightarrow{loc} \tau$ 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 $v'_1, \ldots, v'_n$ of the correct type $\tau_1, \ldots, \tau_n$. By Assumption **??**, we also have $c \leq c''$.

Given that $\Delta :> X \in \beta_{Cnf}(\Psi)$, we have $\Delta :> \beta^G_{Cnf}(H)$. 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:

$$(\!| P |\!) \cup \Delta \vdash \mathsf{LState}_{c'',m,0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*) \tag{55}$$

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 $\hat{b}' = \{\!| c'; (f' \mapsto \hat{v}')^*, \mathsf{parent} \mapsto c, result \mapsto \hat{w} |\!\}$ 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

$$(\!| P |\!) \cup \Delta \vdash \mathsf{H}(c, \{\!| c; (f \mapsto \hat{v})^* [result \mapsto \hat{w}] |\!\}) \tag{56}$$

We define the following set of facts:

$$\Delta' = \Delta \cup \{\mathsf{LState}_{c'',m,0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*)\} \cup \{\mathsf{H}(c, \{\!| c; (f \mapsto \hat{v})^* [result \mapsto \hat{w}] |\!\})\}$$

Equation **??** and Equation **??** 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 **??** 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 **??** 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 **??**, 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:

$$\beta^{G'}_{Heap}(H'') = \beta^G_{Heap}(H_{|dom(H) \backslash \ell}) \cup \beta^{G \cup H'}_{Heap}(H') \cup \mathsf{H}(c, \beta_{Blk}(H(\ell)[result \mapsto w']]))$$

$$\mathsf{H}(c, \beta_{Blk}(H(\ell)[result \mapsto w']])) \;=\; \mathsf{H}(c, \beta_{Blk}(H(\ell))[result \mapsto \beta_{Val}(w')]]))$$
$$=\; \mathsf{H}(c, \beta_{Blk}(H(\ell))[result \mapsto \beta_{Val}(w)]])) \qquad \text{(by lemma \textbf{??})}$$
$$<:\; \mathsf{H}(c, \hat{b}[result \mapsto \hat{w}])) \qquad\qquad \text{(by Proposition \textbf{??})}$$
$$<:\; \Delta' \tag{57}$$

We already saw that $\beta_{Heap}^{G \cup \mathsf{H}'}(H') <: \Delta <: \Delta'$. Moreover $\beta_{Heap}^{G}(H_{|dom(H) \setminus \ell}) \subseteq \beta_{Heap}^{G}(H) <: \Delta <: \Delta'$. These two fact and Equation **??** show that $\beta_{Heap}^{G'}(H'') <: \Delta'$. We can also check that:

$$\beta_{Frm}^{G'}(\underline{\langle \ell, s, \varepsilon, \gamma, \alpha_{\ell.onActivityResult} \rangle}, K_1', (\mathsf{lk}'^{1,j})_j)$$
$$<:\; \mathsf{LState}_{c'',m,0}((\mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(c), (\top_{\tau_j})^{j \leq n}; (\bot)^*; 0^*) <: \Delta'$$

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

Given that $X <: \Delta$, we have $\Delta :> \beta_{Pthr}^{G}(\ell'' :: \gamma)$. Moreover $H(\ell'') = \{\!| c'; (f \mapsto v)^* |\!\}$, therefore there exists $\mathsf{T}(\lambda, \hat{b}) \in \Delta$ such that $\lambda = \beta_{Lab}(\ell'')$ and $\beta_{Blk}(\{\!| c'; (f \mapsto v)^* |\!\}) \sqsubseteq_{Blk}^{\mathsf{nfs}} \hat{b}$. 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 **??** 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 $(\!| P |\!)$:

$$\mathsf{T}(\lambda, \{\!| c'; (f \mapsto \_)^* |\!\}) \wedge c' \leq \mathsf{c}'' \wedge c' \leq \mathsf{Thread} \implies \mathsf{LState}_{c'',\mathsf{run},0}((\mathsf{NFS}(\lambda), \mathsf{NFS}(\lambda)); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(\lambda); (\bot)^*; 0^*)$$
$$\tag{58}$$

We define the set of abstract fact:

$$\Delta' \;=\; \Delta \cup \{\mathsf{LState}_{c'',\mathsf{run},0}((\mathsf{NFS}(\lambda), \mathsf{NFS}(\lambda)); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(\lambda); (\bot)^*; 0^*)\}$$

From Equation **??** 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 :

$$(K_l', (\mathsf{lk}'^{l,j})_j)_{l \leq n+m+1} = (K_l, (\mathsf{lk}^{l,j})_j)_{l \leq n} :: (K_t', ((\{(\ell \mapsto 0) \mid \ell\}) :: \varepsilon)) :: (K_l, (\mathsf{lk}^{l,j})_j)_{n+1 \leq l \leq n+m}$$

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:

$$\beta_{Stk}^{G'}(\Omega :: \varphi' :: \Omega', \psi :: \Xi, (K_l', (\mathsf{lk}'^{l,j})_j)_l) \cup \beta_{Heap}^{G'}(H) \cup \beta_{Stat}(S)$$

Let n_0 be such that Ω is of length $n_0 - 1$. It is quite easy to check that:

$$X' \backslash X \subseteq \beta_{Frm}^{G'}(\underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}, K_{n_0}', (\mathsf{lk}'^{n_0,j})_j) \cup \beta_{Frm}^{G'}(\langle\!\langle \ell, \ell'', \varepsilon, \varepsilon, \alpha' \rangle\!\rangle, K_{n+1}', (\mathsf{lk}'^{n+1,j})_j)$$

Since $\ell'' \in dom(G)$, we have that:

$$\Delta' \;:>\; \{\mathsf{LState}_{c'',\mathsf{run},0}((\mathsf{NFS}(\lambda), \mathsf{NFS}(\lambda)); (\hat{\mathbf{0}}_k)^{k \leq loc}, \mathsf{NFS}(\lambda); (\bot)^*; 0^*)\}$$
$$:>\; \beta_{Frm}^{G'}(\langle\!\langle \ell, \ell'', \varepsilon, \varepsilon, \alpha' \rangle\!\rangle, K_{n+1}', (\mathsf{lk}'^{n+1,j})_j)$$

Moreover since ϕ' only differ from ϕ in the fact that it has a smaller thread stack, we have:

$$\beta_{Frm}^{G'}(\langle \ell, s, \pi, \gamma, \alpha \rangle, K_{n_0}', (\mathsf{lk}'^{n_0,j})_j) \subseteq \beta_{Frm}^{G}(\underline{\langle \ell, s, \pi, \ell'' :: \gamma, \alpha \rangle}, K_{n_0}, (\mathsf{lk}^{n_0,j})_j) <: \Delta$$

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

  Exactly like the (A-DESTROY) case.

- Rule applied is (T-INTENT):

  (T-INTENT)
  $$\frac{(\varphi, \varphi') \in \{(\langle \ell, s, \pi, \gamma, \alpha \rangle, \langle \ell, s, i :: \pi, \gamma, \alpha \rangle), (\underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}, \underline{\langle \ell, s, i :: \pi, \gamma, \alpha \rangle})\}}{\Omega :: \varphi :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', i :: \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega :: \varphi' :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S}$$

  Trivial since there are no changes to the abstraction: $\beta_{Cnf}(\Psi) = \beta_{Cnf}(\Psi')$.

- Rule applied is (T-THREAD):

  (T-THREAD)
  $$\frac{(\varphi, \varphi') \in \{(\langle \ell, s, \pi, \gamma, \alpha \rangle, \langle \ell, s, \pi, \ell_t :: \gamma, \alpha \rangle), (\underline{\langle \ell, s, \pi, \gamma, \alpha \rangle}, \underline{\langle \ell, s, \pi, \ell_t :: \gamma, \alpha \rangle})\}}{\Omega :: \varphi :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \ell_t :: \gamma', \alpha' \rangle\!\rangle :: \Xi' \cdot H \cdot S \Rightarrow \Omega :: \varphi' :: \Omega' \cdot \Xi :: \langle\!\langle \ell, \ell', \pi', \gamma', \alpha' \rangle\!\rangle :: \Xi \cdot H \cdot S}$$

  Trivial since there are no changes to the abstraction: $\beta_{Cnf}(\Psi) = \beta_{Cnf}(\Psi')$.

  $\blacksquare$