A Certified Lightweight Non-Interference Java Bytecode Verifier

Gilles Barthe, David Pichardie, Tamara Rezk

A Certified Lightweight Non-Interference Java Bytecode Verifier*

Gilles Barthe and David Pichardie and Tamara Rezk

INRIA Sophia Antipolis, France
{Gilles.Barthe,David.Pichardie,Tamara.Rezk}@sophia.inria.fr

Abstract. Non-interference is a semantical condition on programs that guarantees the absence of illicit information flow throughout their execution, and that can be enforced by appropriate information flow type systems. Much of previous work on type systems for non-interference has focused on calculi or high-level programming languages, and existing type systems for low-level languages typically omit objects, exceptions, and method calls, and/or do not prove formally the soundness of the type system. We define an information flow type system for a sequential JVM-like language that includes classes, objects, arrays, exceptions and method calls, and prove that it guarantees non-interference. For increased confidence, we have formalized the proof in the proof assistant Coq; an additional benefit of the formalization is that we have extracted from our proof a certified lightweight bytecode verifier for information flow. Our work provides, to our best knowledge, the first sound and implemented information flow type system for such an expressive fragment of the JVM.

1 Introduction

Java security. The Java security architecture combines static and dynamic mechanisms to enforce innocuity of applications; in particular, it features a bytecode verifier that guarantees statically safety properties such as the absence of arithmetic on references, and a stack inspection mechanism that performs access control verifications. However, it lacks of appropriate mechanisms to guarantee stronger confidentiality properties: for example, it has been suggested that the Java security model is not sufficient in security-sensitive applications such as smart cards [18, 23]. One weakness of the model is that it only concentrates on who accesses sensitive information, but not how sensitive information flows through programs.

Language-based security. The goal of language-based security [30] is to provide enforcement mechanisms for end-to-end security policies that go beyond the basic isolation properties ensured by security models for mobile code. In contrast to security models based on access control, language-based security focuses on information-flow policies that track how sensitive information is propagated during execution.

Starting from the seminal work of Volpano and Smith [34], type systems have become a prominent approach for a practical enforcement of information flow policies, and recent research [30] has proposed type-based enforcement mechanisms for advanced programming features such as exceptions, objects [9, 4], interactions [**?**], concurrency [32] and distribution [21]. This line of work has culminated in the design and implementation of information flow type systems for programming languages such as Java [25] and Caml [28].

Our contribution. It is striking to notice that, although mobile code security is the central motivation behind those works, there has been very little effort to study information flow in low-level languages such as Java bytecode. While focusing on source languages is useful to provide developers with assurances that their code does not leak information unduly, it is definitely preferable for users to be provided with enforcement mechanisms that operate at bytecode level, because Java applets are downloaded as JVM bytecode programs.

The purpose of this article is to present a type system to enforce confidentiality of object-oriented applications executing on a virtual machine, and to show that the type system enforces non-interference. Our virtual machine can be viewed, up to minor details, as a sequential fragment of the Java Virtual Machine, and features objects, methods, exceptions, and arrays (the latter are only briefly discussed). The analysis is

* Work partially supported by IST Project MOBIUS, by the RNTL Castles and by the ACI Sécurité SPOPS.

compatible with bytecode verification and can thus be integrated in a standard Java security architecture, provided class files are suitably extended with appropriate information expressed as security signatures for methods.

Thus, this paper provides the first sound analysis for a JVM-like language with objects, arrays, methods, and exceptions. The type system and the soundness results are inspired from [8], but this paper improves substantially on [8]: the operational semantics of the language is more realistic (we provide a treatment of exceptions that is close to that of Java) and both methods and arrays have been incorporated, the security policies are more expressive (we adopt arbitrary lattices of security levels instead of two-element lattices), the enforcement mechanism is more accurate (we rely on preliminary exception analyses to reduce the control flow graph of applications), and the soundness proof has been machine checked using the proof assistant Coq [13].

Relation with information-flow type system for Java. Although our type system has been developed independently from information flow type systems for Java, we have showed in joint work with D. Naumann [7] that source and bytecode type systems are related, in the sense that a standard (non-optimizing) Java compiler will translate programs that are typable in a type system inspired from [4], but extended to exceptions, into programs that are typable in our system. The combination of the present article with [4] thus guarantee that our type system is sufficiently expressive to accept compiled versions of the examples of [4], and more generally that the experimental programming language JFlow [25], of which the language and type system of [4] can be seen as a fragment, can be used to develop information-flow aware applications that are accepted by our type system.

Preliminaries. For every function $f \in A \to B$, $x \in A$ and $v \in B$, we let $f \oplus \{x \mapsto v\}$ denote the unique function f' s.t. $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = v$. Further, we let $A^\star$ denote the set of A-stacks for every set A. We use hd and tl and :: and $+\!\!+$ to denote the head and tail and cons and concatenation operations on stacks.

Security lattice in examples. For simplicity, examples throughout the paper take as partial order of security levels $\mathcal{S} = \{L, H\}$ with $L \leq H$, where H is the high level for confidential data, and L is the low level for observable data.

2 Informal overview

The purpose of this section is to provide an informal account of our security condition, to highlight some salient features of our type system, and finally to provide a high level description of the type soundness proof. In order to avoid a profusion of technical details, we ignore exceptions in a first place, and indicate at the end of the section additional issues that arise when they are considered.

2.1 Policies and attacker model

The security policy is based on the assumption that the attacker can only draw observations on the input/output behavior of methods. On the other hand, we adopt a termination insensitive policy which assumes that the attacker is unable to observe non-termination of programs. Formally, the policy is given by a partial order $(\mathcal{S}, \leq)$ of security levels, and:

- a security level k_{obs} that determines the observational capabilities of the attacker. Essentially, the attacker can observe fields, local variables, and return values whose level is below k_{obs};
- a global policy $\mathsf{ft} : \mathcal{F} \to \mathcal{S}$ that attaches security levels to fields (we let $\mathcal{F}$ denote the set of fields). The global policy is used to determine a notion of equivalence $\sim$ between heaps. Intuitively, two heaps h_1 and h_2 are equivalent if $h_1(l).f = h_2(l).f$ for all locations l and fields f s.t. $\mathsf{ft}(f) \leq k_{\text{obs}}$; the formal definition of heap indistinguishability is rather involved and deferred to Section 4;

– local policies for each method (we let $\mathcal{M}$ denote the set of methods). In a setting where exceptions are ignored, local policies are expressed using security signatures of the form $\boldsymbol{k_v} \xrightarrow{k_h} k_r$ where $\boldsymbol{k_v}$ provides the security level of the method local variables (including methods arguments[1]), k_h is the effect of the method on the heap, and k_r is the return signature, i.e. the security level of the return value. The vector $\boldsymbol{k_v}$ of security levels is used to determine a notion of indistinguishability $\sim_{\boldsymbol{k_v}}$ between arrays of parameters, whereas the return signature is used to define a notion of indistinguishability $\sim_{k_r}$ between return values.

Essentially, a method is safe w.r.t. a signature $\boldsymbol{k_v} \xrightarrow{k_h} k_r$ if:

1. two terminating runs of the method with $\sim_{\boldsymbol{k_v}}$-equivalent inputs, i.e. inputs that cannot be distinguished by an attacker, and equivalent heaps, yield $\sim_{k_r}$-equivalent results, i.e. results that cannot be distinguished by the attacker,
2. the method does not perform field updates on fields whose security level is below k_h—as a consequence, it cannot modify the heap in a way that is observable by an attacker that has access to fields whose security level is below k_h.

Formally, the security condition is expressed relative to the operational semantics of the JVM, which is captured by judgments of the form $h_i, lv \Downarrow_m r, h_f$, meaning that executing the method m with initial heap h_i and parameters lv yields the final heap h_f and the result r.

Then, we say that a method m is *safe* w.r.t. a signature $\boldsymbol{k_v} \xrightarrow{k_h} k_r$ if its method body does not perform field updates on fields of level lower than k_h and if it satisfies the following non-interference property: for all heaps h_i, h_f, h'_i, h'_f, arrays of parameters $\boldsymbol{a}$ and $\boldsymbol{a'}$, and results r and r',

$$\left.\begin{array}{c} h_i, \boldsymbol{a} \Downarrow_m r, h_f \\ h'_i, \boldsymbol{a'} \Downarrow_m r', h'_f \\ h_i \sim h'_i \\ \boldsymbol{a} \sim_{\boldsymbol{k_v}} \boldsymbol{a'} \end{array}\right\} \;\Rightarrow\; h_f \sim h'_f \;\wedge\; r \sim_{k_r} r'$$

There are two important underlying choices in this security condition: first, the security condition focuses on input/output behaviors, and so does not consider the case of executions that hang; however, it also does not consider "wrong" executions that get stuck, as such executions are eliminated by bytecode verification. Second, the security condition is defined on methods, and not on programs, as we aim for a modular verification technique in the spirit of bytecode verification.

2.2 Dealing with unstructured programs

Preventing direct flows with stack types. Any sound information flow type system must prevent direct information leakages that occur through assigning secret values to public variables. In a high level language, avoiding such indirect flows is ensured by setting appropriate rules for assignments; in a typical type system for a high-level language [34], the typing rule for assignments is of the form

$$\frac{\vdash e : \; k \qquad k \leq \boldsymbol{k_v}(x)}{\vdash x := e : \; \boldsymbol{k_v}(x)}$$

where $\boldsymbol{k_v}(x)$ is the security given to variable x by the policy and k is an upper bound of the security level of the variables occurring in the expression e. The constraint $k \leq \boldsymbol{k_v}(x)$ ensures that the value stored in x does not depend of any variable whose security level is greater than that of x, and thus that there is no illicit flow to x.

[1] JVM programs use a fragment of their local variables to store parameter values.

In a low level language where intermediate computations are performed with an operand stack, direct information flows are prevented by assigning a security level to each value in the operand stack, via a so-called *stack type*, and by rejecting programs that attempt storing a value in a low variable when the top of the stack type is high:

$$\frac{P[i] = \mathsf{load}\ x}{i \vdash st \Rightarrow \boldsymbol{k_v}(x) :: st} \qquad \frac{P[i] = \mathsf{store}\ x \quad k \leq \boldsymbol{k_v}(x)}{i \vdash k :: st \Rightarrow st}$$

where st represents an stack type (an stack of security levels) and $\Rightarrow$ represents a relation between the stack type before execution and the stack type after execution of load.

For instance, $x_L = y_H$ is rejected by any sound information flow type system for a while language, because the constraint $H \leq L$ generated by the typing rule for assignment is violated. Likewise, the low level counterpart

$$\begin{array}{l} \mathsf{load}\ y_H \\ \mathsf{store}\ x_L \end{array}$$

cannot be typed as the typing rule for load forces the top of the stack type to high after executing the instruction, and the typing rule for store generates the constraint $H \leq L$.

Preventing indirect flows via security environments Any sound information flow type system must also prevent information leakages that occur through the control flow of programs. In a high level language, avoiding such indirect flows is ensured by setting appropriate rules for branching statements; in a typical type system for a high-level language [34], the typing rule for if statements is of the form

$$\frac{\vdash e :\ k \qquad \vdash c_1 :\ k_1 \qquad \vdash c_2 :\ k_2 \qquad k \leq k_1, k_2}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 :\ k}$$

and ensures that the write effects of c_1 and c_2 are greater than the guard of the branching statement.

To prevent illicit flows in a low-level language, one cannot simply enforce local constraints in the typing rules for branching instructions: one must also enforce global constraints that prevent low assignments and updates to occur under high guards. In order to express the global constraints that are necessary to enforce soundness, we rely on additional information about the program, namely control dependence regions (cdr) which approximate the scope of branching statements. The cdr information:

- is defined relative to a binary successor relation $\mapsto \subseteq \mathcal{PP} \times \mathcal{PP}$ between program points, and a set $\mathcal{PP}_r$ of return points. The successor relation and the set of return points are defined according to the semantics of instructions. Intuitively, j is a successor of i if performing one-step execution from a state whose program point is i may lead to a state whose program point is j. Likewise, j is a return point if it corresponds to a return instruction. In the sequel, we write $i \mapsto$ if $i \in \mathcal{PP}_r$;
- is captured by a function that maps a branching program point i (i.e. a program point with two or more successors) to a set of program points $\mathsf{region}(i)$, called the region of i, and by a partial function that maps branching program points to a junction point $\mathsf{jun}(i)$.

The intuition behind regions and junction points is that $\mathsf{region}(i)$ includes all program points executing under the guard of i and that $\mathsf{jun}(i)$, if it exists is the sole exit to the region of i; in particular, whenever $\mathsf{jun}(i)$ is defined there should be no return instruction in $\mathsf{region}(i)$. The properties to be satisfied by control dependence regions are further discussed in next section.

In the type system, we use cdr information in conjunction with a security environment that attaches to each program point a security level, intuitively the upper bound of all the guards under which the program point execute. More precisely, programs are checked against a security environment se and global constraints arise in the type system as side conditions in the typing rules for branching statements. For instance, the rule for if bytecode is of the form:

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad \forall j' \in \mathsf{region}(i).k \leq se(j')}{i \vdash k :: st \Rightarrow \cdots}$$

(In Section 3, we discuss the possible choices for the result stack type in the conclusion.)

In order to prevent indirect flows, the typing rules for instructions with write effect, e.g. store and putfield, must check that the security level of the variable or field to be written is at least as high as the current security environment. For instance, the rule for store becomes:

$$\frac{P[i] = \mathsf{store}\ x \quad k \sqcup se(i) \leq \boldsymbol{k_v}(x)}{i \vdash k :: st \Rightarrow st}$$

The combination of both rules allows to prevent indirect flows. For instance, the standard example of indirect flow `if` ($\mathtt{y_H}$) $\{\mathtt{x_L} = \mathtt{0};\}$ `else` $\{\mathtt{x_L} = \mathtt{1};\}$ is compiled in our low-level language as

$$\begin{array}{ll} & \mathsf{load}\ y_H \\ & \mathsf{ifeq}\ l_1 \\ & \mathsf{push}\ 0 \\ & \mathsf{store}\ x_L \\ & \mathsf{goto}\ l_2 \\ l_1 : & \mathsf{push}\ 1 \\ & \mathsf{store}\ x_L \\ l_2 : & \ldots \end{array}$$

By requiring that $se(i) \leq \boldsymbol{k_v}(x)$ in the store rule and by requiring a global constraint on the security environment in the rule for ifeq, the type system ensures that the above program will be rejected: $se(i)$ must be H if the store instruction is under the influence of a high ifeq, and thus the transition for the store instruction cannot be typed.

2.3 Verification of control dependence regions

Since information flow type checking of programs is performed w.r.t. control dependence regions, we assume that the cdr information comes bundled with the program, and its correctness is verified by a cdr checker that is included in the TCB (unlike what is claimed in e.g. [35], the cdr information itself is not trusted).

Thus, we assume that the cdr information is given by functions region and jun. To guarantee the correctness of the information that they provide, these functions should satisfy the set of properties given below.

Informally, the properties state that any successor of i either belongs to the region of i, or are equal to $\mathsf{jun}(i)$ (if defined), and $\mathsf{jun}(i)$ is the sole exit to the region of i; in particular if $\mathsf{jun}(i)$ is defined there should be no return instruction in $\mathsf{region}(i)$.

Definition 1. *A cdr structure* $(\mathsf{region}, \mathsf{jun})$ *satisfies the SOAP (Safe Over APproximation) properties if the following properties hold:*

SOAP1: *for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in \mathsf{region}(i)$ or $k = \mathsf{jun}(i)$;*
SOAP2: *for all program points i, j, k, if $j \in \mathsf{region}(i)$ and $j \mapsto k$, then either $k \in \mathsf{region}(i)$ or $k = \mathsf{jun}(i)$;*
SOAP3: *for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $\mathsf{jun}(i)$ is undefined.*

The SOAP properties can be viewed as properties of finite graphs; they can be performed in quadratic time.

Our motivations to bundle the cdr information with programs is that it streamlines the presentation and that it allows us to focus on the information flow analysis itself. However, it is by no means necessary that programs come equipped with their cdr information. In fact, the cdr information can be computed by an analyzer using standard algorithms [**?**,**?**]; furthermore, the complexity of computing regions is the same as the complexity of verifying them, i.e. it can be done in quadratic time. In this case, the TCB includes the analyzer that computes control dependence regions.

2.4 Type system

Our information flow type system adopts the principles of Java bytecode verification, in the sense that it is modular (each method can be verified against its signature in isolation) and that it is defined as a data flow analysis of an abstract transition relation. Formally, the type system is parameterized by:

- a table Γ of method signatures, necessary for typing rules involving method calls;
- a global policy ft that provides the security level of fields;
- a cdr structure $(\mathsf{region}, \mathsf{jun})$ for the method under verification;
- a security environment se;
- a current method signature sgn.

The typing rules are designed to prevent information leakage through imposing appropriate constraints; typing rules are of one of the two forms below, where the rule on the left is used for normal intra-method execution, and the rule on the right is used for return instructions:

$$\frac{P[i] = ins \quad constraints}{\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash st \Rightarrow st'} \qquad \frac{P[i] = ins \quad constraints}{\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash st \Rightarrow}$$

where $st, st' \in \mathcal{S}^\star$ are stacks of security levels, and ins is an instruction found at point i in program P. Typing rules are used to establish a notion of typability. Following Freund and Mitchell [16], typability stipulates the existence of a function, that maps program points to stack types, such that each transition is well-typed.

Definition 2 (Typable method). *A method m is deemed typable w.r.t. a method signature table Γ, a global field policy ft, a signature sgn and a cdr $\mathsf{region} : \mathcal{PP} \to \wp(\mathcal{PP})$ if there exists a security environment $se : \mathcal{PP} \to \mathcal{S}$ and a function $S : \mathcal{PP} \to \mathcal{S}^\star$ such that $S_1 = \varepsilon$ (the operand stack is empty at the initial program point 1), and for all $i, j \in \mathcal{PP}$:*

1. *$i \mapsto j$ implies that there exists $st \in \mathcal{S}^\star$ such that $\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash S_i \Rightarrow st$ and $st \sqsubseteq S_j$;*
2. *$i \mapsto$ implies that $\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash S_i \Rightarrow$;*

where we write S_i instead of $S(i)$ and $\sqsubseteq$ denotes the point-wise partial order on type stack with respect to the partial order taken on security levels.

The definition of typable method is stated to ensure that runs of typable programs (i.e. programs whose methods are typable against their signatures) verify at each step the constraints imposed by the typing rules, provided they are called with parameters that respect the signature of their main method. This can be made more precise by introducing a notion of typable execution, which is cast in terms of one step execution of the virtual machine. The latter is itself formalized as a binary relation $\rightsquigarrow$ between states and states or return values. We formalize the one step execution relation using a big-step semantics for method calls (i.e. we consider that method calls are performed in one step), and thus deal with states that consist of a heap, a local variable map, an operand stack, and a program counter. In the sequel, we implicitly consider execution w.r.t. a method m.

Definition 3 (Typable run).

1. *An execution step $s \rightsquigarrow s'$ is typable with respect to $S \in \mathcal{PP} \to \mathcal{S}^\star$ if there exists st' such that, $i \vdash^\tau S_i \Rightarrow st'$ and $st' \sqsubseteq S_{i'}$, where i (resp. i') is the current program counter of s (resp. s').*
2. *An execution step $s \rightsquigarrow (r, h)$ is typable with respect to $S \in \mathcal{PP} \to \mathcal{S}^\star$ if $i \vdash^\tau S_i \Rightarrow$, where i is the current program counter of s.*
3. *An execution sequence $s_0 \rightsquigarrow s_1 \rightsquigarrow \ldots s_k \rightsquigarrow (r, h)$ is typable with respect to $S \in \mathcal{PP} \to \mathcal{S}^\star$ if*
   - *for all i, $0 \leq i < k$, $s_i \rightsquigarrow s_{i+1}$ is typable with respect to S;*
   - *$s_n \rightsquigarrow (r, h)$ is typable with respect to S.*

Typability of a method against its signature can be performed via a dataflow analysis based on Kildall's algorithm [**?**]. The analysis takes as inputs the local and global policies, the method table, the cdr structure, the security environment, the current signature, and either returns a type $S : \mathcal{PP} \to \mathcal{S}^\star$, or a tag indicating that type-checking has failed. Assuming that the set of security levels is finite, one is sure that the analysis always terminates, and thus it is decidable whether a method is typable against a signature. To decide whether a method is typable, it is possible to either perform a fixpoint computation using Kildall's analysis, or check the method against its certificate (assigning types to junction points), using lightweight bytecode verification techniques [**?**].

Although we do not discuss alternatives in detail, there are several variants to the notion of typable methods, and to verifying typability. Two dimensions of choice, inherited from Java bytecode verification [**?**], are the precision of the analysis (polyvariant vs. monovariant) and its power (checking vs. inferring):

Monovariant vs. polyvariant types: in previous work [5], we opted for polyvariant types that assign to each program point a set of stack types. Polyvariant analyses rely on the finiteness of the set of stack types to guarantee termination. They type more programs, but yield less compact types.
At a technical level, subtyping lemmas at the end of Section 3.4 are not needed to establish soundness of polyvariant analyzes, and the notion of typable run does not need to be based on subtyping in the transitivity rule.

Checking vs inferring: the dataflow analysis performs a fixpoint computation by repeating iterations over the program. Alternatively, it is possible to annotate programs with type information at junction points, and to perform the analysis in one pass. In a nutshell, the idea is to check whether the types computed at junction points are compatible with the declared type in the certificate, and to follow the analysis with the latter if it is the case.

2.5 Proving type soundness

For each of the fragment of the JVM considered in this paper, we adopt a similar method to prove soundness of the type system. The proof of soundness is based on some assumptions concerning the cdr information, two unwinding lemmas and two lemmas about preserving high context.

The unwinding lemmas show that execution of typable programs does not reveal secret information. They are stated relative to the small-step semantics $\rightsquigarrow$ and to a notion of state indistinguishability that is defined component-wise, i.e. two states s and t are indistinguishable if and only if their heaps, local variable maps, and operand stacks are indistinguishable. As shall be explained in Section 3, indistinguishability between operand stacks is defined relative to stack types S and T, and hence we must also defined state indistinguishability relative to stack types. In the sequel, we write $s \sim_{S,T} t$ whenever s and t are equivalent w.r.t. S and T.

The unwinding lemmas deal with a program P that come equipped with its method signature table and a particular method m of P that comes equipped with its cdr structure $(\mathsf{region}, \mathsf{jun})$ and security environment se. We say that a type stack $S \in \mathcal{S}^\star$ is high if all levels in S are not lower than k_{obs}. We say that the security environment se is high in region $\mathsf{region}(i)$ if $se(j) \not\leq k_{\text{obs}}$ for all $j \in \mathsf{region}(i)$.

- *locally respects:* if $s \sim_{S,T} t$, and $\mathsf{pc}(s) = \mathsf{pc}(t) = i$, and $s \rightsquigarrow s'$, $t \rightsquigarrow t'$, $i \vdash S \Rightarrow S'$, and $i \vdash T \Rightarrow T'$, then $s' \sim_{S',T'} t'$.
- *step consistent:* if $s \sim_{S,T} t$ and $s \rightsquigarrow s'$ and $\mathsf{pc}(s) \vdash S \Rightarrow S'$, and security environment at program point $\mathsf{pc}(s)$ is high, and S is high, then $s' \sim_{S',T} t$.

In addition to the unwinding lemmas, we need two lemmas to ensure the preservation of high context.

- *high branching:* if $s \sim_{S,T} t$ with $\mathsf{pc}(s) = \mathsf{pc}(t) = i$ and $\mathsf{pc}(s') \neq \mathsf{pc}(t')$, if $s \rightsquigarrow s'$, $t \rightsquigarrow t'$, $i \vdash S \Rightarrow S'$ and $i \vdash T \Rightarrow T'$, then S' and T' are high and se is high in region $\mathsf{region}(i)$.
- *high step:* if $s \rightsquigarrow s'$, and $\mathsf{pc}(s) \vdash S \Rightarrow S'$, and security environment at program point $\mathsf{pc}(s)$ is high, and S is high, then S' is high.

Using this four hypothesis and the SOAP conditions, we prove that step consistency can be extended to execution of high fragments of code, i.e. we show that any complete execution starting from a state s pointing to a high branching instruction will either pass through a state s' such that $\mathsf{pc}(s') = \mathsf{jun}(\mathsf{pc}(s))$ and $s \sim_{S,S'} s'$, or will terminate with a high value. In the second case, the return signature of the method must be high, otherwise the program is rejected; however, if the return signature of the method is high, then it is trivially safe. Hence we only consider the first possibility in the sequel.

We also exploit the typability of P and the SOAP properties to show that, under the hypotheses of the "locally respects" unwinding lemma, we either have $\mathsf{pc}(s') = \mathsf{pc}(t')$, in which case we can apply again the "locally respects" lemma (unless s' and t' are results), or we can apply the "step consistent" unwinding lemma to both s' and t' (using SOAP1), and execution paths will eventually reach program point $\mathsf{jun}(i)$ in states that are equivalent to s' and t' respectively.

Summarizing, the main result of the paper is to prove for JVM fragments of increasing complexity, the following theorem.

Main theorem 1 *Let P be a program in which each method comes equipped with its method signature, it cdr structure* $(\mathsf{region}, \mathsf{jun})$ *and its security environment.*

1. *If all cdr structures satisfy the SOAP properties and all methods are typable with their signature, then each method is safe, and in particular the program is non-interfering (i.e. it main method is safe).*
2. *The SOAP properties can be verified automatically for each method and associated cdr structure.*
3. *The typability of a method against its signature can be verified automatically.*

2.6 Exceptions

Extending the outline of the previous section to exceptions is an important issue, because exceptions are pervasive in Java programs, and because they introduce several potential sources of information leakage. In [8], the authors show that in absence of methods it is rather direct to prove type soundness for a simplified exception mechanism where only one kind of exception is allowed: indeed, exceptions are just handled by extending the successor relation appropriately and by adding additional typing rules for instructions that may raise exceptions.

In this paper, we allow multiple exceptions together with method calls. Integrating exceptions requires many changes, in particular in the definition of method signatures, which become very similar to the signatures used in Jif, and in the definition of the control dependence regions, which now become parameterized by an exception. These changes are detailed in Section 6.

2.7 Main limitations

The information flow type system presented in this article constitutes, to our best knowledge, the first analysis to be proved sound for a fragment of the JVM that includes objects, methods, and exceptions. There are however a number of limitations in our work: in particular, the security condition does not support declassification, dynamic policies, nor label polymorphism. Overcoming these limitations is left for further work.

Besides, our type system does not support context sensitivity, as the security level of local variables is fixed throughout execution. As a consequence of context insensitivity, the type system restricts the possibilities of local variable reuse. However, Leroy [**?**] argues that removing local variable polymorphism for Java bytecode is important for efficient on-device verification and that it has a negligible impact on performance and resource usage. We believe that his observations remain applicable to our information flow type system.

Finally, we also make the assumption that all methods return a result; this is a harmless departure from Java, which allows us to avoid duplicating many definitions. This assumption is done here for the sake of presentation, but the formal proofs do consider both the cases of methods returning a result, and methods returning no result.

$$
\begin{array}{rll}
\mathit{instr} ::= & \mathsf{binop}\ \mathit{op} & \text{binary operation on stack} \\
\mid & \mathsf{push}\ c & \text{push value on top of stack} \\
\mid & \mathsf{pop} & \text{pop value from top of stack} \\
\mid & \mathsf{swap} & \text{swap the top two operand stack values} \\
\mid & \mathsf{load}\ x & \text{load value of } x \text{ on stack} \\
\mid & \mathsf{store}\ x & \text{store top of stack in variable } x \\
\mid & \mathsf{ifeq}\ j & \text{conditional jump} \\
\mid & \mathsf{goto}\ j & \text{unconditional jump} \\
\mid & \mathsf{return} & \text{return the top value of the stack}
\end{array}
$$

where $\mathit{op} \in \{+, -, \times, /\}$, $c \in \mathbb{Z}$, $x \in \mathcal{X}$, and $j \in \mathcal{PP}$.

Fig. 1. Instruction set for $\mathrm{JVM}_{\mathcal{I}}$

2.8 Summary of subsequent sections

The subsequent sections analyze in turn increasingly complex fragments of the JVM:

- the machine $\mathrm{JVM}_{\mathcal{I}}$, studied in Section 3, includes basic operations to manipulate operand stacks as well as conditional and unconditional jumps, and is expressive enough for compiling programs written in a simple imperative language. In this section, we define and discuss operand stack indistinguishability. The definitions and type system for $\mathrm{JVM}_{\mathcal{I}}$ are adapted from our earlier work [**?**];
- the machine $\mathrm{JVM}_{\mathcal{O}}$, studied in Section 4, is an object-oriented extension of $\mathrm{JVM}_{\mathcal{I}}$ which includes features such as dynamic object creation, instance field accesses and updates, and is expressive enough for compiling intra-procedural statements from [4]. In this section, we define and discuss heap indistinguishability. The definitions and type system for $\mathrm{JVM}_{\mathcal{I}}$ are adapted from our earlier work [8];
- $\mathrm{JVM}_{\mathcal{C}}$, studied in Section 5, is a procedural extension of $\mathrm{JVM}_{\mathcal{O}}$ with method calls, and is expressive enough to compile the language of [4]. The main difficulty is to handle information leakages caused by dynamic method dispatch;
- $\mathrm{JVM}_{\mathcal{G}}$, studied in Section 5, extends $\mathrm{JVM}_{\mathcal{C}}$ with exceptions. The main difficulty is to handle information leakages caused by exceptions, especially when they escape the scope of the method in which they are raised.

For each fragment, we shall define programs, states, and semantics. Then we shall formulate the security policy and the typing rules, and state the unwinding lemmas.

3 The $\mathrm{JVM}_{\mathcal{I}}$ submachine

In this section, we define an information flow type system for a fragment of the JVM with conditional and unconditional jumps and operations to manipulate the stack.

3.1 Programs, memory model and operational semantics

Programs In this fragment, a program consists of a single, non-recursive method. Thus we consider that a $\mathrm{JVM}_{\mathcal{I}}$ program P is given by its list of instructions, taken from the instruction set of Figure 1. We let the set $\mathcal{X}$ be the set of local variables and $\mathcal{V}$ the set of values.

States In this fragment, states do not feature a heap. Thus, the set $\mathsf{State}_{\mathcal{I}}$ of $\mathrm{JVM}_{\mathcal{I}}$ states is defined as the set of triples $\langle i, \rho, os\rangle$, where $i \in \mathcal{PP}$ is the program counter that points to the next instruction to be executed; $\rho \in \mathcal{X} \rightharpoonup \mathcal{V}$ is a partial function from local variables to values (that can be view as an array of values) where the set $\mathcal{V}$ of values is defined as $\mathbb{Z}$, and $os \in \mathcal{V}^\star$ is an operand stack.

Operational semantics The small-step operational semantics of the $\text{JVM}_{\mathcal{I}}$, is given in Figure 2 as a relation $\rightsquigarrow \subseteq \mathsf{State}_{\mathcal{I}} \times (\mathsf{State}_{\mathcal{I}} + \mathcal{V})$. This relation is implicitly parameterized by a program P. $\underline{op}$ denotes here the standard interpretation of operation of op in the domain of values $\mathcal{V}$. The semantics of each instruction is quite standard. Instruction $\mathsf{push}\ c$, pushes a constant c on top of the operand stack. Instruction $\mathsf{binop}\ op$ pops the two top operands of the stack and push the result of the binary operation op using these operands. Instruction pop, just pops the top of the operand stack. Instruction swap, swaps the top two operand stack values. Instruction return ends the execution with the top value of the operand stack. Instruction $\mathsf{load}\ x$ pushes the value currently found in local variable x, on top of the operand stack. Instruction $\mathsf{store}\ x$ pops the top of the stack and stores it in local variable x. Instruction $\mathsf{ifeq}\ j$ pops the top of the stack and depending on weather it is a null value or not, it jumps to the program point j or continue to the next program point. Instruction $\mathsf{goto}\ j$ unconditionally jumps to program point j. For clarity reasons, we hide program points in program examples and use labels to design jump targets.

$$\frac{P[i] = \mathsf{push}\ n}{\langle i, \rho, os\rangle \rightsquigarrow \langle i+1, \rho, n :: os\rangle} \qquad \frac{P[i] = \mathsf{binop}\ op \qquad n_2\ \underline{op}\ n_1 = n}{\langle i, \rho, n_1 :: n_2 :: os\rangle \rightsquigarrow \langle i+1, \rho, n :: os\rangle}$$

$$\frac{P[i] = \mathsf{pop}}{\langle i, \rho, v :: os\rangle \rightsquigarrow \langle i+1, \rho, os\rangle} \qquad \frac{P[i] = \mathsf{swap}}{\langle i, \rho, v_1 :: v_2 :: os\rangle \rightsquigarrow \langle i+1, \rho, v_2 :: v_1 :: os\rangle}$$

$$\frac{P[i] = \mathsf{return}}{\langle i, \rho, v :: os\rangle \rightsquigarrow v} \qquad \frac{P[i] = \mathsf{load}\ x}{\langle i, \rho, os\rangle \rightsquigarrow \langle i+1, \rho, \rho(x) :: os\rangle}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad x \in \mathsf{dom}(\rho)}{\langle i, \rho, v :: os\rangle \rightsquigarrow \langle i+1, \rho \oplus \{x \mapsto v\}, os\rangle} \qquad \frac{P[i] = \mathsf{ifeq}\ j}{\langle i, \rho, 0 :: os\rangle \rightsquigarrow \langle j, \rho, os\rangle}$$

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad n \neq 0}{\langle i, \rho, n :: os\rangle \rightsquigarrow \langle i+1, \rho, os\rangle} \qquad \frac{P[i] = \mathsf{goto}\ j}{\langle i, \rho, os\rangle \rightsquigarrow \langle j, \rho, os\rangle}$$

Fig. 2. Operational Semantics for $\text{JVM}_{\mathcal{I}}$

The transitive closure of $\rightsquigarrow$ to a final value is inductively defined by:

$$\frac{\langle i, \rho, os\rangle \rightsquigarrow v}{\langle i, \rho, os\rangle \Downarrow v} \qquad \frac{\langle i, \rho, os\rangle \rightsquigarrow \langle j, \rho', os'\rangle \quad \langle j, \rho', os'\rangle \Downarrow v}{\langle i, \rho, os\rangle \Downarrow v}$$

The evaluation of a program $\rho \Downarrow v$, from an array of initial local variables ρ to final value is then defined by

$$\rho \Downarrow v \equiv \langle 1, \rho, \varepsilon\rangle \Downarrow v$$

because execution start at program point 1 with an empty operand stack.

Successor relation The successor relation $\mapsto \subseteq \mathcal{PP} \times \mathcal{PP}$ of a program P is defined by the clauses:

- if $P[i] = \mathsf{goto}\ j$, then $i \mapsto j$;
- if $P[i] = \mathsf{ifeq}\ j$, then $i \mapsto i+1$ and $i \mapsto j$;
- if $P[i] = \mathsf{return}$, then i has no successors, and we write $i \mapsto$;
- otherwise, $i \mapsto i+1$.

3.2 Non-Interference

In this fragment, there is no global policy, and a single local policy for the sole method of the program. Furthermore, the local policy does not refer to heap effect, and is thus of the form $\boldsymbol{k_v} \longrightarrow k_r$.

The first step to define the security policy is to introduce a notion of indistinguishability between values. In this case, value indistinguishability is trivial.

Definition 4 (Low value indistinguishability). *Two values v and v' are (low)-indistinguishable, written $v \sim v'$, iff $v = v'$.*

Then, indistinguishability is extended to local variable maps.

Definition 5 (Local variables indistinguishability). *For $\rho, \rho' : \mathcal{X} \rightharpoonup \mathcal{V}$, we have $\rho \sim \rho'$ if ρ and ρ' have the same domain and $\rho(x) \sim \rho'(x)$ for all $x \in \mathsf{dom}(\rho)$ such that $\boldsymbol{k_v}(x) \leq k_{\text{obs}}$.*

Strictly speaking, we should write $\sim_{\boldsymbol{k_v}}$, but usually we simply write $\sim$ since there is no risk of confusion.

Definition 6 (Non-interferent $\text{JVM}_{\mathcal{I}}$ program). *A program P is* non-interferent *w.r.t. its policy $\boldsymbol{k_v} \longrightarrow k_r$, if for every ρ_1, ρ_2, v_1, v_2 such that $\rho_1 \Downarrow v_1$ and $\rho_2 \Downarrow v_2$ and $\rho_1 \sim_{\boldsymbol{k_v}} \rho_2$ and $k_r \leq k_{\text{obs}}$, we have $v_1 \sim v_2$, i.e. $v_1 = v_2$.*

3.3 Typing rules

$$\frac{P[i] = \mathsf{push}\ n}{i \vdash st \Rightarrow se(i) :: st} \qquad \frac{P[i] = \mathsf{binop}\ op}{i \vdash k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2 \sqcup se(i)) :: st}$$

$$\frac{P[i] = \mathsf{pop}}{i \vdash k :: st \Rightarrow st} \qquad \frac{P[i] = \mathsf{swap}}{i \vdash k_1 :: k_2 :: st \Rightarrow k_2 :: k_1 :: st}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad se(i) \sqcup k \leq \boldsymbol{k_v}(x)}{i \vdash k :: st \Rightarrow st} \qquad \frac{P[i] = \mathsf{load}\ x}{i \vdash st \Rightarrow (\boldsymbol{k_v}(x) \sqcup se(i)) :: st}$$

$$\frac{P[i] = \mathsf{goto}\ j}{i \vdash st \Rightarrow st} \qquad \frac{P[i] = \mathsf{return} \qquad se(i) \sqcup k \leq k_r}{i \vdash k :: st \Rightarrow}$$

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad \forall j' \in \mathsf{region}(i),\ k \leq se(j')}{i \vdash k :: st \Rightarrow \mathrm{lift}_k(st)}$$

Fig. 3. Transfer rules for instructions in $\text{JVM}_{\mathcal{I}}$

Figure 3 presents a set of typing rules that guarantee non-interference for $\mathsf{JVM}_{\mathcal{I}}$. $\sqcup$ denotes the lub of two security levels, and for every $k \in \mathcal{S}$, lift_k is the point-wise extension to stack types of $\lambda l.\ k \sqcup l$. All rules are implicitly parameterized by a cdr region, a security environment se and a signature $\boldsymbol{k_a} \longrightarrow k_r$.

Below we comment on some essential rules:

– The transfer rule for an instruction $\mathsf{push}\ n$ prevents direct flows by requiring that the value pushed on top of the operand stack has a security level greater than the security environment at the current program point. The following example, compiled from the source program `return` $\mathtt{y_H}\,\mathtt{?}\,\mathtt{0}\,\mathtt{:}\,\mathtt{1;}$, illustrates the need for this constraint:

$$\begin{array}{ll} & \mathsf{load}\ y_H \\ l_1 : & \mathsf{ifeq}\ l_2 \\ & \left.\begin{array}{l} \mathsf{push}\ 0 \\ \mathsf{goto}\ l_3 \end{array}\right\} \mathsf{region}(l_1) \\ l_2 : & \mathsf{push}\ 1 \\ l_3 : & \mathsf{return} \end{array}$$

The program is interferent with respect to the policy $(y_H : H) \longrightarrow L$, but not typable. Typing rule for return instruction reject this program because the top of the stack type is high. Indeed, instruction $\mathsf{push}\ 0$ and $\mathsf{push}\ 1$ are in the region of the branching instruction $\mathsf{ifeq}\ l_1$ and security environment se is high at this point.

A similar constraint appears in the typing rule for binop for the same reasons.

– the typing rule for ifeq requires the stack type on the right hand side of $\Rightarrow$ to be lifted by the level of the guard, i.e. the top of the input stack type. It is necessary to perform this lifting operation to avoid illicit flows through operand stack leakages. The following example illustrates why we need to lift the operand stack. This is a contrived example because it does not correspond to any simple source code, but it is nevertheless accepted by a standard bytecode verifier.

$$\begin{array}{ll} & \mathsf{push}\ 0 \\ & \mathsf{push}\ 1 \\ & \mathsf{load}\ y_H \\ l_1 : & \mathsf{ifeq}\ l_2 \\ & \left.\begin{array}{l}\mathsf{swap}\\ \mathsf{pop}\\ \mathsf{goto}\ l_3\end{array}\right\}\ \mathsf{region}(l_1) \\ l_2 : & \mathsf{pop} \\ l_3 : & \mathsf{store}\ x_L \end{array}$$

In this example, the final value of variable x_L is equal to the value of y_H. So the program is interferent. It is nevertheless rejected by our type system, thanks to the lift of the operand stack at point l_1 that constrain the top of the stack at point l_3 to be a high value (store rule then prevents the assignment from high to low).
One may argue that lifting the entire stack is too restrictive, as it leads the typing system to reject safe programs; indeed, it should be possible, at the cost of added complexity, to refine the type system to avoid lifting the entire stack.
One may also argue that lifting the stack is unnecessary, because in mostprograms[2] the stack at branching points only has one element, in which case a more restrictive rule of the form below is sufficient:

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad \forall j' \in \mathsf{region}(i).k \leq se(j')}{i \vdash k :: \epsilon \Rightarrow \epsilon}$$

– The transfer rule for return requires $se(i) \leq k_r$ that avoids return instructions under the guard of expressions with a security level greater than k_r. In addition, the rule requires that the value on top of the operand stack has a security level above k_r, since it will be observed by the attacker. The following example illustrates the need for preventing return instructions in high regions. It corresponds to a source program like `if` ($\mathtt{y_H}$) {`return` 0;} `else` {`return` 1;}.

$$\begin{array}{ll} & \mathsf{load}\ y_H \\ l_1 : & \mathsf{ifeq}\ l_2 \\ & \left.\begin{array}{l}\mathsf{push}\ 0\\ \mathsf{return}\end{array}\right\}\ \mathsf{region}(l_1) \\ l_2 : & \left.\begin{array}{l}\mathsf{push}\ 1\\ \mathsf{return}\end{array}\right. \end{array}$$

This program is interferent because there is a return in a high ifeq. This program is rejected by the type system thanks to the ifeq rule which lifts the security environment, and the return rule which prevents the program from returning in a high security environment.

3.4 Type system soundness

The type system is sound, in the sense that if a program is typable then it is non-interferent.

[2] And even if this condition does not hold, code transformation is able to obtain an equivalent program respecting it [**?**]. We will discuss this point in Section .

Theorem 1. *Let P be a* $\mathrm{JVM}_{\mathcal{I}}$ *program and* $(\mathsf{region}, \mathsf{jun})$ *a safe cdr for P (according to SOAP properties). Suppose P is typable with respect to* region *and to a signature $\boldsymbol{k_a} \longrightarrow k_r$. Then P is non-interferent with respect to the policy associated with $\boldsymbol{k_a} \longrightarrow k_r$.*

Soundness proof follows the method sketched in Section 2. The four base lemmas, are based on the notion of state indistinguishability. The main difficulty in defining state indistinguishability resides in defining a good notion of operand stack indistinguishability: in order to account for high branching instructions, indistinguishability between states must encompass states that have operand stacks of different length. Indistinguishability between operand stacks is needed to establish the lemmas that claim that during execution indistinguishability of states is invariant.

We require operand stacks to be indistinguishable point-wise on some common top part, and then to be high in the bottom part on which they may not coincide as shown in Figure 4. High operand stacks are defined relative to a stack type: formally, let $os \in \mathcal{V}^\star$ be an operand stack and $st \in \mathcal{S}^\star$ be a stack type; we write $\mathsf{high}(os, st)$ if os and st have the same length n and $st[i] \not\leq k_{\mathrm{obs}}$ for every $1 \leq i \leq n$.

Definition 7 (Operand stack indistinguishability). *Let $os, os' \in \mathcal{V}^\star$ and $st, st' \in \mathcal{S}^\star$. Then $os \sim_{st,st'} os'$ is defined inductively as follows:*

$$\frac{\mathsf{high}(os, st) \qquad \mathsf{high}(os', st')}{os \sim_{st,st'} os'}$$

$$\frac{os \sim_{st,st'} os' \qquad v \sim v' \qquad k \leq k_{\mathrm{obs}}}{v :: os \sim_{k::st,k::st'} v' :: os'}$$

$$\frac{os \sim_{st,st'} os' \qquad k \not\leq k_{\mathrm{obs}} \qquad k' \not\leq k_{\mathrm{obs}}}{v :: os \sim_{k::st,k'::st'} v' :: os'}$$

Note that in the second rule the top of the two stack types are necessary equal (and low), while in the last rule they can be distinct (but not low). This distinction is necessary because we handle an arbitrary lattice of security levels.

Assuming that programs pass bytecode verification, one can simplify a bit the above definition. Indeed, bytecode verification requires that at each program point the height of the operand stack be fixed. Under this assumption, operand stack equivalence only requires that any two high operand stacks are equivalent and that operand stacks of the same height are equivalent if they are point-wise equivalent.

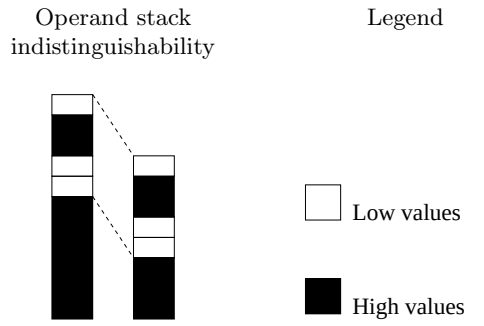

Fig. 4. Operand stack indistinguishability

State indistinguishability can then be defined component-wise on state structure.

Definition 8 (State indistinguishability). *Two states $\langle i, \rho, os\rangle$ and $\langle i', \rho', os'\rangle$ are indistinguishable w.r.t. st, $st' \in \mathcal{S}^\star$, denoted $\langle i, \rho, os\rangle \sim_{st,st'} \langle i', \rho', os'\rangle$, iff $os \sim_{st,st'} os'$ and $\rho \sim \rho'$ hold.*

We now state the four basic lemmas necessary for the soundness proof of theorem 1.

Lemma 1 ($\text{JVM}_\mathcal{I}$ locally respect). *Let $s_1, s_2 \in \text{State}_\mathcal{I}$ be two $\text{JVM}_\mathcal{I}$ states at the same program point i and let two stack types $st_1, st_2 \in \mathcal{S}^\star$ such that $s_1 \sim_{st_1,st_2} s_2$.*

Let $s_1', s_2' \in \text{State}_\mathcal{I}$ and $st_1', st_2' \in \mathcal{S}^\star$ such that $s_1 \rightsquigarrow s_1'$, $s_2 \rightsquigarrow s_2'$, $i \vdash st_1 \Rightarrow st_1'$ and $i \vdash st_2 \Rightarrow st_2'$ then $s_1' \sim_{st_1',st_2'} s_2'$.

Let $v_1, v_2 \in \mathcal{V}$ such that $s_1 \rightsquigarrow v_1$, $s_2 \rightsquigarrow v_2$, $i \vdash st_1 \Rightarrow$ and $i \vdash st_2 \Rightarrow$ then $k_r \leq k_{\text{obs}}$ implies $v_1 \sim v_2$.

Lemma 2 ($\text{JVM}_\mathcal{I}$ step consistent). *Let $\langle i, \rho, os\rangle, s_0 \in \text{State}_\mathcal{I}$ two $\text{JVM}_\mathcal{I}$ states and two stack types $st, st_0 \in \mathcal{S}^\star$ such that $\langle i, \rho, os\rangle \sim_{st,st_0} s_0$, $se(i) \not\leq k_{\text{obs}}$ and $\mathsf{high}(os, st)$.*

If there exists a state $\langle i', \rho', os'\rangle \in \text{State}_\mathcal{I}$ and a stack type $st' \in \mathcal{S}^\star$ such that $\langle i, \rho, os\rangle \rightsquigarrow \langle i, \rho', os'\rangle$ and $i \vdash st \Rightarrow st'$ then $\langle i', \rho', os'\rangle \sim_{st',st_0} s_0$.

If there exists a value $v \in \mathcal{V}$ such that $\langle i, \rho, os\rangle \rightsquigarrow v$ and $i \vdash st \Rightarrow$ then $k_r \not\leq k_{\text{obs}}$.

Lemma 3 ($\text{JVM}_\mathcal{I}$ high branching). *Let $s_1, s_2 \in \text{State}_\mathcal{I}$ be two $\text{JVM}_\mathcal{I}$ states at the same program point i and let $st_1, st_2 \in \mathcal{S}^\star$ such that $s_1 \sim_{st_1,st_2} s_1'$.*

If two states $\langle i_1, \rho_1', os_1'\rangle, \langle i_2, \rho_2', os_2'\rangle \in \text{State}_\mathcal{I}$ and two stack types $st_1', st_2' \in \mathcal{S}^\star$ such that $i_1 \neq i_2$, $s_1 \rightsquigarrow \langle i_1, \rho_1', os_1'\rangle$, $s_2 \rightsquigarrow \langle i_2, \rho_2', os_2'\rangle$, $i \vdash st_1 \Rightarrow st_1'$, $i \vdash st_2 \Rightarrow st_2'$ then $\mathsf{high}(os_1', st_1')$, $\mathsf{high}(os_2', st_2')$ and for all $j \in \mathsf{region}(i)$, $se(j) \not\leq k_{\text{obs}}$.

Lemma 4 ($\text{JVM}_\mathcal{I}$ high step). *Let $\langle i, \rho, os\rangle, \langle i', \rho', os'\rangle \in \text{State}_\mathcal{I}$ be two $\text{JVM}_\mathcal{I}$ states and let $st, st' \in \mathcal{S}^\star$ be two stack types such that $\langle i, \rho, os\rangle \rightsquigarrow \langle i, \rho', os'\rangle$, $i \vdash st \Rightarrow st'$, $se(i) \not\leq k_{\text{obs}}$ and $\mathsf{high}(os, st)$ then $\mathsf{high}(os', st')$.*

Sub-typing lemmas We now present the lemmas relative to the sub-typing notion. They are required to use a monovariant typability notion.

Lemma 5 (High stack type sub-typing). *Let $st_1, st_2 \in \mathcal{S}^\star$ be two stack types such that $st_1 \sqsubseteq st_2$, if $os \in \mathcal{V}^\star$ is an operand stack such that $\mathsf{high}(os, st_1)$ then $\mathsf{high}(os, st_2)$.*

Lemma 6 (indistinguishability double monotony). *Let $st, st_1, st_2 \in \mathcal{S}^\star$ be three stack types such that $st_1 \sqsubseteq st$ and $st_2 \sqsubseteq st$, if $os_1, os_2 \in \mathcal{V}^\star$ are two operand stacks such that $os_1 \sim_{st_1,st_2} os_2$ then $os_1 \sim_{st,st} os_2$.*

Lemma 7 (indistinguishability single monotony). *Let $st_0, st_1, st_2 \in \mathcal{S}^\star$ be three stack types such that $st_1 \sqsubseteq st_2$, if $os, os_0 \in \mathcal{V}^\star$ are two operand stacks such that $os \sim_{st_1,st_0} os_0$ and $\mathsf{high}(os, st_1)$ then $os \sim_{st_2,st_0} os_0$.*

4 $\text{JVM}_\mathcal{O}$: The object-oriented extension of $\text{JVM}_\mathcal{I}$

The object-oriented extension of $\text{JVM}_\mathcal{I}$, namely $\text{JVM}_\mathcal{O}$, includes instance fields, creation of new instances, and null pointers. We assume that programs are not enabled to do pointer arithmetic in $\text{JVM}_\mathcal{O}$ (pointer arithmetic is prevented by standard bytecode verification).

4.1 Programs, memory model and operational semantics

Programs. $\text{JVM}_\mathcal{O}$ programs are as $\text{JVM}_\mathcal{I}$ programs, but also come equipped with a set $\mathcal{C}$ of class names, and a set $\mathcal{F}$ of identifiers representing field names. Programs use an extended set of instructions, given in Figure 5.

$$
\begin{array}{lll}
\mathit{instr} ::= \ldots & & \\
\quad | & \mathsf{new}\ C & \text{create new object in the heap} \\
\quad | & \mathsf{getfield}\ f & \text{load value of field } f \text{ on stack} \\
\quad | & \mathsf{putfield}\ f & \text{store top of stack in field } f
\end{array}
$$

where $C \in \mathcal{C}$ and $f \in \mathcal{F}$.

Fig. 5. Instruction set for $\text{JVM}_{\mathcal{O}}$

States. Compare to $\text{JVM}_{\mathcal{I}}$, the set of $\text{JVM}_{\mathcal{O}}$ values is extended to $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{null\}$, where $\mathcal{L}$ is an (infinite) set of locations and *null* denotes the null pointer. A $\text{JVM}_{\mathcal{O}}$ state is now of the form $\langle i, \rho, os, h\rangle$, where i, ρ, and os are defined as in $\text{JVM}_{\mathcal{I}}$ and h is a heap, that accommodates dynamically created objects. Heaps are modeled as partial functions $h : \mathcal{L} \rightharpoonup \mathcal{O}$, where the set $\mathcal{O}$ of objects is modeled as $\mathcal{C} \times \mathcal{F} \rightharpoonup \mathcal{V}$, i.e. each object $o \in \mathcal{O}$ posses a class (noted $\mathsf{class}(o)$) and a partial function to access field values. We note $o.f$ the access to the value of field f, $o \oplus \{f \mapsto v\}$ denotes the update of an object o at field f with a value v ($h \oplus \{l \mapsto o\}$ is used in the same way for heap update) and Heap is the set of heaps.

Operational semantics. The operational semantics for the new instructions of $\text{JVM}_{\mathcal{O}}$ relies on an allocator function $\mathsf{fresh} : \mathsf{Heap} \to \mathcal{L}$ that given a heap returns the location for that object, and on a function $\mathsf{default} : \mathcal{C} \to \mathcal{O}$ that returns for each class a default object of that class. $\mathsf{default}$ is specified according to the standard Java convention[3]: for all defined field $f \in \mathcal{F}$ of a class $C \in \mathcal{C}$,

$$
\mathsf{default}(C).f = \begin{cases} 0 & \text{if } f \text{ has a numeric type} \\ null & \text{if } f \text{ has a object type} \end{cases}
$$

The semantics is given in Figure 6 as a relation $\rightsquigarrow \subseteq \mathsf{State}_{\mathcal{O}} \times (\mathsf{State}_{\mathcal{O}} + (\mathcal{V} \times \mathsf{Heap}))$.

$$
\frac{P[i] = \mathsf{new}\ C \qquad l = \mathrm{fresh}(h)}{\langle i, \rho, os, h\rangle \rightsquigarrow \langle i+1, \rho, l :: os, h \oplus \{l \mapsto \mathsf{default}(C)\}\rangle}
\qquad
\frac{P[i] = \mathsf{getfield}\ f \qquad l \in dom(h)}{\langle i, \rho, l :: os, h\rangle \rightsquigarrow \langle i+1, \rho, h(l).f :: os, h\rangle}
$$

$$
\frac{P[i] = \mathsf{return}}{\langle i, \rho, v :: os, h\rangle \rightsquigarrow v, h}
\qquad
\frac{P[i] = \mathsf{putfield}\ f \qquad l \in dom(h) \qquad f \in dom(h(l))}{\langle i, \rho, v :: l :: os, h\rangle \rightsquigarrow \langle i+1, \rho, os, h \oplus \{l \mapsto h(l) \oplus \{f \mapsto v\}\}\rangle}
$$

Fig. 6. Operational Semantics for additional $\text{JVM}_{\mathcal{O}}$ instructions

Instruction $\mathsf{new}\ C$ pushes a fresh location on top of the operand stack associated to a new initialized object. The heap is updated with this new object. Instruction $\mathsf{getfield}\ f$ pops a location l from the operand stack. The value of the field f in location l is fetched and pushed onto the operand stack. Instruction $\mathsf{putfield}\ f$ uses the top of the stack to update the object associated with the location in second position on the operand stack. Instruction return now returns the top of the operand stack, and the current heap.

As for $\text{JVM}_{\mathcal{I}}$, we let $\Downarrow$ denote the transitive closure of $\rightsquigarrow$ as in $\text{JVM}_{\mathcal{O}}$ and write $\rho, h \Downarrow v, h'$ as a shorthand for $\langle 1, \rho, \epsilon, h\rangle \Downarrow (v, h')$.

Successor relation. The successor relation is extended with the clause $i \mapsto i + 1$ for all new instructions.

4.2 Non-Interference

Indistinguishability for $\text{JVM}_{\mathcal{O}}$ states is extended and defined relative to a global mapping $\mathsf{ft} : \mathcal{F} \to \mathcal{S}$ that maps fields to security levels. ft will be left implicit in the rest of the paper. In order to extend the notion of

[3] We assume each field f has a declared type.

indistinguishability to heaps we follow [4]. We consider that heaps with different allocations of "high" objects (i.e. objects that have been created in a high security environment) are indistinguishable by an attacker; therefore indistinguishability is defined relative to a bijection β on (a partial set of) locations in the heap. The bijection maps low objects (low objects are objects whose references might be stored in low fields or variables) allocated in the heap of the first state to low objects allocated in the heap of the second state. The objects might be indistinguishable, even if their locations are different during execution. Since values can now also be locations, definition of value indistinguishability is defined also relative to bijection β.

Definition 9 (Value indistinguishability). *Given two values $v_1, v_2 \in \mathcal{V}$, and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ value indistinguishability $v_1 \sim_\beta v_2$ is defined by the clauses:*

$$null \sim_\beta null \qquad \frac{v \in \mathcal{N}}{v \sim_\beta v}$$

$$\frac{v_1, v_2 \in \mathcal{L} \qquad \beta(v_1) = v_2}{v_1 \quad \sim_\beta \quad v_2}$$

Operand stack indistinguishability and local variables indistinguishability are now parameterized by β since values on top of the operand stack and in variables can also be locations.

Definition 10 (Local variables indistinguishability). *For $\rho, \rho' : \mathcal{X} \rightharpoonup \mathcal{V}$ and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, we have $\rho \sim_\beta \rho'$ if ρ and ρ' have the same domain and $\rho(x) \sim_\beta \rho'(x)$ for all $x \in \mathsf{dom}(\rho)$ such that $\boldsymbol{k_v}(x) \leq k_{\text{obs}}$.*

Definition 11 (Operand stack indistinguishability). *Let $os, os' \in \mathcal{V}^\star$, $st, st' \in \mathcal{S}^\star$ and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$. Then $os \sim_{st,st',\beta} os'$ is defined inductively as follows:*

$$\frac{\mathsf{high}(os, st) \qquad \mathsf{high}(os', st')}{os \sim_{st,st',\beta} os'}$$

$$\frac{os \sim_{st,st',\beta} os' \qquad v \sim v' \qquad k \leq k_{\text{obs}}}{v :: os \sim_{k::st,k::st',\beta} v' :: os'}$$

$$\frac{os \sim_{st,st',\beta} os' \qquad k \not\leq k_{\text{obs}} \qquad k' \not\leq k_{\text{obs}}}{v :: os \sim_{k::st,k'::st',\beta} v' :: os'}$$

The definition of object indistinguishability says that two objects are indistinguishable if they have the class and their field values are indistinguishable.

Definition 12 (Object indistinguishability). *Two objects $o_1, o_2 \in \mathcal{O}$ are indistinguishable with respect to a function $\beta \in \mathcal{LL} \rightharpoonup \mathcal{LL}$ if and only if o_1 and o_2 are objects of the same class and for all fields $f \in \mathsf{dom}(o_1)$ such that $\mathsf{ft}(f) \leq k_{\text{obs}}$, $o_1.f \sim_\beta o_2.f$.*

Note that because o_1 and o_2 are objects of the same class we have $\mathsf{dom}(o_1) = \mathsf{dom}(o_2)$ and $o_2(f)$ is well defined.

Heap indistinguishability requires β to be a bijection between the *low domains* (*i.e.* locations that might be reachable from low local variables/fields) of the considered heaps.

Definition 13 (Heap indistinguishability). *Two heaps h_1 and h_2 are indistinguishable with respect to a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, written $h_1 \sim_\beta h_2$, if and only if:*

- *β is a bijection between $\mathsf{dom}(\beta)$ and $\mathsf{rng}(\beta)$;*
- *$\mathsf{dom}(\beta) \subseteq \mathsf{dom}(h_1)$ and $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$;*
- *for every $l \in \mathsf{dom}(\beta)$, $h_1(l) \sim_\beta h_2(\beta(l))$;*

As in $\text{JVM}_{\mathcal{I}}$, state indistinguishability can then be defined component-wise on state structure.

Definition 14 (State indistinguishability). *Two states* $\langle i, \rho, os, h\rangle$ *and* $\langle i', \rho', os', h'\rangle$ *are indistinguishable with respect to a partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ *and two stack types* $st, st' \in \mathcal{S}^\star$*, denoted* $\langle i, \rho, os, h\rangle \sim_{st,st',\beta} \langle i', \rho', os', h'\rangle$*, iff* $os \sim_{st,st',\beta} os'$*,* $\rho \sim_\beta \rho'$ *and* $h \sim_\beta h'$ *hold.*

Finally, non-interference in JVM$_\mathcal{O}$ is extended using the relations defined above.

Definition 15 (Non-interferent JVM$_\mathcal{O}$ program). *A program* P *is* non-interferent *w.r.t. its policy* $\boldsymbol{k_v} \longrightarrow k_r$*, if for every partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ *and every* $\rho_1, \rho_2 \in \mathcal{X} \rightharpoonup \mathcal{V}$*,* $h_1, h_2, h_1', h_2' \in \mathsf{Heap}$*,* $v_1, v_2 \in \mathcal{V}$ *such that* $\rho_1, h_1 \Downarrow v_1, h_1'$*,* $\rho_2, h_2 \Downarrow v_2, h_2'$ *and* $h_1 \sim_\beta h_2$*,* $\rho_1 \sim_{\boldsymbol{k_v},\beta} \rho_2$*, there exists a partial function* $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ *such that* $\beta \subseteq \beta'$*,* $h_1 \sim_{\beta'} h_2$ *and* $k_r \leq k_{\text{obs}}$ *implies* $v_1 \sim_{\beta'} v_2$*.*

Here $\beta \subseteq \beta'$ means that $\mathsf{dom}(\beta) \subseteq \mathsf{dom}(\beta')$ and for all locations $l \in \mathsf{dom}(\beta)$, $\beta(l) = \beta'(l)$. The definition of non-interference allows for β to be extended, in order to handle objects that are dynamically created during execution.

4.3 Typing rules

The abstract transition system of the JVM$_\mathcal{O}$ extends that of the JVM$_\mathcal{I}$ with the typing transfer rules of Figure 7. As in JVM$_\mathcal{I}$, all rules are implicitly parameterized by a cdr region, a security environment se and a signature $\boldsymbol{k_a} \longrightarrow k_r$.

$$\frac{P[i] = \mathsf{new}\ C}{i \vdash st \Rightarrow se(i) :: st}$$

$$\frac{P[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f)}{i \vdash k_1 :: k_2 :: st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{getfield}\ f}{i \vdash k :: st \Rightarrow (k \sqcup \mathsf{ft}(f) \sqcup se(i)) :: st}$$

Fig. 7. Additional typing transfer rules for JVM$_\mathcal{O}$

- The transfer rule for new adds to the stack type the security level of the current program point, which imposes a constraint on security level from which the object can be accessed. For example, if new is executed in a high security environment, then the reference to the object cannot be accessed from a low variable. However, if the object is created in a low security environment it can either be stored in a high or low variable/field.
- The transfer rule for putfield requires that $k_1 \leq \mathsf{ft}(f)$ (where k_1 is the security type of the object of the field) in order to prevent an explicit flow from a high value to a low field. The constraint $se(i) \leq \mathsf{ft}(f)$ prevents an implicit flow caused by an assignment to a low field in a high security environment. Finally, the constraint $k_2 \leq \mathsf{ft}(f)$ prevents modifying low fields of high objects that are alias to a low object.

  The following exampleillustrates this last point. It corresponds to a source program like

```
C xL = new C();
zH = yH ? new C() : xL;
zH.fL = 1;
```

We assume that C is a class that has a low field named f_L. Let x_L be a low variable and y_H, z_H high variables.

$$
\begin{array}{ll}
 & \mathsf{new}\ C \\
 & \mathsf{store}\ x_L \\
 & \mathsf{load}\ y_H \\
l_1 : & \mathsf{ifeq}\ l_2 \\
 & \left.\begin{array}{l}\mathsf{new}\ C \\ \mathsf{goto}\ l_3\end{array}\right\}\ \mathsf{region}(l_1) \\
l_2 : & \mathsf{load}\ x_L \\
l_3 : & \mathsf{store}\ z_H \\
 & \mathsf{load}\ z_H \\
 & \mathsf{push}\ 1 \\
 & \mathsf{putfield}\ f_L
\end{array}
$$

In this program, depending on the test on y_H, variable x_L and z_H might be aliases to the same object (of class C). Hence, the assignment to field f_L might have side effect on the object in x_L. This program is rejected thanks to the putfield rule which avoids this type of leaks due to alias (with the constraint $k_2 \leq \mathsf{ft}(f)$ preventing assignments to low fields from high target objects).

– In the rule for getfield f the value pushed on the operand stack has a security level at least greater than $\mathsf{ft}(f)$ and the level k of the location (to prevent explicit flows) and at least greater than $se(i)$ for implicit flows.

4.4 Type system soundness

Theorem 2. *Let P be a* $\mathrm{JVM}_{\mathcal{O}}$ *program and* $(\mathsf{region}, \mathsf{jun})$ *a safe cdr for P (according to SOAP properties). Suppose P is typable with respect to* region *and to a signature $\boldsymbol{k_v} \longrightarrow k_r$. Then P is non-interferent with respect to the policy associated with $\boldsymbol{k_v} \longrightarrow k_r$.*

The soundness proof closely follows the $\mathrm{JVM}_{\mathcal{I}}$ soundness proof , except that now we have to manipulate heaps and some partial function β.

We finish this section by stating the four basic lemmas necessary for the soundness proof of theorem 2.

Lemma 8 ($\mathrm{JVM}_{\mathcal{O}}$ locally respect). *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $s_1, s_2 \in \mathrm{State}_{\mathcal{O}}$ two* $\mathrm{JVM}_{\mathcal{O}}$ *states at the same program point i and two stack types $st_1, st_2 \in \mathcal{S}^\star$ such that $s_1 \sim_{st_1, st_2, \beta} s_2$.*

If $s_1', s_2' \in \mathrm{State}_{\mathcal{O}}$ and two stack types $st_1', st_2' \in \mathcal{S}^\star$ such that $s_1 \rightsquigarrow s_1'$, $s_2 \rightsquigarrow s_2'$, $i \vdash st_1 \Rightarrow st_1'$ and $i \vdash st_2 \Rightarrow st_2'$ then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $s_1' \sim_{st_1', st_2'} s_2'$ and $\beta \subseteq \beta'$.

If $v_1, v_2 \in \mathcal{V}$ and $s_1 \rightsquigarrow v_1$, $s_2 \rightsquigarrow v_2$, $i \vdash st_1 \Rightarrow$ and $i \vdash st_2 \Rightarrow$ then $k_r \leq k_{\mathrm{obs}}$ implies $v_1 \sim_\beta v_2$.

In the first case, the partial function β may be extended if $P[i]$ is of the form $\mathsf{new}\ C$ and the context is low ($se(i) \leq k_{\mathrm{obs}}$).

Lemma 9 ($\mathrm{JVM}_{\mathcal{O}}$ step consistent). *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $\langle i, \rho, os, h\rangle, \langle i_0, \rho_0, os_0, h_0\rangle \in \mathrm{State}_{\mathcal{O}}$ two* $\mathrm{JVM}_{\mathcal{O}}$ *states and two stack types $st, st_0 \in \mathcal{S}^\star$ such that $\langle i, \rho, os, h\rangle \sim_{st, st_0, \beta} \langle i_0, \rho_0, os_0, h_0\rangle$, $se(i) \not\leq k_{\mathrm{obs}}$ and $\mathsf{high}(os, st)$.*

If a state $\langle i', \rho', os', h'\rangle \in \mathrm{State}_{\mathcal{O}}$ and a stack type $st' \in \mathcal{S}^\star$ such that $\langle i, \rho, os, h\rangle \rightsquigarrow \langle i, \rho', os', h'\rangle$ and $i \vdash st \Rightarrow st'$ then $\langle i', \rho', os', h'\rangle \sim_{st', st_0, \beta} \langle i_0, \rho_0, os_0, h_0\rangle$.

If there exists a value $v \in \mathcal{V}$ such that $\langle i, \rho, os, h\rangle \rightsquigarrow v$ and $i \vdash st \Rightarrow$ then $h' \sim_\beta h_0$ and $k_r \not\leq k_{\mathrm{obs}}$.

Note that we do not need to extend partial function β when the step occurs in a high context ($se(i) \not\leq k_{\mathrm{obs}}$).

Lemma 10 ($\mathrm{JVM}_{\mathcal{O}}$ high branching). *Let β be a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $s_1, s_2 \in \mathrm{State}_{\mathcal{O}}$ two* $\mathrm{JVM}_{\mathcal{O}}$ *states at the same program point i and two stack types $st_1, st_2 \in \mathcal{S}^\star$ such that $s_1 \sim_{st_1, st_2, \beta} s_1'$.*

Let $\langle i_1, \rho_1', os_1', h_1'\rangle, \langle i_2, \rho_2', os_2', h_2'\rangle \in \mathrm{State}_{\mathcal{O}}$ be two states and let $st_1', st_2' \in \mathcal{S}^\star$ be two stack types such that $i_1 \neq i_2$, $s_1 \rightsquigarrow \langle i_1, \rho_1', os_1', h_1'\rangle$, $s_2 \rightsquigarrow \langle i_2, \rho_2', os_2', h_2'\rangle$. If $i \vdash st_1 \Rightarrow st_1'$, $i \vdash st_2 \Rightarrow st_2'$ then $\mathsf{high}(os_1', st_1')$, $\mathsf{high}(os_2', st_2')$ and for all $j \in \mathsf{region}(i)$, $se(j) \not\leq k_{\mathrm{obs}}$.

Lemma 11 (JVM$_\mathcal{O}$ high step). *Let* $\langle i, \rho, os, h\rangle, \langle i', \rho', os', h'\rangle \in \text{State}_\mathcal{O}$ *two* JVM$_\mathcal{O}$ *states and two stack types* $st, st' \in \mathcal{S}^\star$ *such that* $\langle i, \rho, os, h\rangle \rightsquigarrow \langle i, \rho', os', h'\rangle$, $i \vdash st \Rightarrow st'$, $se(i) \not\leq k_{\text{obs}}$ *and* $\mathsf{high}(os, st)$ *then* $\mathsf{high}(os', st')$.

5 JVM$_\mathcal{C}$: The Method Extension of JVM$_\mathcal{O}$

The purpose of this section is to extend our analysis to methods. The extension is compatible with bytecode verification, in the sense that the analysis is modular.

5.1 Programs, memory model and operational semantics

Programs. Each program comes equipped with a set $\mathcal{M}$ of method names, and a set $\mathcal{C}$ of classes, as in JVM$_\mathcal{O}$. The set of classes is now organised as a hierarchy to model the inheritance of class. This hierarchy will be used to resolve virtual calls.

Each method m posseses a list of instructions P_m. For simplicity, we impose that all methods return a value. The set of instructions of JVM$_\mathcal{O}$ is extended with the new instruction $\mathsf{invokevirtual}\ m_{\text{ID}}$ for calling a virtual method. Here m_{ID} is a method identifier which may correspond to several methods in the class hierarchy according to overriding of methods. We assume there is a function lookup_P attached to each program P that takes a method identifier and a class name and returns the method to be executed.

States. While JVM states contain a frame stack to handle method invocations, it is convenient for showing the correctness of static analyzes to rely on an equivalent semantics where method invocation is performed in one big step transition. Hence a JVM$_\mathcal{C}$ state is defined as in JVM$_\mathcal{O}$.

Operational semantics. While small-step semantics uses a call stack to store the calling context and retrieve it during a return instruction, the big step semantics directly calls the full evaluation of the called method from an initial state to a return value and uses it to continue the current computation. The big-step operational semantics is given in Figure 8. As can be seen in the first rule, semantics of instructions is like in JVM$_\mathcal{O}$,

$$\frac{P_m[i] = ins \qquad ins \neq \mathsf{invokevirtual}\ m_{\text{ID}} \qquad \langle i, \rho, os, h\rangle \rightsquigarrow_{\text{JVM}_\mathcal{O}} \langle i', \rho', os', h'\rangle}{\langle i, \rho, os, h\rangle \overset{(0)}{\rightsquigarrow}_m \langle i', \rho', os', h'\rangle}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\text{ID}} \qquad m' = \mathsf{lookup}_P(m_{\text{ID}}, \mathsf{class}(h(l))) \\ l \in dom(h) \qquad \mathsf{length}(os_1) = \mathsf{nbArguments}(m_{\text{ID}}) \\ \langle 1, \{this \mapsto l, \boldsymbol{x} \mapsto os_1\}, \epsilon, h\rangle \Downarrow_{m'}^{(n)} v, h' \end{array}}{\langle i, \rho, os_1 :: l :: os_2, h\rangle \overset{(n+1)}{\rightsquigarrow}_m \langle i+1, \rho, v :: os_2, h'\rangle}$$

$$\frac{\langle i, \rho, os, h\rangle \rightsquigarrow_{\text{JVM}_\mathcal{O}} v}{\langle i, \rho, os, h\rangle \Downarrow_m^{(0)} v} \qquad \frac{\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_m \langle j, \rho', os', h'\rangle \quad \langle j, \rho', os', h'\rangle \Downarrow_m^{(p)} v}{\langle i, \rho, os, h\rangle \Downarrow_m^{(n+p)} v}$$

Fig. 8. Operational Semantics for JVM$_\mathcal{C}$

except for the new instruction $\mathsf{invokevirtual}$. The second rule gives the semantics of the virtual call. The location l is used to resolve the virtual call. Thanks to the class of l and the identifier m_{ID}, a method m' is found in the class hierarchy (trough the lookup operator). The transitive closure of $\rightsquigarrow_m$ is then used to obtain the result of the execution of m'. Execution of m' is initialised with location l for the reserved variable *this* and the elements of the operand stack os_1 for the other variables[4].

[4] We assume all other variable used for local computation in the method are initialised by a default value according to their type

We opt for a big-step operational semantics because it simplifies the notion of indistinguishability between states. In the presence of a small-step semantics states possess stack of frames (one frame corresponding to each method in the calling chain) and hence indistiguishability must take account of frames of high and low methods which can throw and propagate low and high exceptions. It is also needed for indistiguishability to state if the method is invoked in a low or high target object. Using this alternative semantics has brought a significant simplification to the proofs of the analysis.

The relation $\rightsquigarrow$ is now parametrised by a counter representing the number of method calls occurring between two consecutive steps. The transitive closure of $\rightsquigarrow$ is simultaneously defined with a similar counter.

As in other JVM fragments, we note $\rho, h \Downarrow_m^{(n)} v, h'$ when $\langle 1, \rho, \epsilon, h\rangle \Downarrow_m^{(n)} v, h'$. We note $\rho, h \Downarrow_m v, h'$ for $\exists n,\ \rho, h \Downarrow_m^{(n)} v, h'$.

Successor relation. We extend the successor relation of $\text{JVM}_{\mathcal{O}}$ with the clause $i \mapsto i{+}1$ for the new instruction invokevirtual. It illustrates our modular verification technique : *cdr* is computed method by method.

5.2 Non-Interference

Non-Interference for a $\mathsf{JVM}_{\mathcal{C}}$ program is given by local policies defined by security signatures for every method and a global policy defined by a mapping of fields to security levels, namely ft.

As mentioned in Section 2, method signatures are of the form

$$\boldsymbol{k_v} \xrightarrow{k_h} k_r$$

where $\boldsymbol{k_v}$ provides the security level of the method arguments (and to all intermediate variables used in the method), k_h is the effect of the method on the heap and k_r is the security level of the result of the method.

The *heap effect level* k_h is needed to make a modular analysis. It represents a lower bound for security levels of fields that are affected during execution of the method.

A method is allowed to perform field updates only on fields whose level is greater than k_h. We formally define this notion of *side effect preorder*.

Definition 16 (Side effect preorder). *Two heaps $h_1, h_2 \in \mathsf{Heap}$ are* side effect preordered *with respect to a security level $k \in \mathcal{S}$ (noted $h_1 \preceq_k h_2$) if and only if $\mathsf{dom}(h_1) \subseteq \mathsf{dom}(h_2)$ and for all location $l \in \mathsf{dom}(h_1)$ and all fields $f \in \mathcal{F}$ such that $k \not\leq \mathsf{ft}(f)$, $h_1(l).f = h_2(l).f$.*

This allows to define the notion of *side-effect-safe method*.

Definition 17. *A method m is* side-effect-safe *with respect to a security level k_h if for all local variable $\rho \in \mathcal{X} \rightharpoonup \mathcal{V}$, all heaps $h, h' \in \mathsf{Heap}$ and value $v \in \mathcal{V}$, $\rho, h \Downarrow_m v, h'$ implies $h \preceq_{k_h} h'$.*

The notion of non-interferent method can be stated using the same indistiguishability relation as in $\text{JVM}_{\mathcal{O}}$. A method m is called *non-interferent for signature $\boldsymbol{k_v} \longrightarrow k_r$* if every time it is executed with indistinguishable arguments according to $\boldsymbol{k_v}$ and indistiguishable heaps according to the global policy ft, then the results of the method by normal termination are indistinguishable by k_r and its heaps are indistiguishable according to the global policy.

Definition 18 (Non-interferent $\text{JVM}_{\mathcal{C}}$ method). *A method m is* non-interferent *w.r.t. a policy $\boldsymbol{k_v} \longrightarrow k_r$, if for every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every $\rho_1, \rho_2 \in \mathcal{X} \rightharpoonup \mathcal{V}$, $h_1, h_2, h_1', h_2' \in \mathsf{Heap}$, $v_1, v_2 \in \mathcal{V}$ such that $\rho_1, h_1 \Downarrow_m v_1, h_1'$, $\rho_2, h_2 \Downarrow_m v_2, h_2'$ and $h_1 \sim_\beta h_2$, $\rho_1 \sim_{\boldsymbol{k_v}, \beta} \rho_2$, there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$, $h_1 \sim_{\beta'} h_2$ and $k_r \leq k_{\text{obs}}$ implies $v_1 \sim_{\beta'} v_2$.*

The notion of safe method is then defined by conjunction of the two previous definitions.

Definition 19 (Safe $\text{JVM}_{\mathcal{C}}$ method). *A method m is* safe *w.r.t. a policy $\boldsymbol{k_v} \xrightarrow{k_h} k_r$ if m is side-effect-safe with respect to k_h and m is non-interferent with respect to $\boldsymbol{k_v} \longrightarrow k_r$.*

Let Γ be a table of method signatures. This table associates to each method identifier[5] m_{ID} and security level $k \in \mathcal{S}$, a security signature $\Gamma_m[k]$. This signature gives the security policy of the method m called on object of level k (as in [] for source program). The set of security signature of a method m is defined as $\mathrm{Policies}_\Gamma(\mathrm{m}) = \{\ \Gamma_{\mathrm{m}}[\mathrm{k}] \mid \mathrm{k} \in \mathcal{S}\ \}$. In the rest of the paper Γ will often be left implicit. We use it to define the notion of *safe program*.

Definition 20 (Safe $\mathrm{JVM}_{\mathcal{C}}$ program). *A program is* safe *with respect to a table of method signatures Γ if for all its method m, m is safe with respect to all policies in* $\mathrm{Policies}_\Gamma(\mathrm{m})$.

Example 1. Let P be a program that includes a method m and a class C with field f. Let m have variables x_1, x_2, y and no handlers defined. Let the non-interference policy for P be given by a security signature $L, L, H \xrightarrow{L} H$ for m and a security signature $\xrightarrow{L} L$ for **main**, and a global mapping ft such that $\mathsf{ft}(f) = L$.

If the code of m is defined by:

```
    new C
    store x2
    load x1
    ifeq l1
    load x2
    push 1
    putfield f
l1  load x2
    getfield f
    return
```

then method m is non-interferent because: starting from equal values for x (x represents the low part of the state, since the security signature says that x is low), the result of the method will always be the value of the low field f that is 1, hence every results are indistiguishable by L, as stated in the signature of m. The method cannot terminate abnormally since new C is not a null value and there are no affections to the high fields. This respects the low write effect of the method required by the policy.

Now assume that in program point 6 there is an instruction load y instead of load x_2. Since according to the local policy, y is a high variable, its value might be different for 2 different indistinguishable states. For example assume that for two states with value of x equal to 1 variable y has values *null* and a location in the domain of the heap that points to an object of class C. Then m will terminate abnormally in the first case, and normally in the second. Since this depends on the value of high variable y, the results are not indistinguishable by L as stated by the security signature (exceptional effect). Hence, m is interferent.

5.3 Typing rules

The information flow type system enforces a method-wise verification strategy, using method signatures in the transfer rule for method invocation. All typing rules are those of the $\mathrm{JVM}_{\mathcal{O}}$ typing rules, except for putfield which needs a modification and the virtual call rule which is new. This two rules are given in Figure 9.

Concerning putfield only one constraint is added w.r.t. the previous $\mathrm{JVM}_{\mathcal{O}}$ rule. The new constraint $k_h \leq \mathsf{ft}(f)$ prevents modification of fields with a level not greater than the heap effect of the current method.

The typing rule for virtual call contains several constraints. The heap effect level of the called method is constrained in several ways. The goal of the constraint $k \leq k'_h$ is to avoid invocation of methods with low effect on the heap with high target objects. Two different target objects (in two executions) may mean that the body of the method to be executed is different in each execution. If the effect of the method is low ($k_h \leq k_{\mathrm{obs}}$), then low memory is differently modified in both executions, leading to information leak. The constraint $se(i) \leq k'_h$ prevents implicit flows (low assignment in high regions) during execution of the called method. The constraint $k_h \leq k'_h$ prevents the called method to update field with a level lower that k_h.

[5] Associating signatures with method identifiers instead of methods allows to enforce that method overriding preserves declared security signatures.

$$\frac{P_m[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f) \qquad k_h \leq \mathsf{ft}(f)}{\mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} k_r, i \vdash k_1 :: k_2 :: st \Rightarrow st}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \boldsymbol{k'_a} \xrightarrow{k'_h} k'_r \\ k \sqcup k_h \sqcup se(i) \leq k'_h \qquad \mathsf{length}(st_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \boldsymbol{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \boldsymbol{k'_a}[i+1] \end{array}}{\mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} k_r, i \vdash st_1 :: k :: st_2 \Rightarrow (k'_r \sqcup k \sqcup se(i)) :: st_2}$$

Fig. 9. New transfer rules for instructions of JVM$_{\mathcal{C}}$

The security level of the return value is $(k'_r \sqcup k \sqcup se(i))$. The security level k'_r in $(k'_r \sqcup k \sqcup se(i))$, obtained from the signature of m_{ID} , prevents that its result flows to variables or fields with lower security level. The security level k prevents flows due to execution of two distinct methods.

We include here an example that illustrates how object-oriented features can lead to interference. We refer to [4] for further examples.

Example 2. Let class C be a super class of a class D. Let method *foo* be declared in D, and a method m declared in C and overridden in D as illustrated by the following source program[6]:

```
class C {
  int m() {return 0;}
}
class D extends C {
  int m() {return 1;}
  int foo(boolean y_H) {return (y_H ? new C() : this).m();}
}
```

```
D.foo :              C.m :     D.m :
    load y_H         push 0    push 1
    ifeq l_1         return    return
    new C
    goto l_2
l_1 : load this
l_2 : invokevirtual m
    return
```

At run time, either code $C.m$ or code $D.m$ is executed depending on the value of high variable y_H. Information about y_H may be inferred by observing the return value of method m.

5.4 Type system soundness

Theorem 3. *Let P be a* JVM$_{\mathcal{C}}$ *program, Γ a table of signatures and for all method m in P,* $(\mathsf{region}_m, \mathsf{jun}_m)$ *a safe cdr for m (according to SOAP properties). Suppose all methods m in P are typable with respect to* region_m *and to all signatures in* $\mathrm{Policies}_{\Gamma}(\mathrm{m})$. *Then P is safe with respect to Γ.*

We assume now P is program with an associated signature table Γ.

[6] We omit the call of the initialiser.

Side-effect safety. The first part of the soundness proof consist in proving that all methods of a typable program are side-effect-safe.

In this paragraph m is suppose to be a method of P, region a cdr function for m, se a security environment, and $\boldsymbol{k_a} \xrightarrow{k_h} k_r$ a security signature.

We show that all instruction step transforms a heap h into a heap h' such that $h \preceq_{k_h} h'$. In this first lemma neither virtual call or return instructions are considered.

Lemma 12. *Let $\langle i, \rho, os, h\rangle, \langle i, \rho', os', h'\rangle \in \mathrm{State}_{\mathrm{JVM}_\mathcal{C}}$ be two states such that*

$$\langle i, \rho, os, h\rangle \overset{(0)}{\rightsquigarrow}_m \langle i', \rho', os', h'\rangle$$

Let two stack types $st, st' \in \mathcal{S}^\star$ such that

$$\mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} k_r, i \vdash st \Rightarrow st'$$

then $h \preceq_{k_h} h'$.

The next lemma treats the case of the return instruction.

Lemma 13. *Let a method m and $\langle i, \rho, os, h\rangle \in \mathrm{State}_{\mathrm{JVM}_\mathcal{C}}$ a state, $h' \in \mathsf{Heap}$ a heap and $v \in \mathcal{V}$ a value such that*

$$\langle i, \rho, os, h\rangle \overset{(0)}{\rightsquigarrow}_m v, h'$$

Let a stack type $st \in \mathcal{S}^\star$ such that

$$\mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} k_r, i \vdash st \Rightarrow$$

then $h \preceq_{k_h} h'$.

For the special case of virtual call we need an inductive hypothesis about the side-effect safety of all methods in P. We hence introduce the notion of *Side-effect-safe at order n*.

Definition 21 (Side-effect-safe at order n). *A method m is* side-effect-safe at order n *with respect to a security level k_h if for all state $\langle i, \rho, os, h\rangle \in \mathrm{State}_{\mathrm{JVM}_\mathcal{C}}$, all heap $h' \in \mathsf{Heap}$ and value $v \in \mathcal{V}$, $\langle i, \rho, os, h\rangle \Downarrow_m^{(n)} v, h'$ implies $h \preceq_{k_h} h'$.*

Lemma 14. *Let n an integer and suppose all method m' in P are side-effect-safe at order n with respect to the heap effect level of all the policies in* $\mathrm{Policies}_\Gamma(\mathrm{m}')$.

Let $\langle i, \rho, os, h\rangle, \langle i, \rho', os', h'\rangle \in \mathrm{State}_{\mathrm{JVM}_\mathcal{C}}$ two states such that

$$\langle i, \rho, os, h\rangle \overset{(n+1)}{\rightsquigarrow}_m \langle i', \rho', os', h'\rangle$$

Let two stack types $st, st' \in \mathcal{S}^\star$ such that

$$\mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} k_r, i \vdash st \Rightarrow st'$$

then $h \preceq_{k_h} h'$.

We then can conclude about the side-effect safety of all methods in P, using an induction on the number of virtual call on semantics derivation and an induction on the derivation length.

Lemma 15. *For all method m in P, let $(\mathsf{region}_m, \mathsf{jun}_m)$ be a safe cdr for m. Suppose all methods m in P are typable with respect to region_m and to all signatures in* $\mathrm{Policies}_\Gamma(\mathrm{m})$. *Then all method m is side-effect-safe with respect to the heap effect level of all the policies in* $\mathrm{Policies}_\Gamma(\mathrm{m})$.

Non-interference. The soundness proof for non-interference reuses all the basic lemmas proved for $\text{JVM}_{\mathcal{O}}$ which are still valid for the instruction of the $\text{JVM}_{\mathcal{O}}$ (except invokevirtual). The virtual call requires specific lemmas given below.

Definition 22 (non-interference at order n). *A method m is* non-interferent at order n *with respect to a security signature $\boldsymbol{k_v} \longrightarrow k_r$, if for every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every states $\langle 1, \rho_1, \epsilon_1, h_1\rangle, \langle 1, \rho_2, \epsilon, h_2\rangle \in \text{State}_{\text{JVM}_{\mathcal{C}}}$, every heaps $h_1', h_2' \in \mathsf{Heap}$, every values $v_1, v_2 \in \mathcal{V}$ and every integer $n_1, n_2 \in \mathbb{N}$ such that $\langle i, \rho_1, os_1, h_1\rangle \Downarrow_m^{(n_1)} v_1, h_1'$, $\langle i, \rho_2, os_2, h_2\rangle \Downarrow_m^{(n_2)} v_2, h_2'$, $n_1 \leq n$, $n_2 \leq n$ and $\langle 1, \rho_1, \epsilon, h_1\rangle \sim_{\beta,\epsilon,\epsilon} \langle 1, \rho_2, \epsilon, h_2\rangle$ there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$, $h_1' \sim_{\beta'} h_2'$ and $k_r \leq k_{\text{obs}}$ implies $v_1 \sim_{\beta'} v_2$.*

Lemma 16 ($\text{JVM}_{\mathcal{C}}$ locally respect for virtual calls). *Let n an integer, let P a program and a table of signature Γ such that all its method m' are non-interferent at order n, with respect to all the policies in* $\text{Policies}_{\Gamma}(\text{m}')$ *and side-effect-safe with respect to the heap effect level of all the policies in* $\text{Policies}_{\Gamma}(\text{m}')$.

Let m be a method in P, $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ a partial function, $s_1, s_2 \in \text{State}_{\mathcal{C}}$ two $\text{JVM}_{\mathcal{C}}$ states at the same program point i and two stack types $st_1, st_2 \in \mathcal{S}^\star$ such that $s_1 \sim_{st_1,st_2,\beta} s_2$.

If there exist two states $s_1', s_2' \in \text{State}_{\mathcal{C}}$ and two stack types $st_1', st_2' \in \mathcal{S}^\star$ such that $s_1 \rightsquigarrow_m^{(n_1+1)} s_1'$ with $n_1 \leq n$, $s_2 \rightsquigarrow_m^{(n_2+1)} s_2'$ with $n_2 \leq n$, $\Gamma, \mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} k_r, i \vdash st_1 \Rightarrow st_1'$ and $\Gamma, \mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} k_r, i \vdash st_2 \Rightarrow st_2'$ then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $s_1' \sim_{st_1',st_2',\beta'} s_2'$ and $\beta \subseteq \beta'$.

Lemma 17 ($\text{JVM}_{\mathcal{C}}$ step consistent for virtual calls). *Let P be a program and a table of signature Γ such that all its method m' are side-effect-safe with respect to the heap effect level of all the policies in* $\text{Policies}_{\Gamma}(\text{m}')$. *Let m a method in P, β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $\langle i, \rho, os, h\rangle, s_0 \in \text{State}_{\mathcal{C}}$ two $\text{JVM}_{\mathcal{C}}$ states, and two stack types $st, st_0 \in \mathcal{S}^\star$ such that $\langle i, \rho, os, h\rangle \sim_{st,st_0,\beta} s_0$, $se(i) \not\leq k_{\text{obs}}$ and $\mathsf{high}(os, st)$.*

If there exists a state $\langle i', \rho', os', h'\rangle \in \text{State}_{\mathcal{C}}$ and a stack type $st' \in \mathcal{S}^\star$ such that $\langle i, \rho, os, h\rangle \rightsquigarrow_m^{(n+1)} \langle i, \rho', os', h'\rangle$ and $\Gamma, \mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} k_r, i \vdash st \Rightarrow st'$ then $\langle i', \rho', os', h'\rangle \sim_{st',st_0,\beta} s_0$.

Virtual call is not a branching source in $\text{JVM}_{\mathcal{C}}$ so no high branching lemma is required for this instruction.

Lemma 18 ($\text{JVM}_{\mathcal{C}}$ high step for virtual calls). *Let m a method, let two $\text{JVM}_{\mathcal{C}}$ states $\langle i, \rho, os, h\rangle, \langle i', \rho', os', h'\rangle \in \text{State}_{\mathcal{C}}$ and two stack types $st, st' \in \mathcal{S}^\star$ such that $\langle i, \rho, os, h\rangle \rightsquigarrow_m^{(n+1)} \langle i, \rho', os', h'\rangle$, $\Gamma, \mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} k_r, i \vdash st \Rightarrow st'$, $se(i) \not\leq k_{\text{obs}}$ and $\mathsf{high}(os, st)$ then $\mathsf{high}(os', st')$.*

6 $\text{JVM}_{\mathcal{G}}$: The exception-handling extension of $\text{JVM}_{\mathcal{C}}$

In this section we show how $\text{JVM}_{\mathcal{C}}$ is extended with an exception handling mechanism.

The extension of the type system to multiple exceptions is achieved by a fine-grained definition of control dependence regions that is parameterized by a class-analysis and an exception-analysis. The class analysis returns an over-approximation of classes of exceptions of a program point, and the exception analysis which are the (maybe) escaping exceptions of a method. For the soundness of the information flow type system, we assume that both the class-analysis and the exception-analysis are in the trusted computing base. Thus, the type system exploits the information of the class analysis and signature of methods (that coincides with the exception-analysis results) to add contraints on the security environment according to adequate regions for the type of escaping exceptions (if any).

6.1 Programs, memory model and operational semantics

Programs Programs are similar to those in the $\text{JVM}_{\mathcal{C}}$ model. However, the instruction set of the $\text{JVM}_{\mathcal{C}}$ is extended with the bytecode throw.

Furthermore, we assume that programs come equipped with a partial function[7] $\mathsf{Handler}_m : \mathcal{PP} \times \mathcal{C} \rightharpoonup \mathcal{PP}$ that for each method m selects the appropriate handler for a given program point. If an exception of class

[7] This opaque handling function hides the notions of handler list and sub-class used in Java.

$C \in \mathcal{C}$ is thrown at program point $i \in \mathcal{PP}$ then, if $\mathsf{Handler}_m(i) = t$, then the control will be transfered to program point t, and if $\mathsf{Handler}_m(i, C)$ is undefined (noted $\mathsf{Handler}_m(i, C) \uparrow$), the exception is uncaught in method m.

States $\text{JVM}_\mathcal{G}$ states are those of $\text{JVM}_\mathcal{C}$ except for final states which can now correspond to uncaught exceptions. We hence model final states as $(\mathcal{V} + \mathcal{L}) \times \mathsf{Heap}$: a final is either of the form $(v, h) \in \mathcal{V} \times \mathsf{Heap}$ for normal termination, or of the form $(\langle l \rangle, h) \in \mathcal{L} \times \mathsf{Heap}$ for abrupt termination by an uncaught exception pointed by a location l in the heap h.

Operational semantics We give in Figure 10 the semantics of exception-throwing instructions in $\mathsf{JVM}_\mathcal{G}$. All they other rules of $\text{JVM}_\mathcal{C}$ stay in the same form. There are three more rules for the virtual call instruction. The first and the second model the cases where execution of the called method terminates by an uncaught exception. In first rule the thrown exception is caught in method m while in second rule it is uncaught and m then terminates abnormally. In both cases, we impose that thrown exception has been statically predicted by the result $\mathsf{excAnalysis}(m_{\text{ID}})$ of the exception analysis[8]. The third rule corresponds to a null pointer exception thrown because the virtual call was asked on a null reference. We note **np** the Java class associated to the null pointer exception. When a native exception **np** is thrown the catching mechanism is model by the function $\mathsf{RuntimeExceptionHandling}$. Each instruction which performs access on reference ($\mathsf{getfield}\ f$, $\mathsf{putfield}\ f$ and throw) have now similar semantics rules. The last two rules concern the new instruction throw which throws the exception pointed by the reference on top of the stack. Transition are now parameterised by a tag $\tau \in \{\emptyset\} + \mathcal{C}$ to describe the nature of the transition (see the successor relation below). We will sometimes omit the tag τ in the notation $\overset{(n)}{\rightsquigarrow}_{m,\tau}$ for clarity.

Successor relation The successor relation is now decorated by an element (called *tag*) in $\{\emptyset\} + \mathcal{C}$ in order to reflect the nature of the underlying semantics step: $\emptyset$ for a normal step (as in $\text{JVM}_\mathcal{C}$) and $c \in \mathcal{C}$ for a step where an exception of class C has been thrown. The definition of this new relation is given in Figure 11. This relation can be statically computed thanks to the handler function of each method. Successors of a throw instruction are approximated thanks to the class analysis result and successors of a $\mathsf{invokevirtual}$ thanks to the exception analysis result of the called method.

SOAP properties Cdr results are now associated not only to program points but also to tags:

$$\mathsf{region}_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \to \wp(\mathcal{PP}) \qquad \mathsf{jun}_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \rightharpoonup \mathcal{PP}$$

We call *return point* a point i such that there exists $\tau \in \{\emptyset\} + \mathcal{C}$ with $i \mapsto^\tau$. When possible we will write $i \mapsto j$ for $\exists \tau,\ i \mapsto^\tau j$.

SOAP1: for all program points i, j, k and tag τ such that $i \mapsto j$, $i \mapsto^\tau k$ and $j \neq k$ (i is hence a branching point), $k \in \mathsf{region}(i, \tau)$ or $k = \mathsf{jun}(i, \tau)$;
SOAP2: for all program points i, j, k and tag τ, if $j \in \mathsf{region}(i, \tau)$ and $j \mapsto k$, then either $k \in \mathsf{region}(i, \tau)$ or $k = \mathsf{jun}(i, \tau)$;
SOAP3: for all program points i, j and tag τ, if $j \in \mathit{region}(i, \tau)$ (or $i = j$) and j is a return point then $\mathsf{jun}(i, \tau)$ is undefined;
SOAP4: for all program points i and tags τ_1, τ_2, if $\mathsf{jun}(i, \tau_1)$ and $\mathsf{jun}(i, \tau_2)$ are defined and $\mathsf{jun}(i, \tau_1) \neq \mathsf{jun}(i, \tau_2)$ then $\mathsf{jun}(i, \tau_1) \in \mathsf{region}(i, \tau_2)$ or $\mathsf{jun}(i, \tau_2) \in \mathsf{region}(i, \tau_1)$;
SOAP5: for all program points i, j and tag τ, if $j \in \mathit{region}(i, \tau)$ (or $i = j$) and j is a return point then for all tag τ' such that $\mathsf{jun}(i, \tau')$ is defined, $\mathsf{jun}(i, \tau') \in \mathsf{region}(i, \tau)$.

[8] This hypothesis is directly put as precondition of the semantics rules, in the same way that only well-typed state are considered when assuming a program is byte-code verified. It is straightforward to show that our instrumented semantics coincides with the standard semantics if the exception analysis is safe.

$$\frac{\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_m \langle i', \rho', os', h'\rangle \text{ in } \mathrm{JVM}_{\mathcal{C}} \text{ semantics}}{\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\emptyset} \langle i', \rho', os', h'\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad m' = \mathsf{lookup}_P(m_{\mathrm{ID}}, \mathsf{class}(h(l))) \\ l \in dom(h) \qquad \mathsf{length}(os_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ \langle 1, \{this \mapsto l, \boldsymbol{x} \mapsto os_1\}, \epsilon, h\rangle \Downarrow_{m'}^{(n)} \langle l'\rangle, h' \\ e = \mathsf{class}(h'(l')) \qquad \mathsf{Handler}_m(i, e) = t \qquad e \in \mathsf{excAnalysis}(m_{\mathrm{ID}})\end{array}}{\langle i, \rho, os_1 :: l :: os_2, h\rangle \overset{(n+1)}{\rightsquigarrow}_{m,e} \langle t, \rho, l' :: \epsilon, h'\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad m' = \mathsf{lookup}_P(m_{\mathrm{ID}}, \mathsf{class}(h(l))) \\ l \in dom(h) \qquad \mathsf{length}(os_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ \langle 1, \{this \mapsto l, \boldsymbol{x} \mapsto os_1\}, \epsilon, h\rangle \Downarrow_{m'}^{(n)} \langle l'\rangle, h' \\ e = \mathsf{class}(h'(l')) \qquad \mathsf{Handler}_m(i, e) \uparrow \qquad e \in \mathsf{excAnalysis}(m_{\mathrm{ID}})\end{array}}{\langle i, \rho, os_1 :: l :: os_2, h\rangle \overset{(n+1)}{\rightsquigarrow}_{m,e} \langle l'\rangle, h'}$$

$$\frac{P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad l' = \mathrm{fresh}(h) \qquad \mathsf{length}(os_1) = \mathsf{nbArguments}(m_{\mathrm{ID}})}{\langle i, \rho, os_1 :: null :: os_2, h\rangle \overset{(0)}{\rightsquigarrow}_{m,\mathbf{np}} \mathsf{RuntimeExceptionHandling}(h, l', \mathbf{np}, i, \rho)}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad l' = \mathrm{fresh}(h)}{\langle i, \rho, null :: os, h\rangle \overset{(0)}{\rightsquigarrow}_{m,\mathbf{np}} \mathsf{RuntimeExceptionHandling}(h, l', \mathbf{np}, i, \rho)}$$

$$\frac{P_m[i] = \mathsf{putfield}\ f \qquad l' = \mathrm{fresh}(h)}{\langle i, \rho, v :: null :: s, h\rangle \overset{(0)}{\rightsquigarrow}_{m,\mathbf{np}} \mathsf{RuntimeExceptionHandling}(h, l', \mathbf{np}, i, \rho)}$$

$$\frac{P_m[i] = \mathsf{throw} \qquad l' = \mathrm{fresh}(h)}{\langle i, \rho, null :: s, h\rangle \overset{(0)}{\rightsquigarrow}_{m,\mathbf{np}} \mathsf{RuntimeExceptionHandling}(h, l', \mathbf{np}, i, \rho)}$$

$$\frac{\begin{array}{c}P_m[i] = \mathsf{throw} \qquad l \in \mathsf{dom}(h) \qquad e = \mathsf{class}(h(l)) \\ \mathsf{Handler}_m(i, e) = t \qquad e \in \mathsf{classAnalysis}(m, i)\end{array}}{\langle i, \rho, l :: os, h\rangle \overset{(0)}{\rightsquigarrow}_{m,e} \langle t, \rho, l :: \epsilon, h\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \mathsf{throw} \qquad l \in \mathsf{dom}(h) \qquad e = \mathsf{class}(h(l)) \\ \mathsf{Handler}_m(i, e) \uparrow \qquad e \in \mathsf{classAnalysis}(m, i)\end{array}}{\langle i, \rho, l :: os, h\rangle \overset{(0)}{\rightsquigarrow}_{m,e} \langle l\rangle, h}$$

with $\mathsf{RuntimeExceptionHandling} : \mathsf{Heap} \times \mathcal{L} \times \mathcal{C} \times \mathcal{PP} \times (\mathcal{X} \rightharpoonup \mathcal{V}) \to \mathrm{State} + (\mathcal{L} \times \mathsf{Heap})$ defined by

$$\mathsf{RuntimeExceptionHandling}(h, l', C, i, \rho) = \begin{cases} \langle t, \rho, l' :: \epsilon, h \oplus \{l' \mapsto \mathsf{default}(C)\}\rangle & \text{if } \mathsf{Handler}_m(i, C) = t \\ \langle l'\rangle, h \oplus \{l' \mapsto \mathsf{default}(C)\} & \text{if } \mathsf{Handler}_m(i, C) \uparrow \end{cases}$$

Fig. 10. New operational semantics rules for $\mathrm{JVM}_{\mathcal{G}}$

$$\frac{i \mapsto_{\mathrm{JVM}_{\mathcal{C}}} j}{i \mapsto^{\emptyset} j} \qquad \frac{i \mapsto_{\mathrm{JVM}_{\mathcal{C}}}}{i \mapsto^{\emptyset}}$$

$$\frac{P_m[i] \in \{\mathsf{getfield}\ f, \mathsf{putfield}\ f, \mathsf{throw}, \mathsf{invokevirtual}\ m_{\mathrm{ID}}\} \qquad \mathsf{Handler}(i, \mathbf{np}) = t}{i \mapsto^{\mathbf{np}} t}$$

$$\frac{P_m[i] \in \{\mathsf{getfield}\ f, \mathsf{putfield}\ f, \mathsf{throw}, \mathsf{invokevirtual}\ m_{\mathrm{ID}}\} \qquad \mathsf{Handler}(i, \mathbf{np}) \uparrow}{i \mapsto^{\mathbf{np}}}$$

$$\frac{P_m[i] = \mathsf{throw} \qquad C \in \mathsf{classAnalysis}(m, i) \qquad \mathsf{Handler}(i, C) = t}{i \mapsto^{C} t}$$

$$\frac{P_m[i] = \mathsf{throw} \qquad C \in \mathsf{classAnalysis}(m, i) \qquad \mathsf{Handler}(i, C) \uparrow}{i \mapsto^{C}}$$

$$\frac{P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad C \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \qquad \mathsf{Handler}(i, C) = t}{i \mapsto^{C} t}$$

$$\frac{P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad C \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \qquad \mathsf{Handler}(i, C) \uparrow}{i \mapsto^{C}}$$

Fig. 11. Successor relation for $\mathrm{JVM}_{\mathcal{G}}$

Junction points uniquely delimits ends of regions. SOAP1 expresses that successors of branching points belongs (or ends) the region associated with the same kind as their successor relation. SOAP2 says that a successor of a point in a region is either still in the same region or at this end. SOAP3 forbids junction points for a region which contains (or start with) a return point. SOAP4 and SOAP5 express properties between regions of a same program point but with different tags. SOAP4 says that if two differently tagged regions end in distinct points, the junction point of one must belong to the region of the other. SOAP5 imposes that the junction point of a region must be within every region which contains (or starts with) a return point and is decorated with a different tag.

Figure 12 presents an example of safe cdr for an abstract transition system.

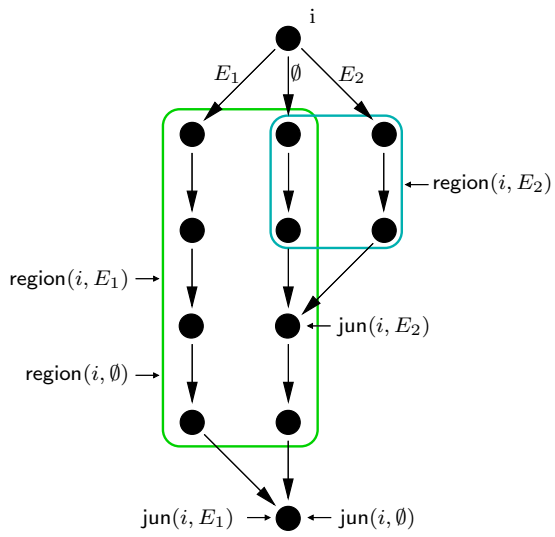

Fig. 12. Example of cdr in $\mathrm{JVM}_{\mathcal{G}}$. Only relevant tags are presented here.

6.2 Non-Interference

Method signatures are now of the form

$$\boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}$$

where $\boldsymbol{k_v}$, k_h are defined as in $\mathsf{JVM}_\mathcal{C}$ but $\boldsymbol{k_r}$ (called *output level*) is now a list of security level of the form $\{n : k_n, e_1 : k_{e_1}, \ldots e_n : k_{e_n}\}$, where k_n is the security level of the return value and e_i is an exception class that might be propagated by the method in a security environment (or due to an exception-throwing instruction) of level k_i. In the rest of the paper we will write $\boldsymbol{k_r}[n]$ instead of k_n and $\boldsymbol{k_r}[e_i]$ instead of k_{e_i}.

This new notion of output level is associated with a new notion of *output indistinguishability*.

Definition 23 (Output indistinguishability). *Given a partial function $\beta \in \mathcal{LL} \rightharpoonup \mathcal{LL}$, a output level $\boldsymbol{k_r}$, indistinguishability of two final states in method m is defined by the clauses:*

$$\frac{h_1 \sim_\beta h_2 \qquad \boldsymbol{k_r}[n] \leq k_{\text{obs}} \Rightarrow v_1 \sim_\beta v_2}{(v_1, h_1) \sim_{\beta,\boldsymbol{k_r}} (v_2, h_2)}$$

$$\frac{h_1 \sim_\beta h_2 \qquad \mathsf{class}(h_1(l_1)) : k \in \boldsymbol{k_r} \qquad k \leq k_{\text{obs}} \qquad l_1 \sim_\beta l_2}{(\langle l_1 \rangle, h_1) \sim_{\beta,\boldsymbol{k_r}} (\langle l_2 \rangle, h_2)}$$

$$\frac{h_1 \sim_\beta h_2 \qquad \mathsf{class}((h_1(l_1)) : k \in \boldsymbol{k_r} \qquad k \not\leq k_{\text{obs}}}{(\langle l_1 \rangle, h_1) \sim_{\beta,\boldsymbol{k_r}} (v_2, h_2)}$$

$$\frac{h_1 \sim_\beta h_2 \qquad \mathsf{class}(h_2(l_2)) : k \in \boldsymbol{k_r} \qquad k \not\leq k_{\text{obs}}}{(v_1, h_1) \sim_{\beta,\boldsymbol{k_r}} (\langle l_2 \rangle, h_2)}$$

$$\frac{h_1 \sim_\beta h_2 \qquad \mathsf{class}(h_1(l_1)) : k_1 \in \boldsymbol{k_r} \qquad \mathsf{class}(h_2(l_2)) : k_2 \in \boldsymbol{k_r} \qquad k_1 \not\leq k_{\text{obs}} \qquad k_2 \not\leq k_{\text{obs}}}{(\langle l_1 \rangle, h_1) \sim_{\beta,\boldsymbol{k_r}} (\langle l_2 \rangle, h_2)}$$

In each cases, heaps must be indistinguishable. This definition implies that if indistinguishability outputs are of different nature (like normal value/exception or two exceptions from different classes) the security level of the corresponding exception must be high in the output signature $\boldsymbol{k_r}$. When outputs are of similar nature (two normal values or two exceptions of the same class) they are indistinguishable as soon as the corresponding security level in $\boldsymbol{k_r}$ is low.

The previous definition and the next definition of non-interference rely on indistinguishability definitions already proposed for the $\text{JVM}_\mathcal{O}$ (*c.f.* page 16).

Definition 24 (Non-interferent $\text{JVM}_\mathcal{G}$ method). *A method m is* non-interferent *w.r.t. a policy $\boldsymbol{k_v} \longrightarrow \boldsymbol{k_r}$, if for every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every $\rho_1, \rho_2 \in \mathcal{X} \rightharpoonup \mathcal{V}$, $h_1, h_2, h_1', h_2' \in \mathsf{Heap}$, $r_1, r_2 \in \mathcal{V} + \mathcal{L}$ such that $\rho_1, h_1 \Downarrow_m r_1, h_1'$, $\rho_2, h_2 \Downarrow_m r_2, h_2'$ and $h_1 \sim_\beta h_2$, $\rho_1 \sim_{\boldsymbol{k_v},\beta} \rho_2$, there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$ and $(r_1, h_1) \sim_{\beta',\boldsymbol{k_r}} (r_2, h_2)$.*

Like in $\text{JVM}_\mathcal{C}$, we impose a *side-effect-safe* (*c.f.* page 20 for a formal definition) on methods. This notion is used when virtual call occur in a high context in order to enforce that no modification is done on low information during the execution of the called method.

Definition 25 (Safe $\text{JVM}_\mathcal{G}$ method). *A method m is* safe *w.r.t. a policy $\boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}$ m is side-effect-safe with respect to k_h and m is non-interferent with respect to $\boldsymbol{k_v} \longrightarrow \boldsymbol{k_r}$.*

Definition 26 (Safe $\text{JVM}_\mathcal{G}$ program). *A program is* safe *with respect to a table of method signature Γ if for all its method m, m is safe with respect to all policies in* $\text{Policies}_\Gamma(\text{m})$[9].

[9] $\text{Policies}_\Gamma(\text{m})$ has been defined in page 21.

6.3 Typing rules

Typing rules for the previous JVM$_\mathcal{C}$ are extended (or sometimes modified) with rules given in figure 13. These rules concern only exception-throwing and branching instruction. The other instructions keep the same rules as in JVM$_\mathcal{C}$.

Observe that the typing judgement is now parameterized by a tag $\tau \in \{\emptyset\} + \mathcal{C}$. It will be used to describe without ambiguity which typing constraint must be verified according to kind of execution performed in the semantics.

This notion of tag requires to update the notion of *typable method*.

Definition 27 (Typable method). *A method m is typable w.r.t. a method signature table Γ, a global field policy* ft*, a signature sgn and a cdr* $\mathsf{region}_m : \mathcal{PP} \to \wp(\mathcal{PP})$ *if there exists a security environment $se : \mathcal{PP} \to \mathcal{S}$ and a function $S : \mathcal{PP} \to \mathcal{S}^\star$ such that $S_1 = \varepsilon$ and for all $i, j \in \mathcal{PP}$, $e \in \{\emptyset\} + \mathcal{C}$:*

1. *$i \mapsto^e j$ implies there exists $st \in \mathcal{S}^\star$ such that $\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash^e S_i \Rightarrow st$ and $st \sqsubseteq S_j$;*
2. *$i \mapsto^e$ implies $\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash^e S_i \Rightarrow$*

Typable example The following method mays throw two kind of exception: an exception of class `C` if the parameter `x` is true and of class **np** in the other case. The first exception depends on `x` while the second depends both on `x` and `y`. Normal return depends on `y` because execution terminates normally only if it is not *null*.

```
int m(boolean x,C y) throws C {
  if (x) {throw new C();}
  else {y.f = 3;};
  return 1;
}
```

At the bytecode level we obtain the following method:

0 : load x
1 : ifeq 4
2 : new C
3 : throw
4 : load y
5 : push 3
6 : putfield f
7 : const 1
8 : return

Such a method is typable with the signature

$$m : (\mathit{this} : L,\ x : L,\ y : H) \xrightarrow{H} \{n : H,\ C : L,\ \mathbf{np} : H\}$$

thanks to the cdr[10], the type stacks and the security environment given below:

i	region$(i, \cdot)$	jun$(i, \cdot)$	S_i	$se(i)$
0	$\emptyset$	1	ε	L
1	$\{2,3,4,5,6,7,8\}$	undef	$L :: \varepsilon$	L
2	$\emptyset$	3	ε	L
3	$\emptyset$	undef	$L :: \varepsilon$	L
4	$\emptyset$	5	ε	L
5	$\emptyset$	6	$H :: \varepsilon$	L
6	$\{7,8\}$	undef	$L :: H :: \varepsilon$	L
7	$\emptyset$	8	ε	H
8	$\emptyset$	undef	$H :: \varepsilon$	H

[10] In this example, it is safe to take the same cdr for all tags, so we do not distinguish them here.

$$\frac{P_m[i] = \mathsf{ifeq}\ j \qquad \forall j' \in \mathsf{region}(i, \emptyset),\ k \leq se(j')}{\Gamma, \mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\emptyset} k :: st \Rightarrow \mathrm{lift}_k(st)}$$

$$\frac{P_m[i] = \mathsf{return} \qquad k \sqcup se(i) \leq \boldsymbol{k_r}[n]}{\Gamma, \mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\emptyset} k :: st \Rightarrow}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \boldsymbol{k'_a} \xrightarrow{k'_h} \boldsymbol{k'_r} \\ k \sqcup k_h \sqcup se(i) \leq k'_h \qquad \mathsf{length}(st_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \boldsymbol{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \boldsymbol{k'_a}[i+1] \\ k_e = \bigsqcup \{\ \boldsymbol{k'_r}[e] \mid e \in \mathsf{excAnalysis}(m_{\mathrm{ID}})\ \} \\ \forall j \in \mathsf{region}(i, \emptyset),\ k \sqcup k_e \leq se(j) \end{array}}{\Gamma, \mathsf{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\emptyset} st_1 :: k :: st_2 \Rightarrow \mathrm{lift}_{k \sqcup k_e}\ \big((\boldsymbol{k'_r}[n] \sqcup se(i)) :: st_2\big)}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \boldsymbol{k'_a} \xrightarrow{k'_h} \boldsymbol{k'_r} \\ k \sqcup k_h \sqcup se(i) \leq k'_h \qquad \mathsf{length}(st_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \boldsymbol{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \boldsymbol{k'_a}[i+1] \\ e \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \qquad \forall j \in \mathsf{region}(i, e),\ k \sqcup \boldsymbol{k'_r}[e] \leq se(j) \qquad \mathsf{Handler}(i, e) = t \end{array}}{\Gamma, \mathsf{region}, se, \boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow (k \sqcup \boldsymbol{k'_r}[e]) :: \varepsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \boldsymbol{k'_v} \xrightarrow{k'_h} \boldsymbol{k'_r} \\ k \sqcup k_h \sqcup se(i) \leq k'_h \qquad \mathsf{length}(st_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \boldsymbol{k'_v}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \boldsymbol{k'_v}[i+1] \\ e \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \qquad k \sqcup se(i) \sqcup \boldsymbol{k'_r}[e] \leq \boldsymbol{k_r}[e] \qquad \forall j \in \mathsf{region}(i, e),\ k \sqcup \boldsymbol{k'_r}[e] \leq se(j) \qquad \mathsf{Handler}(i, e) \uparrow \end{array}}{\Gamma, \mathsf{region}, se, \boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow}$$

$$\frac{\begin{array}{c} P[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f) \qquad k_h \leq \mathsf{ft}(f) \\ \forall j \in \mathsf{region}(i, \emptyset),\ k_2 \leq se(j) \end{array}}{\Gamma, \mathsf{region}, se, \boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\emptyset} k_1 :: k_2 :: st \Rightarrow \mathrm{lift}_{k_2} st}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f) \\ \forall j \in \mathsf{region}(i, \mathbf{np}),\ k_2 \leq se(j) \qquad \mathsf{Handler}(i, \mathbf{np}) = t \end{array}}{\Gamma, \mathsf{region}, se, \boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\mathbf{np}} k_1 :: k_2 :: st \Rightarrow k_2 \sqcup se(i) :: \epsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f) \\ k_2 \leq \boldsymbol{k_r}[\mathbf{np}] \qquad \forall j \in \mathsf{region}(i, \mathbf{np}),\ k_2 \leq se(j) \qquad \mathsf{Handler}(i, \mathbf{np}) \uparrow \end{array}}{\Gamma, \mathsf{region}, se, \boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\mathbf{np}} k_1 :: k_2 :: st \Rightarrow}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \forall j \in \mathsf{region}(i, \emptyset),\ k \leq se(j)}{\Gamma, \mathsf{region}, se, \boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\emptyset} k :: st \Rightarrow \mathrm{lift}_k((\mathsf{ft}(f) \sqcup se(i)) :: st)}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \forall j \in \mathsf{region}(i, \mathbf{np}),\ k \leq se(j) \qquad \mathsf{Handler}(i, \mathbf{np}) = t}{\Gamma, \mathsf{region}, se, \boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\mathbf{np}} (k :: st \Rightarrow k \sqcup se(i)) :: \epsilon}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \forall j \in \mathsf{region}(i, \mathbf{np}),\ k \leq se(j) \qquad \mathsf{Handler}(i, \mathbf{np}) \uparrow \qquad k \leq \boldsymbol{k_r}[\mathbf{np}]}{\Gamma, \mathsf{region}, se, \boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\mathbf{np}} k :: st \Rightarrow}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{throw} \qquad e \in \mathsf{classAnalysis}(i) \cup \{\mathbf{np}\} \\ \forall j \in \mathsf{region}(i, e),\ k \leq se(j) \qquad \mathsf{Handler}(i, e) = t \end{array}}{\Gamma, \mathsf{region}, se, \boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{e} k :: st \Rightarrow k \sqcup se(i) :: \epsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{throw} \qquad e \in \mathsf{classAnalysis}(i) \cup \{\mathbf{np}\} \\ k \leq \boldsymbol{k_r}[e] \qquad \forall j \in \mathsf{region}(i, e),\ k \leq se(j) \qquad \mathsf{Handler}(i, e) \uparrow \end{array}}{\Gamma, \mathsf{region}, se, \boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{e} k :: st \Rightarrow}$$

Fig. 13. Transfer rules for instructions of $\mathrm{JVM}_{\mathcal{G}}$

The next method gives an example of code with method invocation where fine grain exception handling is necessary. To keep a short example here we give compressed version of a compiled code.

foo :
0 load o_L
1 load x_L
2 load y_H
3 invokevirtual m
4 store z_H
5 load o_L
6 push 1
7 putfield f_L
handler : $[0, 3], \mathsf{NullPointer} \rightarrow 4$

i	$\mathsf{region}(i, \emptyset)$	$\mathsf{jun}(i, \emptyset)$	$\mathsf{region}(i, NP)$	$\mathsf{jun}(i, NP)$	$\mathsf{region}(i, C)$	$\mathsf{jun}(i, C)$	S_i	$se(i)$
0	$\emptyset$	1	$\emptyset$	1	$\emptyset$	1	ε	L
1	$\emptyset$	2	$\emptyset$	2	$\emptyset$	2	$L :: \varepsilon$	L
2	$\emptyset$	3	$\emptyset$	3	$\emptyset$	3	$L :: L :: \varepsilon$	L
3	$\emptyset$	4	$\emptyset$	4	$\{4, 5, 6, 7, \ldots\}$	$\ldots$	$H :: L :: L :: \varepsilon$	L
4	$\emptyset$	5	$\emptyset$	5	$\emptyset$	5	$H :: \varepsilon$	L
5	$\emptyset$	6	$\emptyset$	6	$\emptyset$	6	ε	L
6	$\emptyset$	7	$\emptyset$	7	$\emptyset$	7	$H :: \varepsilon$	L
7	$\{\ldots\}$	$\ldots$	$\{\ldots\}$	$\ldots$	$\{\ldots\}$	$\ldots$	$L :: L :: \varepsilon$	L

Update $o_L.f_L = 1$ at point 7 is accepted if and only if $se(6)$ and $se(7)$ are low. Thanks to the fine grain regions, typing rule for virtual call only propagate exception levels of m in distinct regions:

$$\forall j \in \mathsf{region}(i, \mathbf{np}) = \emptyset,\ \boldsymbol{k_r}[\mathbf{np}] = H \leq se(j)$$
$$\forall j \in \mathsf{region}(i, C) = \{4, 5, 6, 7, \ldots\},\ \boldsymbol{k_r}[C] = L \leq se(j)$$

It follows that $se(6)$ and $se(7)$ are low and the update is accepted by our type system.

6.4 Type system soundness

Theorem 4. *Let P be a* $\mathrm{JVM}_{\mathcal{G}}$ *program, Γ a table of signatures,* ft *a global field policy, and for all method m in P,* $(\mathsf{region}_m, \mathsf{jun}_m)$ *a safe cdr for m (according to SOAP properties). Suppose all methods m in P are typable with respect to Γ,* ft, region_m *and to all signatures in* $\mathrm{Policies}_\Gamma(\mathrm{m})$. *Then P is safe with respect to Γ.*

Side-effect safety The first part of the soundness proof consist in proving that all methods of a typable program are side-effect-safe.

In this paragraph m is supposed to be a method of P, region a cdr function for m, se a security environment and $\boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}$ a security signature.

We show that all instruction step transform a heap h into a heap h' such that $h \preceq_{k_h} h'$. For the special case of virtual call we need an inductive hypothesis about the side-effect safety of all methods in P. We hence introduce the notion of *Side-effect-safe at order n*.

Definition 28 (Side-effect-safe at order n). *A method m is* side-effect-safe at order n *with respect to a security level k_h if for all state $\langle i, \rho, os, h\rangle \in \mathrm{State}_{\mathrm{JVM}_{\mathcal{G}}}$, all heap $h' \in \mathsf{Heap}$ and result $t \in \mathcal{V} + \mathcal{L}$, $\langle i, \rho, os, h\rangle \Downarrow_m^{(n)} r, h'$ implies $h \preceq_{k_h} h'$.*

This first lemma focuses on instruction which do not end the execution of the m.

Lemma 19. *Let n an integer and suppose all method m' in P are side-effect-safe at all order p, $p < n$ with respect to the heap effect level of all the policies in* $\text{Policies}_\Gamma(\text{m}')$.

Let $\langle i, \rho, os, h\rangle, \langle i, \rho', os', h'\rangle \in \text{State}_{\text{JVM}_\mathcal{G}}$ two states and $p \in \mathbb{N}$, $p \leq n$ such that

$$\langle i, \rho, os, h\rangle \overset{(p)}{\rightsquigarrow}_{m,\tau} \langle i', \rho', os', h'\rangle$$

Let two stack types $st, st' \in \mathcal{S}^\star$ such that

$$i \vdash^\tau st \Rightarrow st'$$

then $h \preceq_{k_h} h'$.

Proof. By a case analysis on the instruction $P_m[i]$. According to the form $\langle i, \rho, os, h\rangle \rightsquigarrow_m \langle i', \rho', os', h'\rangle$ of the semantic step, only four cases appear:

Case 1: $P_m[i]$ does not modify the heap. We can simply conclude by reflexivity of $\preceq_{k_h}$.

Case 2: $P_m[i] = new\ C$ or an instruction throwing a null pointer exception. h' is of the form $h \oplus \{\text{fresh}(h) \mapsto \mathsf{default}(C)\}$ and we can conclude with lemma 36.

Case 3: $P_m[i] = \mathsf{putfield}\ f$. h and h' only differ in field f and the corresponding typing rule implies $k_h \leq \mathsf{ft}(f)$. Hence h and h' correspond for all field f' such that $k_h \not\leq \mathsf{ft}(f')$: $h \preceq_{k_h} h'$ holds.

Case 4: $P_m[i] = \mathsf{invokevirtual}\ m_{\text{ID}}$ and the called method has terminated normally or with an exception which is caught in m. p is necessarily of the form $p' + 1$ and we have an hypothesis like

$$\langle 1, \{this \mapsto l, \boldsymbol{x} \mapsto os_1\}, \epsilon, h\rangle \Downarrow_{m'}^{(p')} (r', h')$$

But $p' < n$ so m' is side-effect safe at order p' and we can hence conclude that $h \preceq_{k_h} h'$.

The next lemma treats the case of return instruction.

Lemma 20. *Let n an integer and suppose all method m' in P are side-effect-safe at all order p, $p < n$ with respect to the heap effect level of all the policies in* $\text{Policies}_\Gamma(\text{m}')$.

Let a method m and $\langle i, \rho, os, h\rangle \in \text{State}_{\text{JVM}_\mathcal{G}}$ a state, $h' \in \mathsf{Heap}$ a heap, $r \in \mathcal{V} + \mathcal{L}$ a result and $p \in \mathbb{N}$, $p \leq n$ such that

$$\langle i, \rho, os, h\rangle \overset{(p)}{\rightsquigarrow}_{m,\tau} r, h'$$

Let a stack type $st \in \mathcal{S}^\star$ such that

$$i \vdash^\tau st \Rightarrow$$

then $h \preceq_{k_h} h'$.

Proof. By a case analysis on the instruction in $P_m[i]$. The form of the semantic step constrains to be an instruction ending the execution of m. We hence are in one the following cases:

Case 1 $P_m[i] = \mathsf{return}$. h is hence equal to h' and we conclude by reflexivity of $\preceq_{k_h}$.

Case 2 if $P_m[i]$ is an instruction throwing a null pointer exception, h' is of the form $h \oplus \{\text{fresh}(h) \mapsto \mathsf{default}(C)\}$ and we can conclude with lemma 36.

Case 3: $P_m[i] = \mathsf{invokevirtual}\ m_{\text{ID}}$ and the called method has terminated abnormally with an exception uncaught in m. p is necessarily of the form $p' + 1$ and we have an hypothesis like

$$\langle 1, \{this \mapsto l, \boldsymbol{x} \mapsto os_1\}, \epsilon, h\rangle \Downarrow_{m'}^{(p')} \langle l'\rangle, h'$$

But $p' < n$ so m' is side-effect safe at order p' and we can hence conclude that $h \preceq_{k_h} h'$.

We then conclude about the side-effect safety of all methods in P, using an induction on the number of virtual call on semantics derivation and an induction on the derivation length.

Proposition 1 (side-effect safety). *Let for all method m in P, $(\mathsf{region}_m, \mathsf{jun}_m)$ a safe cdr for m. Suppose all methods m in P are typable with respect to region_m and to all signatures in $\mathrm{Policies}_\Gamma(\mathrm{m})$. Then all method m are side-effect-safe with respect to the heap effect level of all the policies in $\mathrm{Policies}_\Gamma(\mathrm{m})$.*

Proof. We show for all $n \in \mathbb{N}$ that all method m in P are side-effect-safe at order n with respect to the heap effect level of all the policies in $\mathrm{Policies}_\Gamma(\mathrm{m})$. We use a strong induction on n. So we suppose all method m in P are side-effect-safe at order k, for any $k < n$. We take a method m and prove now is it side-effect-safe at order n with respect to any heap effect level k_h of all the policies in $\mathrm{Policies}_\Gamma(\mathrm{m})$.

Given a state $\langle i_0, \rho_0, os_0, h_0\rangle$ and a final state (r, h') such that $\langle i_0, \rho_0, os_0, h_0\rangle \Downarrow^{(n)} (r, h')$ we have to prove that $h_0 \preceq_{k_h} h'$.

There is necessarily a derivation

$$\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots \langle i_k, \rho_k, os_k, h_k\rangle \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h)$$

with $n_0 + \cdots + n_k \leq n$.

P is typable so there exists $S \in \mathcal{PP} \to \mathcal{S}^\star$ and $st_1, \ldots, st_{k-1} \in \mathcal{S}^\star$ such that

$$i_0 \vdash^{\tau_0} S(i_0) \Rightarrow st_1 \quad i_1 \vdash^{\tau_1} S(i_1) \Rightarrow st_2 \quad \cdots \quad i_k \vdash^{\tau_k} S(i_k) \Rightarrow$$

For all $i \in \{0, \ldots, k\}$, $n_i \leq n$ so thanks to lemmas 19 and 20 we have

$$h_0 \preceq_{k_h} h_1 \preceq_{k_h} \cdots \preceq_{k_h} h_k \preceq_{k_h} h$$

We conclude by transitivity of $\preceq_{k_h}$.

Non-interference proof When a method execution ends in a high context (*i.e.* in a region where the security environment is high), the type system enforces the output level to be high according to the kind of termination of the method (normal or with an uncaught exception). This notion of *high result* is captured by the following formal definition.

Definition 29 (High result). *Given $(r, h) \in (\mathcal{V} + \mathcal{L}) \times \mathsf{Heap}$ and an output level $\boldsymbol{k_r}$, the predicate $\mathsf{highResult}_{\boldsymbol{k_r}}(r, h)$ is inductively defined by:*

$$\frac{\boldsymbol{k_r}[n] \not\leq k_{\mathrm{obs}} \qquad v \in \mathcal{V}}{\mathsf{highResult}_{\boldsymbol{k_r}}(v, h)} \qquad \frac{\boldsymbol{k_r}[\mathsf{class}(h(l))] \not\leq k_{\mathrm{obs}} \qquad l \in \mathsf{dom}(h)}{\mathsf{highResult}_{\boldsymbol{k_r}}(\langle l\rangle, h)}$$

We then prove non-interference using the four lemmas sketched in section 2. From now, we assume P is a program, Γ a table of signature and ft a global field policy such that all its method m are side-effect-safe with respect to the heap effect level of all the policies in $\mathrm{Policies}_\Gamma(\mathrm{m})$.

In this paragraph m is suppose to be a method of P, $(\mathsf{region}, \mathsf{jun})$ a cdr function for m, *se* a security environment, $S \in \mathcal{PP} \to \mathcal{S}^\star$ a stack type annotation and $\boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}$ a security signature.

Lemma 21 ($\mathrm{JVM}_\mathcal{G}$ high step). *Let $\langle i, \rho, os, h\rangle, \langle i', \rho', os', h'\rangle \in \mathrm{State}_\mathcal{G}$ two $\mathrm{JVM}_\mathcal{G}$ states and two stack types $st, st' \in \mathcal{S}^\star$ such that*

$$\frac{\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} \langle i, \rho', os', h'\rangle \qquad i \vdash st \Rightarrow^\tau st' \qquad se(i) \not\leq k_{\mathrm{obs}} \qquad \mathsf{high}(os, st)}{}$$

then

$$\mathsf{high}(os', st')$$

Proof. By case analysis on $P_m[i]$. Only non final steps are considered here. For each case we first resume the hypotheses and state the result to prove.

$P_m[i]$	Hypotheses	Goal
push n	$\mathsf{high}(os, st)$	$\mathsf{high}(n :: os, se(i) :: st)$
binop op	$\mathsf{high}(n_1 :: n_2 :: os, k_1 :: k_2 :: st)$	$\mathsf{high}(n_1 \underline{op}\ n_2 :: os, se(i) :: (k_1 \sqcup k_2 \sqcup se(i)) :: st)$
pop	$\mathsf{high}(v :: os, k :: st)$	$\mathsf{high}(os, st)$
store x	$\mathsf{high}(v :: os, k :: st)$	$\mathsf{high}(os, st)$
load x	$\mathsf{high}(os, st)$	$\mathsf{high}(\rho(x) :: os, (\boldsymbol{k_v}(x) \sqcup se(i)) :: st)$
goto j	$\mathsf{high}(os, st)$	$\mathsf{high}(os, st)$
ifeq j	$\mathsf{high}(n :: os, k :: st)$	$\mathsf{high}(os, \mathrm{lift}_k(st))$
new C	$\mathsf{high}(os, st)$	$\mathsf{high}(l :: os, se(i) :: st)$
getfield f	$\mathsf{high}(l :: os, k :: st)$	$\mathsf{high}(h(l).f :: os, \mathrm{lift}_k((\mathsf{ft}(f) \sqcup se(i)) :: st))$
	$\mathsf{high}(null :: os, k :: st)$	$\mathsf{high}(l' :: \epsilon, k \sqcup se(i) :: \epsilon)$
putfield f	$\mathsf{high}(v :: l :: os, k_1 :: k_2 :: st)$	$\mathsf{high}(os, \mathsf{lift}_{k_2} st)$
	$\mathsf{high}(v :: null :: os, k_1 :: k_2 :: st)$	$\mathsf{high}(l' :: \epsilon, k_2 \sqcup se(i) :: \epsilon)$
invokevirtual m_{ID}	$\mathsf{high}(os_1 :: l :: os_2, st_1 :: k :: st_2)$	$\mathsf{high}(v :: os_2, \mathrm{lift}_{k \sqcup k_e}\ ((\boldsymbol{k'_r}[n] \sqcup se(i)) :: st_2))$
	$\mathsf{high}(os_1 :: null :: os_2, st_1 :: k :: st_2)$	$\mathsf{high}(l' :: \epsilon, k \sqcup \boldsymbol{k'_r}[e] :: \epsilon)$
throw	$\mathsf{high}(l :: os, k :: st)$	$\mathsf{high}(l :: \epsilon, k \sqcup se(i) :: \epsilon)$
	$\mathsf{high}(null :: os, k :: st)$	$\mathsf{high}(l' :: \epsilon, k \sqcup se(i) :: \epsilon)$

Each case is easily proved using the fact that $se(i) \not\leq k_{\mathrm{obs}}$ and the following generic property:

$$\forall k \in \mathcal{L}, os \in \mathcal{V}^\star, st \in \mathcal{S}^\star,\ \mathsf{high}(os, st) \text{ implies } \mathsf{high}(os, \mathrm{lift}_k(st))$$

Lemma 22 (JVM$_\mathcal{G}$ step consistent). *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $\langle i, \rho, os, h\rangle, \langle i_0, \rho_0, os_0, h_0\rangle \in \mathrm{State}_\mathcal{G}$ two JVM$_\mathcal{G}$ states and two stack types $st, st_0 \in \mathcal{S}^\star$ such that*

$$\langle i, \rho, os, h\rangle \sim_{st, st_0, \beta} \langle i_0, \rho_0, os_0, h_0\rangle \qquad se(i) \not\leq k_{\mathrm{obs}} \qquad \mathsf{high}(os, st)$$

1. *If there exists a state $\langle i', \rho', os', h'\rangle \in \mathrm{State}_\mathcal{G}$, a tag $\tau \in \{\emptyset\} + \mathcal{C}$ and a stack type $st' \in \mathcal{S}^\star$ such that*

$$\langle i, \rho, os, h\rangle \overset{(n)}{\hookrightarrow}_{m,\tau} \langle i', \rho', os', h'\rangle \qquad i \vdash^\tau st \Rightarrow st'$$

   then

$$\langle i', \rho', os', h'\rangle \sim_{st', st_0, \beta} \langle i_0, \rho_0, os_0, h_0\rangle$$

2. *If there exists a result $r \in \mathcal{V} + \mathcal{L}$, a heap $h' \in \mathsf{Heap}$ and a tag $\tau \in \{\emptyset\} + \mathcal{C}$ such that*

$$\langle i, \rho, os, h\rangle \overset{(n)}{\hookrightarrow}_{m,\tau} r, h' \qquad i \vdash^\tau st \Rightarrow$$

   then

$$\mathsf{highResult}_{\boldsymbol{k_r}}(r, h') \quad \text{and} \quad h' \sim_\beta h_0$$

Proof.

1. Non-terminal step of the form $\langle i, \rho, os, h\rangle \overset{(n)}{\hookrightarrow}_{m,\tau} \langle i', \rho', os', h'\rangle$. Proofs can be reduce to examining local variables and heaps because, from $\mathsf{high}(os, st)$ and $os \sim_{st, st_0, \beta} os_0$ we first remark that $\mathsf{high}(os_0, st_0)$. Using the previous lemma (JVM$_\mathcal{G}$ high step), we know that $\mathsf{high}(os', st')$. As a consequence, we already know that $os' \sim_{st', st_0, \beta} os_0$ for any kind of instruction. We now prove the remaining $\rho' \sim_{\boldsymbol{k_v}, \beta} \rho_0$ and $h' \sim_\beta h_0$ properties.
   Local variables: Only the instruction store x modifies the local variable. We hence have to prove

$$\rho \oplus \{x \mapsto v\} \sim_{\boldsymbol{k_v}, \beta} \rho_0$$

We remark that $\boldsymbol{k_v}(x) \not\leq k_{\text{obs}}$ thanks to the typing rule which impose $k \leq \boldsymbol{k_v}(x)$ and the hypothesis $\mathsf{high}(v :: os, k :: st)$ which gives $k \not\leq k_{\text{obs}}$. Since local variables indistinguishability only depends on low variable and the modified variable x is high, we are done.
Heaps: Instruction which modify heaps are $\mathsf{new}\ C$, normal execution of $\mathsf{putfield}\ f$, $\mathsf{invokevirtual}\ m_{\text{ID}}$ and all instructions which throw a null pointer exception. We now examine this different cases and conclude using an appropriate technical lemma put in appendix :
- $\mathsf{new}\ C$. We conclude by lemma 37.
- $\mathsf{putfield}\ f$. We conclude by lemma 38.
- $\mathsf{invokevirtual}\ m_{\text{ID}}$. We conclude by lemma 39.
- Null pointer throwing. We conclude by lemma 37.

2. We make a case analysis on $P_m[i]$ for step of the form $\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} r, h'$.
   - return. The corresponding typing rule enforce $k \leq \boldsymbol{k_r}[n]$ with $\mathsf{high}(v :: os, k :: st)$ so $\boldsymbol{k_r}[n] \not\leq k_{\text{obs}}$ holds. The heap is not modified here so we are done.
   - $\mathsf{getfield}\ f$ (uncaught null pointer exception). By the typing rule, we have $k \leq \boldsymbol{k_r}[\mathbf{np}]$ and $\mathsf{high}(\mathit{null} :: os, k :: st)$ so $\boldsymbol{k_r}[\mathbf{np}] \not\leq k_{\text{obs}}$. The proof concerning heap is similar to the case where a null pointer exception is caught (see previous item).
   - $\mathsf{putfield}\ f$ (uncaught null pointer exception). By the typing rule, we have $k_2 \leq \boldsymbol{k_r}[\mathbf{np}]$ and $\mathsf{high}(v :: \mathit{null} :: os, k_1 :: k_2 :: st)$ so $\boldsymbol{k_r}[\mathbf{np}] \not\leq k_{\text{obs}}$. The proof concerning heap is similar to the case where a null pointer exception is caught (see previous item).
   - $\mathsf{invokevirtual}\ m_{\text{ID}}$ (uncaught exception e). By the typing rule, we have $k \leq \boldsymbol{k_r}[e]$ and $\mathsf{high}(os_1 :: v :: os_2, st_1 :: k :: st_2)$ so $\boldsymbol{k_r}[e] \not\leq k_{\text{obs}}$. The proof concerning heap is similar to the case where the exception is caught (see previous item).
   - throw. $k \leq \boldsymbol{k_r}[e]$ and $\mathsf{high}(l :: os, k :: st)$ (or $\mathsf{high}(\mathit{null} :: os, k :: st)$) lead to the expected condition $\boldsymbol{k_r}[e] \not\leq k_{\text{obs}}$. If $e \neq \mathbf{np}$, the heap is not modified. If $e = \mathbf{np}$ we conclude about the heap condition as when a null pointer exception is caught (see previous item).

These two lemmas about high steps are then used (with SOAP properties) to characterise execution of high fragment of code. The proof is more complex than the one sketched in section 2 because of the tag that now decorate regions.

We now give the definition of *typable execution* that will be used in the next proofs.

Definition 30 (typable execution).

- *An execution step* $\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} \langle i', \rho', os', h'\rangle$ *is typable with respect to* $S \in \mathcal{PP} \to \mathcal{S}^\star$ *if there exists* st' *such that,* $i \vdash^\tau S_i \Rightarrow st'$ *and* $st' \sqsubseteq S_{i'}$.
- *An execution step* $\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} (r, h')$ *is typable with respect to* $S \in \mathcal{PP} \to \mathcal{S}^\star$ *if* $i \vdash^\tau S_i \Rightarrow$.
- *An execution sequence* $s_0 \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_0} \dots s_k \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h')$ *is typable with respect to* $S \in \mathcal{PP} \to \mathcal{S}^\star$ *if*
  - *for all* i, $0 \leq i < k$, $s_i \overset{(n_i)}{\rightsquigarrow}_{m,\tau_i} s_{i+1}$ *is typable with respect to* S;
  - $s_n \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h')$ *is typable with respect to* S.

The first lemma prove that high executions end in a junction point or in a return point.

Lemma 23 (Iterated step consistent). *Let* β *a partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $\langle i_0, \rho_0, os_0, h_0\rangle, \langle i, \rho, os, h\rangle \in \text{State}_{\mathcal{G}}$ *two* $\text{JVM}_{\mathcal{G}}$ *states and a stack type* $st \in \mathcal{S}^\star$ *such that*

$$\langle i_0, \rho_0, os_0, h_0\rangle \sim_{S_{i_0}, st, \beta} \langle i, \rho, os, h\rangle \qquad \mathsf{high}(os_0, S_{i_0})$$

Let $i \in \mathcal{PP}$ *and* $\tau \in \{\emptyset\} + \mathcal{C}$ *such that*

$$i_0 \in \mathsf{region}(i, \tau) \qquad \textit{se is high in region } \mathsf{region}(i, \tau)$$

Suppose we have a derivation

$$\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots \langle i_k, \rho_k, os_k, h_k\rangle \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h')$$

and suppose this derivation is typable with respect to S. *Then one of the following case holds:*

1. $\mathsf{jun}(i,\tau)$ *is defined and there exists* j *with* $0 < j \leq k$ *such that*

$$i_j = \mathsf{jun}(i,\tau), \quad \langle i_j, \rho_j, os_j, h_j\rangle \sim_{S_{i_j},st,\beta} \langle i, \rho, os, h\rangle \quad \textit{and} \quad \mathsf{high}(os_j, S_{i_j})$$

2. $\mathsf{jun}(i,\tau)$ *is undefined,* $k \in \mathsf{region}(i,\tau)$, $\mathsf{highResult}_{\boldsymbol{k_r}}(r,h')$ *and* $h' \sim_\beta h$.

Proof. By induction on k.

- $k = 0$ we can directly apply case 2 of lemma 22 to conclude we are here in case 2. $\mathsf{jun}(i,\tau)$ is undefined because of SOAP3.
- we suppose the statement is true for a given k and we prove it now for $k+1$. First note that $se(i_0) \not\leq k_{\text{obs}}$, $\mathsf{high}(os_0, S_{i_0})$ and $i_0 \vdash S_{i_0} \Rightarrow st_1'$ hold for some st_1' such that $st_1' \sqsubseteq S_{i_1}$, so we have

$$\langle i_1, \rho_1, os_1, h_1\rangle \sim_{st_1',st,\beta} s \quad \text{and} \quad \mathsf{high}(os_1, st_1')$$

by case 1 of lemma 22 and by lemma 21.
Thanks to subtyping lemmas 7 and 5 (stated in page 14) we have:

$$\langle i_1, \rho_1, os_1, h_1\rangle \sim_{S_{i_1},st,\beta} s \quad \text{and} \quad \mathsf{high}(os_1, S_{i_1})$$

Then, using SOAP2, we have $i_1 \in \mathsf{region}(i,\tau)$ or $i_1 = \mathsf{jun}(i,\tau)$.
In the first case, it is sufficient to invoke induction hypothesis on derivation

$$\langle i_1, \rho_1, os_1, h_1\rangle \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} \cdots \langle i_{k+1}, \rho_{k+1}, os_{k+1}, h_{k+1}\rangle \overset{(n_{k+1})}{\rightsquigarrow}_{m,\tau_{k+1}} (r,h)$$

to conclude. Let's justify we can use it:
- $i_1 \in \mathsf{region}(i,\tau)$;
- the derivation is on length k and is typable;
- $\langle i_1, \rho_1, os_1, h_1\rangle \sim_{S_{i_1},st,\beta} s$ and $\mathsf{high}(os_1, S_{i_1})$ hold

In the second case we can conclude we are in case 1 by taking $j = 1$ and because we know $\langle i_1, \rho_1, os_1, h_1\rangle \sim_{S_{i_1},st,\beta} s$ and $\mathsf{high}(os_1, S_{i_1})$ hold.

In order to describe high executions starting from high branching we first prove a technical lemma where one execution starts in a junction point. Such a lemma would not have been necessary if we had not distinguished regions according to tags.

Lemma 24 (Iterated step consistent on junction point). *Let* β *a partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ *and* $\langle i_0, \rho_0, os_0, h_0\rangle, \langle i_0', \rho_0', os_0', h_0'\rangle \in \text{State}_{\mathcal{G}}$ *two* $\text{JVM}_{\mathcal{G}}$ *states such that*

$$\langle i_0, \rho_0, os_0, h_0\rangle \sim_{S_{i_0},S_{i_0'},\beta} \langle i_0', \rho_0', os_0', h_0'\rangle, \qquad \mathsf{high}(os_0, S_{i_0}), \quad \textit{and} \quad \mathsf{high}(os_0', S_{i_0'})$$

Let $i \in \mathcal{PP}$ *and* $\tau, \tau' \in \{\emptyset\} + \mathcal{C}$ *such that*

$$\begin{array}{ll} i_0 \in \mathsf{region}(i,\tau) & \textit{se is high in region } \mathsf{region}(i,\tau) \\ i_0' = \mathsf{jun}(i,\tau') & \textit{se is high in region } \mathsf{region}(i,\tau') \end{array}$$

Suppose we have a derivation

$$\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots \langle i_k, \rho_k, os_k, h_k\rangle \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r,h)$$

and suppose this derivation is typable with respect to S. *Suppose we have a derivation*

$$\langle i_0', \rho_0', os_0', h_0'\rangle \overset{(n_0')}{\rightsquigarrow}_{m,\tau_0'} \cdots \langle i_k', \rho_k', os_k', h_k'\rangle \overset{(n_k')}{\rightsquigarrow}_{m,\tau_k'} (r',h')$$

and suppose this derivation is typable with respect to S.

Then one of the following case holds:

1. *there exists* j, j' *with* $0 \leq j \leq k$ *and* $0 \leq j' \leq k'$ *such that*

$$i_j = i'_{j'} \quad \textit{and} \quad \langle i_j, \rho_j, os_j, h_j \rangle \sim_{S_{i_j}, S_{i'_j}, \beta} \langle i'_{j'}, \rho_{j'}, os_{j'}, h_{j'} \rangle$$

2. $(r, h) \sim_{\boldsymbol{k_r}, \beta} (r', h')$

Proof. We first invoke lemma 23 on the derivation $\langle i_0, \rho_0, os_0, h_0 \rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots (r, h)$ and the region $\mathsf{region}(i, \tau)$. We have then two cases:

1. There exists j, with $0 < j \leq k$ such that $i_j = \mathsf{jun}(i, \tau)$ and $\langle i_j, \rho_j, os_j, h_j \rangle \sim_{S_{i_j}, S_{i'_0}, \beta} \langle i'_0, \rho'_0, os'_0, h'_0 \rangle$.
   If $\mathsf{jun}(i, \tau) = \mathsf{jun}(i, \tau')$ where are in case 1 with $j' = 0$.
   If not, we can invoke SOAP4: $\mathsf{jun}(i, \tau) \in \mathsf{region}(i, \tau')$ or $\mathsf{jun}(i, \tau') \in \mathsf{region}(i, \tau)$.
   - If $i_j = \mathsf{jun}(i, \tau) \in \mathsf{region}(i, \tau')$, we invoke lemma 23 on the derivation $\langle i_j, \rho_j, os_j, h_j \rangle \overset{(n_j)}{\rightsquigarrow}_{m,\tau_j} \cdots (r, h)$ and the region $\mathsf{region}(i, \tau')$. Either there exists q, with $j < q \leq k$ such that $i_q = \mathsf{jun}(i, \tau') = i'_0$ and $\langle i_q, \rho_q, os_q, h_q \rangle \sim_{S_{i_q}, S_{i'_0}, \beta} \langle i'_0, \rho'_0, os'_0, h'_0 \rangle$ and we can conclude we are in case 1 with $j = q$ and $j' = 0$. Or $k \in \mathsf{region}(i, \tau')$ and we obtain a contradiction (thanks to SOAP3) because $\mathsf{jun}(i, \tau')$ is defined and k is a return point.
   - If $i'_0 = \mathsf{jun}(i, \tau') \in \mathsf{region}(i, \tau)$, we invoke lemma 23 on the derivation $\langle i'_0, \rho'_0, os'_0, h'_0 \rangle \overset{(n'_0)}{\rightsquigarrow}_{m,\tau'_0} \cdots (r', h')$ and the region $\mathsf{region}(i, \tau)$. Either there exists j', with $0 < j' \leq k'$ such that $i'_{j'} = \mathsf{jun}(i, \tau) = i_j$ and $\langle i'_{j'}, \rho_{j'}, os_{j'}, h_{j'} \rangle \sim_{S_{i'_{j'}}, S_{i_0}, \beta} \langle i_0, \rho_0, os_0, h_0 \rangle$ and we can conclude we are in case 1. Or $k' \in \mathsf{region}(i, \tau)$ and we obtain a contradiction (tanks to SOAP3) because $\mathsf{jun}(i, \tau) = i_j$ is defined and k' is a return point.
2. $\mathsf{jun}(i, \tau)$ is undefined, $k \in \mathsf{region}(i, \tau)$, $\mathsf{highResult}_{\boldsymbol{k_r}}(r, h)$ and $h \sim_\beta h'_0$. k is a return point in region $\mathsf{region}(i, \tau)$ so, thanks to SOAP5, we know that $i'_0 = \mathsf{jun}(i, \tau') \in \mathsf{region}(i, \tau)$. We can hence invoke lemma 23 on the derivation $\langle i'_0, \rho'_0, os'_0, h'_0 \rangle \overset{(n'_0)}{\rightsquigarrow}_{m,\tau'_0} \cdots (r', h')$ and the region $\mathsf{region}(i, \tau)$. Either there exists j', with $0 < j' \leq k'$ such that $i'_{j'} = \mathsf{jun}(i, \tau) = i_j$ and $\langle i'_{j'}, \rho_{j'}, os_{j'}, h_{j'} \rangle \sim_{S_{i'_{j'}}, S_{i_0}, \beta} \langle i_0, \rho_0, os_0, h_0 \rangle$ and we can conclude we are in case 1. Or $k' \in \mathsf{region}(i, \tau)$, $\mathsf{highResult}_{\boldsymbol{k_r}}(r', h')$, $h' \sim_\beta h_0$ and we can conclude we are in case 2.

Thanks to the previous lemma, we establish now that after a high branching, both executions stay high until reaching a common point or a return point.

Lemma 25 (Iterated step consistent after branching). *Let* β *a partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ *and* $\langle i_0, \rho_0, os_0, h_0 \rangle, \langle i'_0, \rho'_0, os'_0, h'_0 \rangle \in \mathrm{State}_{\mathcal{G}}$ *two* $\mathrm{JVM}_{\mathcal{G}}$ *states such that*

$$\langle i_0, \rho_0, os_0, h_0 \rangle \sim_{S_{i_0}, S_{i'_0}, \beta} \langle i'_0, \rho'_0, os'_0, h'_0 \rangle \qquad \mathsf{high}(os_0, st_0) \qquad \mathsf{high}(os'_0, st'_0)$$

Let $i \in \mathcal{PP}$ *and* $\tau, \tau' \in \{\emptyset\} + \mathcal{C}$ *such that*

$$i \mapsto^{\tau} i_0 \quad \textit{and} \quad i \mapsto^{\tau'} i'_0$$

Suppose that se is high in region $\mathsf{region}_m(i, \tau)$ *and also in region* $\mathsf{region}_m(i, \tau')$*. Suppose we have a derivation*

$$\langle i_0, \rho_0, os_0, h_0 \rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots \langle i_k, \rho_k, os_k, h_k \rangle \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h)$$

and suppose this derivation is typable with respect to S*. Suppose we have a derivation*

$$\langle i'_0, \rho'_0, os'_0, h'_0 \rangle \overset{(n'_0)}{\rightsquigarrow}_{m,\tau'_0} \cdots \langle i'_k, \rho'_k, os'_k, h'_k \rangle \overset{(n'_k)}{\rightsquigarrow}_{m,\tau'_k} (r', h')$$

and suppose this derivation is typable with respect to S*. Then one of the following case holds:*

1. *there exists* j, j' *with* $0 \leq j \leq k$ *and* $0 \leq j' \leq k'$ *such that* $i_j = i'_{j'}$ *and* $\langle i_j, \rho_j, os_j, h_j\rangle \sim_{S_{i_j}, S_{i'_{j'}}, \beta} \langle i'_{j'}, \rho_{j'}, os_{j'}, h_{j'}\rangle$;
2. $(r, h) \sim_{\boldsymbol{k_r}, \beta} (r', h')$

Proof. If $i_0 = i'_0$ then case 1 trivially holds.

If $i_0 \neq i'_0$ then, using SOAP1 two times, we distinguish four cases:

1. $i_0 \in \mathsf{region}(i, \tau)$ and $i'_0 \in \mathsf{region}(i, \tau')$. We first invoke lemma 23 on the derivation $\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots (r, h)$ and the region $\mathsf{region}(i, \tau)$. We hence obtain two cases. In the first case there exists j, with $0 < j \leq k$ such that $i_j = \mathsf{jun}(i, \tau)$ and $\langle i_j, \rho_j, os_j, h_j\rangle \sim_{S_{i_j}, S_{i'_0}, \beta} \langle i'_0, \rho'_0, os'_0, h'_0\rangle$ and we can conclude 1 thanks to Lemma 24. In the second case $k \in \mathsf{region}(i, \tau)$, $\mathsf{highResult}_{\boldsymbol{k_r}}(r, h)$ and $h \sim_\beta h'_0$. In this case, we invoke lemma 23 on the derivation $\langle i'_0, \rho'_0, os'_0, h'_0\rangle \overset{(n'_0)}{\rightsquigarrow}_{m,\tau'_0} \cdots (r', h')$ and the region $\mathsf{region}(i, \tau')$. We again obtain two cases. If there exists j', with $0 < j' \leq k'$ such that $i_{j'} = \mathsf{jun}(i, \tau')$ and $\langle i_{j'}, \rho_{j'}, os_{j'}, h_{j'}\rangle \sim_{S_{i_{j'}}, S_{i_0}, \beta} \langle i_0, \rho_0, os_0, h_0\rangle$, we can conclude we are in case 1 thanks to Lemma 24. In the second case $k' \in \mathsf{region}(i, \tau')$, $\mathsf{highResult}_{\boldsymbol{k_r}}(r', h')$ and $h' \sim_\beta h_0$. We hence have

$$\begin{cases} \mathsf{highResult}_{\boldsymbol{k_r}}(r, h) \\ \mathsf{highResult}_{\boldsymbol{k_r}}(r', h') \\ h' \sim_\beta h_0 \\ h \sim_\beta h'_0 \\ h_0 \sim_\beta h'_0 \end{cases}$$

   which implies $(r, h) \sim_{\boldsymbol{k_r}, \beta} (r', h')$.
2. $i_0 = \mathsf{jun}(i, \tau)$ and $i'_0 \in \mathsf{region}(i, \tau')$. We easily conclude by Lemma 24.
3. $i_0 \in \mathsf{region}(i, \tau)$ and $i'_0 = \mathsf{jun}(i, \tau')$. We easily conclude by Lemma 24.
4. $i_0 = \mathsf{jun}(i, \tau)$ and $i'_0 = \mathsf{jun}(i, \tau')$. If $\mathsf{jun}(i, \tau) = \mathsf{jun}(i, \tau')$ we are in case 1. If not, SOAP4 applies : $i_0 = \mathsf{jun}(i, \tau) \in \mathsf{region}(i, \tau')$ or $i'_0 = \mathsf{jun}(i, \tau') \in \mathsf{region}(i, \tau)$. In both cases Lemma 24 allows to conclude.

We now must prove similar lemmas about high execution for the special cases where a branching occurs in a return point. This first lemma is a corollary of Lemma 22.

Lemma 26 ($\mathrm{JVM}_{\mathcal{G}}$ **final step consistent).** *Let* β *a partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $\langle i, \rho, os, h\rangle \in \mathrm{State}_{\mathcal{G}}$ *a* $\mathrm{JVM}_{\mathcal{G}}$ *state,* $h_0 \in \mathsf{Heap}$ *a heap and a stack type* $st \in \mathcal{S}^\star$ *such that*

$$h \sim_\beta h_0 \qquad se(i) \not\leq k_{\mathrm{obs}} \qquad \mathsf{high}(os, st)$$

1. *If there exists a state* $\langle i', \rho', os', h'\rangle \in \mathrm{State}_{\mathcal{G}}$, *a tag* $\tau \in \{\emptyset\} + \mathcal{C}$ *and a stack type* $st' \in \mathcal{S}^\star$ *such that*

$$\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} \langle i, \rho', os', h'\rangle \qquad i \vdash^\tau st \Rightarrow st'$$

   then

$$\mathsf{high}(os', st') \quad \text{and} \quad h' \sim_\beta h_0$$

2. *If there exists a result* $r \in \mathcal{V} + \mathcal{L}$, *a heap* $h' \in \mathsf{Heap}$ *and a tag* $\tau \in \{\emptyset\} + \mathcal{C}$ *such that*

$$\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} r, h' \qquad i \vdash^\tau st \Rightarrow$$

   then

$$\mathsf{highResult}_{\boldsymbol{k_r}}(r, h') \quad \text{and} \quad h' \sim_\beta h_0$$

Proof. Similar to the proof of lemma 22.

Lemma 27 (Iterated final step consistent after branching). *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $\langle i_0, \rho_0, os_0, h_0\rangle \in \mathrm{State}_{\mathcal{G}}$ a $\mathrm{JVM}_{\mathcal{G}}$ states and $h'_0 \in \mathsf{Heap}$ a heap such that*

$$h_0 \sim_\beta h'_0 \qquad \mathsf{high}(os_0, S_{i_0})$$

Let $i \in \mathcal{PP}$ and $\tau, \tau' \in \{\emptyset\} + \mathcal{C}$ such that

$$i_0 \in \mathsf{region}(i, \tau) \quad \textit{and} \quad i \mapsto^{\tau'}$$

Suppose that se is high in region $\mathsf{region}_m(i, \tau)$. Suppose we have a derivation

$$\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots \langle i_k, \rho_k, os_k, h_k\rangle \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h)$$

and suppose this derivation is typable with respect to S. Then

$$\mathsf{highResult}_{\boldsymbol{k_r}}(r, h) \quad \text{and} \quad h \sim_\beta h'_0$$

Proof. By induction on k.

- $k = 0$ we can directly apply case 2 of lemma 26 to conclude.
- we suppose the statement is true for a given k and we prove it now for $k+1$. First note that $se(i_0) \not\leq k_{\mathrm{obs}}$, $\mathsf{high}(os_0, S_{i_0})$ and $i_0 \vdash S_{i_0} \Rightarrow st_1$ hold for some st_1 such that $st_1 \sqsubseteq S_{i_1}$, so we have

  $$h_1 \sim_\beta h'0 \quad \text{and} \quad \mathsf{high}(os_1, st_1)$$

  by case 1 of lemma 26 and by lemma 21.
  Thanks to subtyping lemma 5 we have also:

  $$\mathsf{high}(os_1, S_{i_1})$$

  Then, using SOAP2, we have $i_1 \in \mathsf{region}(i, \tau)$ or $i_1 = \mathsf{jun}(i, \tau)$.
  In the first case, it is sufficient to invoke induction hypothesis on derivation

  $$\langle i_1, \rho_1, os_1, h_1\rangle \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} \cdots \langle i_{k+1}, \rho_{k+1}, os_{k+1}, h_{k+1}\rangle \overset{(n_{k+1})}{\rightsquigarrow}_{m,\tau_{k+1}} (r, h)$$

  to conclude. Let's justify we can use it:
  - $i_1 \in \mathsf{region}(i, \tau)$;
  - the derivation is on length k and is typable;
  - $h_1 \sim_\beta h'_0$ and $\mathsf{high}(os_1, S_1)$ hold

  In the second case we have a contradiction because SOAP3 implies that $\mathsf{jun}(i, \tau)$ is undefined.

This lemma ends the proof about high executions. We now examine parallel executions of methods, proving first the two remaining lemmas sketched in section 2. Method calls requires to use a notion of *non-interference at order n* to state these lemmas.

Definition 31 (non-interference at order n). *A method m is* non-interferent at order n *with respect to a security signature $\boldsymbol{k_v} \longrightarrow \boldsymbol{k_r}$, if for every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every states $\langle 1, \rho_1, \epsilon, h_1\rangle, \langle 1, \rho_2, \epsilon, h_2\rangle \in \mathrm{State}_{\mathrm{JVM}_{\mathcal{G}}}$, every heaps $h'_1, h'_2 \in \mathsf{Heap}$, every results $r_1, r_2 \in \mathcal{V} + \mathcal{L}$ and every integer $n_1, n_2 \in \mathbb{N}$ such that*

$$\langle i, \rho_1, os_1, h_1\rangle \Downarrow_m^{(n_1)} r_1, h'_1, \quad \langle i, \rho_2, os_2, h_2\rangle \Downarrow_m^{(n_2)} r_2, h'_2, \quad n_1 \leq n \quad n_2 \leq n$$

and

$$\langle 1, \rho_1, \epsilon, h_1\rangle \sim_{\beta,\epsilon,\epsilon} \langle 1, \rho_2, \epsilon, h_2\rangle$$

there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that

$$\beta \subseteq \beta' \quad \textit{and} \quad (r_1, h'_1) \sim_{\boldsymbol{k_r},\beta'} (r_2, h'_2)$$

Lemma 28 ($\mathrm{JVM}_{\mathcal{G}}$ high branching). *Let n an integer such that all method m' in P are non-interferent at all order k, $k < n$, with respect to all the policies in* $\mathrm{Policies}_{\Gamma}(\mathrm{m}')$.

Let m a method in P, $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ a partial function, $s_1, s_2 \in \mathrm{State}_{\mathcal{G}}$ two $\mathrm{JVM}_{\mathcal{G}}$ states at the same program point i and two stack types $st_1, st_2 \in \mathcal{S}^\star$ such that

$$s_1 \sim_{st_1, st_2, \beta} s_2$$

1. *If there exists two states $\langle i_1, \rho_1', os_1', h_1' \rangle, \langle i_2, \rho_2', os_2', h_2' \rangle \in \mathrm{State}_{\mathcal{G}}$ and two stack types $st_1', st_2' \in \mathcal{S}^\star$ such that*

$$\begin{array}{rl} & i_1 \neq i_2 \\ s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} \langle i_1, \rho_1', os_1', h_1' \rangle & n_1 \leq n \\ s_2 \overset{(n_2)}{\rightsquigarrow}_{m,\tau_2} \langle i_2, \rho_2', os_2', h_2' \rangle & n_2 \leq n \\ i \vdash^{\tau_1} st_1 \Rightarrow st_1' \quad i \vdash^{\tau_2} st_2 \Rightarrow st_2' & \end{array}$$

   then

$$\begin{array}{ll} \mathsf{high}(os_1', st_1') & \textit{and} \quad \textit{se is high in } \mathsf{region}(i, \tau_1) \\ \mathsf{high}(os_2', st_2') & \textit{and} \quad \textit{se is high in } \mathsf{region}(i, \tau_2) \end{array}$$

2. *If there exists a state $\langle i_1, \rho_1', os_1', h_1' \rangle \in \mathrm{State}_{\mathcal{G}}$, a final result $(r_2, h_2') \in (\mathcal{V} + \mathcal{L}) \times \mathsf{Heap}$ and a stack type $st_1' \in \mathcal{S}^\star$ such that*

$$\begin{array}{rl} s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} \langle i_1, \rho_1', os_1', h_1' \rangle & n_1 \leq n \\ s_2 \overset{(n_2)}{\rightsquigarrow}_{m,\tau_2} (r_2, h_2') & n_2 \leq n \\ i \vdash^{\tau_1} st_1 \Rightarrow st_1' \quad i \vdash^{\tau_2} st_2 \Rightarrow & \end{array}$$

   then

$$\mathsf{high}(os_1', st_1') \quad \textit{and} \quad \textit{se is high in } \mathsf{region}(i, \tau_1)$$

Proof. By case analysis on $P_m[i]$. We only consider branching instructions here.

Case: $P_m[i] = \mathsf{ifeq}\ j$. By semantics, $\tau_1 = \emptyset$, $\tau_2 = \emptyset$, $s_1 = \langle i, \rho_1, n_1 :: os_1, h_1 \rangle$ and $s_2 = \langle i, \rho_2, n_2 :: os_2, h_2 \rangle$. Since $i_1 \neq i_2$, we have necessarily $n_1 \neq n_2$. By typability, $st_1 = k_1 :: st_1''$, $st_2 = k_2 :: st_2''$, $st_1' = \mathrm{lift}_{k_1} st_1''$ and $st_2' = \mathrm{lift}_{k_2} st_2''$. Since $n_1 :: os_1 \sim_{k_1 :: st_1'', k_2 :: st_2'', \beta} n_2 :: os_2$, and $n_1 \neq n_2$ we deduce that $k_1 \not\leq k_{\mathrm{obs}}$ and $k_2 \not\leq k_{\mathrm{obs}}$. The first consequence is $\mathsf{high}(os_1, \mathrm{lift}_{k_1} st_1'')$ and $\mathsf{high}(os_2, \mathrm{lift}_{k_2} st_2'')$. By typability, $\forall j' \in \mathsf{region}(i, \emptyset),\ k_1 \leq se(j')$ and hence se is high in region $\mathsf{region}(i, \emptyset)$.

Case: $P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}}$. There are several kinds of branching for method calls. s_1 is necessarily of the form $\langle i, \rho_1, os_1 :: v_1 :: os_1', h_1 \rangle$ and s_2 of the form $\langle i, \rho_2, os_2 :: v_2 :: os_2', h_2 \rangle$ with $v_1, v_2 \in \mathcal{L} \cup \{null\}$. Furthermore, $st_1 = st_1' :: k_1 :: st_1''$ and $st_2 = st_2' :: k_2 :: st_2''$. We then make a first distinction according if one of v_1 or v_2 is equal to *null*.

Case 1: v_1 or v_2 is equal to *null*. There is only a branching if the other value is not null. We only make the case $v_1 = null$ and $v_2 = l_2 \in \mathcal{L}$, the other case is done by symmetry.

By hypothesis

$$os_1 :: null :: os_1' \sim_{st_1' :: k_1 :: st_1'', st_2' :: k_2 :: st_2'', \beta} os_2 :: l_2 :: os_2'$$

and $\mathsf{length}(os_1) = \mathsf{length}(st_1) = \mathsf{length}(os_2) = \mathsf{length}(st_2)$, so necessarily $k_1 \not\leq k_{\mathrm{obs}}$ and $k_2 \not\leq k_{\mathrm{obs}}$. We deduce then $\mathsf{high}(os_1', (k_1 \sqcup \boldsymbol{k_r'}[\mathbf{np}]) :: \epsilon)$ (if $\mathsf{Handler}_m(i, \mathbf{np}) = t$) and that se is high in region $\mathsf{region}(i, \emptyset)$ thanks to the typability constraint $\forall j \in \mathsf{region}(i, \emptyset),\ (k_1 \sqcup \boldsymbol{k_r'}[\mathbf{np}]) \leq se(j)$ (which occurs in both typing rules, if **np** is caught or not).

We then examine the transition $\langle i, \rho_2, os_2 :: l_2 :: os_2', h_2 \rangle \overset{(n_2)}{\rightsquigarrow}_m \cdots$. There are three cases: normal termination of the called method, termination by an exception which can be caught or uncaught in

m. In all cases the corresponding typing rules enforce that all element of the next stack type (if there is any one) are greater or equal to k_2 and for all point j in $\mathsf{region}(i, \tau_2)$ $k_2 \leq se(i)$. Hence we are done since $k_2 \not\leq k_{\text{obs}}$.

Case 2: both v_1 and v_2 are in $\mathcal{L}$ (we note them l_1 and l_2 from now). By hypothesis

$$os_1 :: l_1 :: os_1' \sim_{st_1'::k_1::st_1'', st_2'::k_2::st_2'', \beta} os_2 :: l_2 :: os_2'$$

Since $\mathsf{length}(os_1) = \mathsf{length}(st_1) = \mathsf{length}(os_2) = \mathsf{length}(st_2)$, we have by case analysis on lemme 34:

- either $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. In this case we make a similar reasoning as before . There are three cases for the first execution and three others for the second but in all cases the corresponding typing rules enforce that all element of the next stack type (if there is any one) are greater or equal to k_1 (or k_2) and for all point j in $\mathsf{region}(i, \tau_1)$ (or $\mathsf{region}(i, \tau_2)$) $k_1 \leq se(i)$ (or $k_2 \leq se(i)$).
- or $k_1 = k_2$ and $l_1 \sim_\beta l_2$. Since $h_1 \sim_\beta h_2$ we deduce $\mathsf{class}(h_1(l_1)) = \mathsf{class}(h_2(l_2))$ and as a consequence the same method m' is called in both execution. $\Gamma_{m_{\text{ID}}}[k_1]$ and $\Gamma_{m_{\text{ID}}}[k_2]$ are then equals to a same signature $\boldsymbol{k_v'} \xrightarrow{k_h'} \boldsymbol{k_r'}$. Again, there are three cases for each execution but each times we have

$$\{this \mapsto l_1, \boldsymbol{x} \mapsto os_1\} \sim_{\boldsymbol{k_v'}, \beta} \{this \mapsto l_2, \boldsymbol{x} \mapsto os_2\}$$

thanks to lemma 35 and typability hypotheses

$$k_1 \leq \boldsymbol{k_v'}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \boldsymbol{k_v'}[i+1]$$
$$k_2 \leq \boldsymbol{k_v'}[0] \qquad \forall i \in [0, \mathsf{length}(st_2) - 1],\ st_2[i] \leq \boldsymbol{k_v'}[i+1]$$

Now, since m' is non-interferent at order $\max(n_1', n_2') < n$ (with $n' + 1 = n_1$ and $n_2' + 1 = n_2$) and

$$\langle 1, \{this \mapsto l_1, \boldsymbol{x} \mapsto os_1\}, \epsilon, h_1\rangle \Downarrow_{m'}^{(n_1')} r_1, h_1'$$

and

$$\langle 1, \{this \mapsto l_2, \boldsymbol{x} \mapsto os_2\}, \epsilon, h_2\rangle \Downarrow_{m'}^{(n_2')} r_2, h_2'$$

we deduce that

$$(r_1, h_1') \sim_{\beta', \boldsymbol{k_r'}} (r_2, h_2') \tag{1}$$

for some β' such that $\beta \subseteq \beta'$. The nature of r_1 and r_2 depends on the kind of transition that occurs:

* $\tau_1 = \emptyset$ and $\tau_2 = e \in \mathcal{C} : e$ is caught or uncaught in m but in both cases, (1) gives $\boldsymbol{k_r'}[e] \not\leq k_{\text{obs}}$. By typability, $\forall j \in \mathsf{region}(i, \emptyset),\ k_1 \sqcup k_e \leq se(j)$ with $k_e = \bigsqcup \{\ \boldsymbol{k_r'}[e_0] \mid e_0 \in \mathsf{excAnalysis}(m_{\text{ID}})\ \}$ and $\forall j \in \mathsf{region}(i, e),\ k_2 \sqcup \boldsymbol{k_r'}[e] \leq se(j)$. Hence se is high in $\mathsf{region}(i, \emptyset)$ and in $\mathsf{region}(i, e)$ since $\boldsymbol{k_r'}[e] \not\leq k_{\text{obs}}$ and by the same way $k_e \not\leq k_{\text{obs}}$.
* $\tau_1 = e_1 \in \mathcal{C}$ and $\tau_2 = e_2 \in \mathcal{C}$: branching only occurs if $e_1 \neq e_2$. But since $(\langle l_1'\rangle, h_1') \sim_{\beta', \boldsymbol{k_r'}} (\langle l_2'\rangle, h_2')$ with $e_j = \mathsf{class}(h_j'(l_j'))$, $j \in \{1, 2\}$, we have necessarily $\boldsymbol{k_r}[e_1] \not\leq k_{\text{obs}}$ and $\boldsymbol{k_r}[e_2] \not\leq k_{\text{obs}}$.
  By typability, $\forall j \in \mathsf{region}(i, e_1),\ k_1 \sqcup \boldsymbol{k_r'}[e_1] \leq se(j)$ and $\forall j \in \mathsf{region}(i, e_2),\ k_2 \sqcup \boldsymbol{k_r'}[e_2] \leq se(j)$ so se is high in region $\mathsf{region}(i, e_1)$ and $\mathsf{region}(i, e_2)$.
  Let $j \in \{1, 2\}$, If e_j is uncaught there is nothing to prove. If e_j is caught we must establish that $\mathsf{high}(l_j' :: \epsilon, (k_j \sqcup \boldsymbol{k_r'}[e_j]) :: \varepsilon)$. This is easy since $\boldsymbol{k_r}[e_j] \not\leq k_{\text{obs}}$.
* in the others cases, either there is no branching or the case is symmetric to a previous one.

Case: $P_m[i] = \mathsf{getfield}\ f$. Branching only occurs if s_1 is of the form $\langle i, \rho_1, null :: os_1, h_1\rangle$ and s_2 of the form $\langle i, \rho_2, l_2 :: os_2, h_2\rangle$ (or in the symmetric case). Hence $\tau_1 = \mathbf{np}$ and $\tau_2 = \emptyset$ and by typability, $st_1 = k_1 :: st_1''$ and $st_2 = k_2 :: st_2''$.
Since $null :: os_1 \sim_{k_1::st_1'', k_2::st_2'', \beta} l_2 :: os_2$, we have necessarily $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. By typability, $\forall j \in \mathsf{region}(i, \mathbf{np}),\ k_1 \leq se(j)$ and $\forall j \in \mathsf{region}(i, \emptyset),\ k_2 \leq se(j)$. Hence se is high in region $\mathsf{region}(i, \mathbf{np})$ and $\mathsf{region}(i, \emptyset)$.

Concerning operand stacks, we have by typability $st'_2 = \mathrm{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st''_2)$ and since $k_2 \not\leq k_{\mathrm{obs}}$ we deduce that $\mathsf{high}(os'_2, \mathrm{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st''_2))$.

In the first transition there is something to prove only if **np** is caught. We must then establish that $\mathsf{high}(l'_1 :: \epsilon, (k_1 \sqcup se(i)) :: \epsilon)$. Since $k_1 \not\leq k_{\mathrm{obs}}$ we are done.

Case: $P_m[i] = \mathsf{putfield}\ f$. Branching only occurs if s_1 is of the form $\langle i, \rho_1, v_1 :: null :: os_1, h_1\rangle$ and s_2 of the form $\langle i, \rho_2, v_2 :: l_2 :: os_2, h_2\rangle$ (or in the symmetric case). Hence $\tau_1 = \mathbf{np}$ and $\tau_2 = \emptyset$ and by typability, $st_1 = k_1 :: k'_1 :: st''_1$ and $st_2 = k_2 :: k'_2 :: st''_2$.

Since $v_1 :: null :: os_1 \sim_{k_1::k'_1::st''_1, k_2::k'_2::st''_2, \beta} v_2 :: l_2 :: os_2$, we have necessarily $k'_1 \not\leq k_{\mathrm{obs}}$ and $k'_2 \not\leq k_{\mathrm{obs}}$. By typability, $\forall j \in \mathsf{region}(i, \mathbf{np}),\ k'_1 \leq se(j)$ and $\forall j \in \mathsf{region}(i, \emptyset),\ k'_2 \leq se(j)$. Hence se is high in region $\mathsf{region}(i, \mathbf{np})$ and $\mathsf{region}(i, \emptyset)$.

Concerning operand stacks, we have by typability $st'_2 = \mathrm{lift}_{k'_2}(st''_2)$ and since $k'_2 \not\leq k_{\mathrm{obs}}$ we deduce that $\mathsf{high}(os_2, \mathrm{lift}_{k'_2}(:: st''_2))$.

In the first transition there is something to prove only if **np** is caught. We must then establish that $\mathsf{high}(l'_1 :: \epsilon, ('k_1 \sqcup se(i)) :: \epsilon)$. Since $k'_1 \not\leq k_{\mathrm{obs}}$ we are done.

Case: $P_m[i] = \mathsf{throw}$. s_1 is of the form $\langle i, \rho_1, v_1 :: os_1, h_1\rangle$ and s_2 of the form $\langle i, \rho_2, v_2 :: os_2, h_2\rangle$ with v_1 and v_2 in $\mathcal{L} \cup \{\mathbf{np}\}$. By typability, $st_1 = k_1 :: st''_1$ and $st_2 = k_2 :: st''_2$.

We then make a first distinction according if one of v_1 or v_2 is equal to *null*. In both cases we show that $k_1 \not\leq k_{\mathrm{obs}}$ and $k_2 \not\leq k_{\mathrm{obs}}$.

Case 1: v_1 or v_2 is equal to *null*. There is only a branching if the other value is not null. We only make the case $v_1 = null$ and $v_2 = l_2 \in \mathcal{L}$, the other case is done by symmetry.

By hypothesis,

$$null :: os_1 \sim_{k_1::st''_1, k_2::st''_2, \beta} l_2 :: os_2$$

Hence we have necessarily $k_1 \not\leq k_{\mathrm{obs}}$ and $k_2 \not\leq k_{\mathrm{obs}}$.

Case 2: both v_1 and v_2 are in $\mathcal{L}$ (we note them l_1 and l_2 from now). There is a branching only if $h_1(l_1)$ and $h_2(l_2)$ are of distinct classes. This has for consequence that $k_1 \not\leq k_{\mathrm{obs}}$ and $k_2 \not\leq k_{\mathrm{obs}}$, since by hypothesis

$$l_1 :: os_1 \sim_{k_1::st''_1, k_2::st''_2, \beta} l_2 :: os_2$$

Now, by typability we have $\forall j \in \mathsf{region}(i, \tau_1),\ k_1 \leq se(j)$ and $\forall j \in \mathsf{region}(i, \tau_2),\ k_2 \leq se(j)$. Hence se is high in region $\mathsf{region}(i, \tau_1)$ and $\mathsf{region}(i, \tau_2)$.

Concerning operand stacks, for $i \in \{1, 2\}$ we have something to prove only if τ_i is caught. We must then establish that $\mathsf{high}(l_i :: \epsilon, (k_i \sqcup se(i)) :: \epsilon)$. Since $k_i \not\leq k_{\mathrm{obs}}$ we are done.

Lemma 29 (JVM$_{\mathcal{G}}$ locally respect). *Let n an integer such that all method m' in P are non-interferent at all order k, $k < n$, with respect to all the policies in* $\mathrm{Policies}_\Gamma(\mathrm{m}')$.

Let m a method in P, $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ a partial function, $s_1, s_2 \in \mathrm{State}_{\mathcal{G}}$ two JVM$_{\mathcal{G}}$ states at the same program point i and two stack types $st_1, st_2 \in \mathcal{S}^\star$ such that $s_1 \sim_{st_1, st_2, \beta} s_2$.

1. *If there exists two states $s'_1, s'_2 \in \mathrm{State}_{\mathcal{G}}$ and two stack types $st'_1, st'_2 \in \mathcal{S}^\star$ such that*

$$\begin{gathered} s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} s'_1, \quad n_1 \leq n \\ s_2 \overset{(n_2)}{\rightsquigarrow}_{m,\tau_2} s'_2, \quad n_2 \leq n \\ i \vdash^{\tau_1} st_1 \Rightarrow st'_1 \quad i \vdash^{\tau_2} st_2 \Rightarrow st'_2 \end{gathered}$$

then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that

$$s'_1 \sim_{st'_1, st'_2, \beta} s'_2 \quad \textit{and} \quad \beta \subseteq \beta'$$

2. *If there exists a state $\langle i'_1, \rho'_1, os'_1, h'_1\rangle \in \mathrm{State}_{\mathcal{G}}$, a final result $(r_2, h'_2) \in (\mathcal{V} + \mathcal{L}) \times \mathsf{Heap}$ and a stack types $st'_1, \in \mathcal{S}^\star$ such that*

$$\begin{gathered} s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} \langle i'_1, \rho'_1, os'_1, h'_1\rangle \quad n_1 \leq n \\ s_2 \overset{(n_2)}{\rightsquigarrow}_{m,\tau_2} (r_2, h'_2) \quad n_2 \leq n \\ i \vdash^{\tau_1} st_1 \Rightarrow st'_1 \quad i \vdash^{\tau_2} st_2 \Rightarrow \end{gathered}$$

then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ *such that*

$$h'_1 \sim_{\beta'} h'_2, \quad \mathsf{highResult}_{\boldsymbol{k_r}}(r_2, h'_2) \quad \textit{and} \quad \beta \subseteq \beta'$$

3. *If there exists two final results* $(r_1, h'_1), (r_2, h'_2) \in (\mathcal{V} + \mathcal{L}) \times \mathsf{Heap}$ *such that*

$$\begin{array}{rl} s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} (r_1, h'_1) & n_1 \leq n \\ s_2 \overset{(n_2)}{\rightsquigarrow}_{m,\tau_2} (r_2, h'_2) & n_2 \leq n \\ i \vdash st_1 \Rightarrow \quad i \vdash st_2 \Rightarrow & \end{array}$$

then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ *such that*

$$(r_1, h'_1) \sim_{\boldsymbol{k_r}, \beta'} (r_2, h'_2) \quad \textit{and} \quad \beta \subseteq \beta'$$

Proof. By a case analysis on the instruction that is executed.

Case: $P_m[i] = \mathsf{push}\ n$

By semantics, $s_1 = \langle i, \rho_1, os_1, h_1 \rangle$, and $s_2 = \langle i, \rho_2, os_2, h_2 \rangle$, and $s'_1 = \langle i+1, \rho_1, n :: os_1, h_1 \rangle$, and $s'_2 = \langle i+1, \rho_2, n :: os_2, h_2 \rangle$. By typability, $st'_1 = se(i) :: st_1$ and $st'_2 = se(i) :: st_2$.

We take $\beta' = \beta$. Local variable and heaps are not modified so indistinguishability properties on them still hold. For operand stacks, by hypothesis we have $os_1 \sim_{st_1, st_2, \beta} os_2$. Hence and since $n \sim_\beta n$ holds, we have $n :: os_1 \sim_{se(i)::st_1, se(i)::st_2, \beta} n :: os_2$, thanks to lemma 32.

Case: $P_m[i] = \mathsf{binop}\ op$

By semantics, $s_1 = \langle i, \rho_1, n_1 :: n_2 :: os_1, h_1 \rangle$, and $s_2 = \langle i, \rho_2, n'_1 :: n'_2 :: os_2, h_2 \rangle$ and $s'_1 = \langle i+1, \rho_1, n :: os_1, h_1 \rangle$, and $s'_2 = \langle i+1, \rho_2, n' :: os_2, h_2 \rangle$. By typability, $st_1 = k_1 :: k_2 :: st$, $st'_1 = k_1 \sqcup k_2 \sqcup se(i) :: st$, $st_2 = k'_1 :: k'_2 :: st'$ and $st'_2 = k'_1 \sqcup k'_2 \sqcup se(i) :: st'$.

We take $\beta' = \beta$. Local variable and heaps are not modified so indistinguishability properties on them still hold. For operand stacks, we make two cases:

- $k_1 \sqcup k_2 \leq k_{\text{obs}}$. In this case, since $k_1 \leq k_{\text{obs}}$ and $k_2 \leq k_{\text{obs}}$, lemma 34 and hypothesis $n_1 :: n_2 :: os_1 \sim_{\beta, k_1::k_2::st, k'_1::k'_2::st'} n'_1 :: n'_2 :: os_2$ give $k_1 = k'_1$, $k_2 = k'_2$, $n_1 = n'_1$, $n_2 = n'_2$ and $os_1 \sim_{st, st'} os_2$. Hence $n = n'$ and $k_1 \sqcup k_2 \sqcup se(i) = k'_1 \sqcup k'_2 \sqcup se(i)$ so $n :: os_1 \sim_{k'_1 \sqcup k'_2 \sqcup se(i)::st', k'_1 \sqcup k'_2 \sqcup se(i)::st', \beta} n' :: os_2$ by lemma 32.
- $k_1 \sqcup k_2 \not\leq k_{\text{obs}}$. Hence $k_1 \not\leq k_{\text{obs}}$ or $k_2 \not\leq k_{\text{obs}}$ and thanks to lemma 34, $k'_1 \not\leq k_{\text{obs}}$ or $k'_2 \not\leq k_{\text{obs}}$. In both cases $k'_1 \sqcup k'_2 \not\leq k_{\text{obs}}$. We hence conclude that $k_1 \sqcup k_2 \sqcup se(i) \not\leq k_{\text{obs}}$ and $k'_1 \sqcup k'_2 \sqcup se(i) \not\leq k_{\text{obs}}$. Since $os_1 \sim_{st, st'} os_2$, we finally obtain $n :: os_1 \sim_{k'_1 \sqcup k'_2 \sqcup se(i)::st', k'_1 \sqcup k'_2 \sqcup se(i)::st', \beta} n' :: os_2$.

Case: $P_m[i] = \mathsf{load}\ x$

By semantics, $s_1 = \langle i, \rho_1, os_1, h_1 \rangle$, and $s_2 = \langle i, \rho_2, os_2, h_2 \rangle$, and $s'_1 = \langle i+1, \rho_1, \rho_1(x) :: os_1, h_1 \rangle$, and $s'_2 = \langle i+1, \rho_2, \rho_2(x) :: os_2, h_2 \rangle$. By typability, $st'_1 = \boldsymbol{k_v}_m(x) \sqcup se(i) :: st_1$ and $st'_2 = \boldsymbol{k_v}_m(x) \sqcup se(i) :: st_2$.

We take $\beta' = \beta$. Local variables and heaps are not modified so indistinguishability properties on them still hold.

For operand stacks, by hypothesis we have $os_1 \sim_{st_1, st_2, \beta} os_2$ and also by hypothesis of variable indistinguishability we have $\rho_1(x) \sim_\beta \rho_2(x)$ if $\boldsymbol{k_v}(x) \leq k_{\text{obs}}$. We hence conclude by lemma 31.

Case: $P_m[i] = \mathsf{store}\ x$

By semantics, $s_1 = \langle i, \rho_1, v :: os_1, h_1 \rangle$, and $s_2 = \langle i, \rho_2, v' :: os_2, h_2 \rangle$, and $s'_1 = \langle i+1, \rho_1 \oplus \{x \mapsto v\}, os_1, h_1 \rangle$, and $s'_2 = \langle i+1, \rho_2 \oplus \{x \mapsto v'\}, os_2, h_2 \rangle$. By typability, $st_1 = k_1 :: st'_1$ and $st_2 = k_2 :: st'_2$.

We take $\beta' = \beta$. Heaps are not modified so indistinguishability properties on them still hold.

For local variables we have to check for every variable y that $\rho_1 \oplus \{x \mapsto v\}(y) \sim_\beta \rho_2 \oplus \{x \mapsto v'\}(y)$ whenever $\boldsymbol{k_v}(y) \leq k_{\text{obs}}$. This is valid by hypothesis for every variable except x. For x we must prove that $v \sim_\beta v'$ whenever $\boldsymbol{k_v}(x) \leq k_{\text{obs}}$. We then make a case analysis using lemma 34 on hypothesis $v :: os_1 \sim_{k_1::st'_1, k_2::st'_2, \beta} v' :: os_2$.

- In first case $k_1 = k_2$, $k_1 \leq k_{\text{obs}}$ and $v \sim_\beta v'$. This last hypothesis allows us to conclude our proof of $\rho_1 \sim_\beta \rho_2$.

– In second case $k_1 \leq k_{\text{obs}}$ and $k_2 \leq k_{\text{obs}}$. By typability, we have $k_1 \leq \boldsymbol{k_v}(x)$ and $k_2 \leq \boldsymbol{k_v}(x)$. Hence $\boldsymbol{k_v}(x) \not\leq k_{\text{obs}}$ and we can conclude our proof of $\rho_1 \sim_\beta \rho_2$.

For operand stacks, $os_1 \sim_{st'_1, st'_2, \beta} os_2$ easily come from lemma 34.

Case: $P_m[i] = \mathsf{ifeq}\ j$

By semantics, $s_1 = \langle i, \rho_1, n :: os_1, h_1\rangle$, $s_2 = \langle i, \rho_2, n' :: os_2, h_2\rangle$, $s'_1 = \langle i'_1, \rho_1, os_1, h_1\rangle$ (where i'_1 can be $i+1$ or j) and $s'_2 = \langle i'_2, \rho_2, os_2, h_2\rangle$ (where i'_2 can be $i+1$ or j). By typability, $st_1 = k :: st$, $st_2 = k' :: st'$, $st'_1 = \text{lift}_k(st)$ and $st'_2 = \text{lift}_{k'}(st')$.

We take $\beta' = \beta$. Local variables and heaps are not modified so indistinguishability properties on them still hold.

Operand stack indistinguishability $os_1 \sim_{st, st', \beta} os_2$ finally holds because of hypothesis $n :: os_1 \sim_{k::st, k'::st', \beta} n' :: os_2$ and lemma 34.

Case: $P_m[i] = \mathsf{goto}\ j$

This case is trivial. We take $\beta' = \beta$. Neither local variables, operand stacks or heaps are modified so indistinguishability properties on them still hold.

Case: $P_m[i] = \mathsf{return}$

By semantics, $s_1 = \langle i, \rho_1, v :: os_1, h_1\rangle$, and $s_2 = \langle i, \rho_2, v' :: os_2, h_2\rangle$, and $(r_1, h'_1) = (v, h_1)$, and $(r_2, h'_2) = (v', h_2)$. By typability, st_1 is of the form $k :: st$ and st_2 is of the form $k' :: st'$.

We have to prove $(v, h_1) \sim_{\beta, \boldsymbol{k_r}} (v', h_2)$, that is, $h_1 \sim_\beta h_2$ and $\boldsymbol{k_r}[n] \leq k_{\text{obs}} \Rightarrow v \sim_\beta v'$.

Heap indistinguishability holds by hypothesis. We then make a case analysis on hypothesis $v :: os_1 \sim_{k::st, k'::st', \beta} v' :: os_2$ using lemma 34.

– In first case $k = k'$, $k \leq k_{\text{obs}}$ and $v \sim_\beta v'$. This last hypothesis allows us to prove $\boldsymbol{k_r}[n] \leq k_{\text{obs}} \Rightarrow v \sim_\beta v'$.

– In second case $k \leq k_{\text{obs}}$ and $k' \leq k_{\text{obs}}$. Hence $\boldsymbol{k_r}[n] \not\leq k_{\text{obs}}$ since by typability, $k \leq \boldsymbol{k_r}[n]$ and $k' \leq \boldsymbol{k_r}[n]$. So $\boldsymbol{k_r}[n] \leq k_{\text{obs}} \Rightarrow v \sim_\beta v'$ is trivially true.

Case: $P_m[i] = \mathsf{new}\ C$

By semantics, $s_1 = \langle i, \rho_1, os_1, h_1\rangle$, $s_2 = \langle i, \rho_2, os_2, h_2\rangle$, $s'_1 = \langle i+1, \rho_1, l :: os_1, h_1 \oplus \{l \mapsto \mathsf{default}_C\}\rangle$, where $l = \text{fresh}(h_1)$, and $s'_1 = \langle i+1, \rho_2, l' :: os_2, h_2 \oplus \{l' \mapsto \mathsf{default}_C\}\rangle$, where $l' = \text{fresh}(h_2)$. By typability, $st'_1 = se(i) :: st_1$ and $st'_2 = se(i) :: st_2$.

We first choose β' according to a case analysis on $se(i)$. If $se(i) \leq k_{\text{obs}}$, we choose $\beta' = \beta \oplus \{l \mapsto l'\}$. If $se(i) \not\leq k_{\text{obs}}$, we choose $\beta' = \beta$. In both case $\beta \subseteq \beta'$ thanks to lemma 40.

Local variables are not modified so indistinguishability properties on them still hold for β', thanks to lemma 44.

For operand stack indistinguishability, $os_1 \sim_{\beta, st_1, st_2} os_2$ and since $\beta \subseteq \beta'$, $os_1 \sim_{\beta', st_1, st_2} os_2$ also holds, thanks to lemma 45. If $se(i) \not\leq k_{\text{obs}}$, we obtain by definition of operand stack indistinguishability, $l :: os_1 \sim_{\beta', se(i)::st_1, se(i)::st_2} l' :: os_2$. If $se(i) \leq k_{\text{obs}}$, we only have to prove $l \sim_{\beta'} l'$ to conclude. This follow easily from definition of β' since $\beta'(l) = l'$.

Heap indistinguishability comes from lemma 40 when $se(i) \leq k_{\text{obs}}$ and from lemma 41 when $se(i) \not\leq k_{\text{obs}}$.

Case: $P_m[i] = \mathsf{putfield}\ f$

By semantics, $s_1 = \langle i_1, \rho_1, v :: l :: os_1, h_1\rangle$ and $s_2 = \langle i_2, \rho_2, v' :: l' :: os_2, h_2\rangle$, but there are several options to consider for the successors of s_1 and s_2:

– There are no exceptions ($\tau_1 = \emptyset$ and $\tau_2 = \emptyset$) and $s'_1 = \langle i_1 + 1, \rho_1, os_1, h_1 \oplus \{l \mapsto h_1(l) \oplus \{f \mapsto v\}\}\rangle$ and $s'_2 = \langle i_1 + 1, \rho_2, os_2, h_2 \oplus \{l' \mapsto h_2(l') \oplus \{f \mapsto v'\}\}\rangle$. By typability, $st_1 = k_1 :: k'_1 :: st$, $st_2 = k_2 :: k'_2 :: st'$, $st'_1 = \mathsf{lift}_{k'_1} st$ and $st'_2 = \mathsf{lift}_{k'_2} st'$.

We take $\beta' = \beta$.

By hypothesis and semantics (variables do not change) we have indistinguishability of variables $\rho_1 \sim_\beta \rho_2$.

We know by hypothesis that $v :: l :: os_1 \sim_{\beta, st_1, st_2} v' :: l' :: os_2$ and hence, by lemma 34, $os_1 \sim_{\beta, st'_1, st'_2} os_2$. We then make a case analysis using lemma 34.

- either $k'_1 = k'_2$, $k'_1 \leq k_{\text{obs}}$ and $l \sim_\beta l'$. In this case we conclude operand stack indistinguishability $os_1 \sim_{\beta, \mathsf{lift}_{k'_1} st, \mathsf{lift}_{k'_2} st'} os_2$ by lemma 33.
- or $k'_1 \not\leq k_{\text{obs}}$ and $k'_2 \not\leq k_{\text{obs}}$. We can hence claim that $\mathsf{high}(os_1, \mathsf{lift}_{k'_1} st)$ and $\mathsf{high}(os_2, \mathsf{lift}_{k'_2} st')$, and finally conclude about operand stack indistinguishability.

To check indistinguishability of heaps we remark that h_1 is only updated in location l and field f, and similarly h_2 is only updated in location l' and field f. Hence, if $\mathsf{ft}(f) \not\leq k_{\text{obs}}$ heap indistinguishability still holds. If $\mathsf{ft}(f) \leq k_{\text{obs}}$ we have $k_1 \leq k_{\text{obs}}$ and $k_2 \leq k_{\text{obs}}$ since by typability, the constraint $k_1 \leq \mathsf{ft}(f)$ and $k_2 \leq \mathsf{ft}(f)$ hold. Hence the lemma 34 gives us $v \sim_\beta v'$ and $l \sim_\beta l'$. This allows us to conclude about heap indistinguishability.

- One execution is normal ($\tau_2 = \emptyset$) and there is a null pointer exception in the other execution ($\tau_1 = \mathbf{np}$) , there is a handler t in m, and $s_1' = \langle t, \rho_1, l'' :: \epsilon, h_1 \oplus\{l'' \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$ and $s_2' = \langle i_1 + 1, \rho_2, os_2, h_2 \oplus\{o \mapsto h_2(l') \oplus\{f \mapsto v'\}\}\rangle$. By typability, $st_1 = k_1 :: k_1' :: st$, $st_2 = k_2 :: k_2' :: st'$, $st_2' = \mathsf{lift}_{k_2'} st'$ and $st_1' = k_1' \sqcup se(i) :: \epsilon$.
  By hypothesis and semantics (variables do not change) we have indistinguishability of variables $\rho_1 \sim_\beta \rho_2$.
  Because of $v :: null :: os_1 \sim_{k_1::k_1'::st, k_2::k_2'::st', \beta} v' :: l' :: os_2$ and lemma 34 we have necessarily $k_1' \not\leq k_{\text{obs}}$ and $k_2' \not\leq k_{\text{obs}}$. We can hence claim that $\mathsf{high}(l'' :: \epsilon, k_1' \sqcup se(i) :: \epsilon)$ and $\mathsf{high}(os_2, \mathsf{lift}_{k_2'} st_2')$, and finally conclude about operand stack indistinguishability $l'' :: \epsilon \sim_{\beta, k_1' \sqcup se(i)::\epsilon, \mathsf{lift}_{k_2'} st'} os_2$.
  To check indistinguishability of

$$h_1 \oplus\{l'' \mapsto \mathsf{default}_{\mathbf{np}}\} \sim_\beta h_2 \oplus\{l' \mapsto h_2(l') \oplus\{f \mapsto v'\}\}$$

  we first invoke lemma 41 (l'' is a fresh location for h_1) to establish $h_1 \oplus\{l'' \mapsto \mathsf{default}_{\mathbf{np}}\} \sim_\beta h_2$. By typability, $k_1' \leq \mathsf{ft}(f)$ so $\mathsf{ft}(f) \not\leq k_{\text{obs}}$. This allows to conclude about heap indistinguishability.
- One execution is normal ($\tau_2 = \emptyset$) and there is a null pointer exception ($\tau_1 = \mathbf{np}$), but no handler for it in m, and $s_1' = \langle\langle l''\rangle, h_1 \oplus\{l'' \mapsto \mathbf{default}_{\mathbf{np}}\}\rangle$; and s_2' is equal to $\langle i_1{+}1, \rho_2, os_2, h_2 \oplus\{o \mapsto h_2(l') \oplus\{f \mapsto v'\}\}\rangle$.
  By typability, $st_1 = k_1 :: k_1' :: st$, $st_2 = k_2 :: k_2' :: st'$, $st_2' = \mathsf{lift}_{k_2'} st'$.
  We take $\beta' = \beta$.
  Heap indistinguishability holds for the same reason as in the previous case.
  Because of $v :: null :: os_1 \sim_{k_1::k_1'::st, k_2::k_2'::st', \beta} v' :: l' :: os_2$ and lemma 34 we have necessarily $k_1' \not\leq k_{\text{obs}}$ and $k_2' \not\leq k_{\text{obs}}$. By typability, we have the constraint $k_1' \leq \boldsymbol{k_r}[\mathbf{np}]$. Hence $\boldsymbol{k_r}[\mathbf{np}] \not\leq k_{\text{obs}}$ and $\mathsf{highResult}_{\boldsymbol{k_r}}(\langle l''\rangle, h_1 \oplus\{l'' \mapsto \mathbf{default}_{\mathbf{np}}\})$ hold.
- There are two null pointer exceptions due to putfield and there is no handler for them in m, s_1' is equal to $\langle\langle l_1\rangle, h_1 \oplus\{l_1 \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$; and s_2' is equal to $\langle\langle l_2\rangle, h_2 \oplus\{l_2 \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$. By typability, $st_1 = k_1 :: k_1' :: st$ and $st_2 = k_2 :: k_2' :: st'$.
  We take $\beta' = \beta \oplus\{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence lemma 40 give us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus\{l_1 \mapsto \mathsf{default}_{\mathbf{np}}\} \sim_{\beta'} h_2 \oplus\{l_2 \mapsto \mathsf{default}_{\mathbf{np}}\}$.
  We then make case analysis on $\boldsymbol{k_r}[\mathbf{np}]$.
  - if $\boldsymbol{k_r}[\mathbf{np}] \not\leq k_{\text{obs}}$, we have easily $(\langle l_1\rangle, h_1') \sim_{\boldsymbol{k_r}, \beta'} (\langle l_2\rangle, h_2')$.
  - if $\boldsymbol{k_r}[\mathbf{np}] \leq k_{\text{obs}}$, we have $(\langle l_1\rangle, h_1') \sim_{\boldsymbol{k_r}, \beta'} (\langle l_2\rangle, h_2')$ thanks to $l_1 \sim_{\beta'} l_2$ (since $\beta'(l_1) = l_2$).
- There are two null pointer exceptions and there is a handler for null pointer exception and program point i, namely t, s_1' is equal to $\langle t, \rho_1, l_1 :: \epsilon, h_1 \oplus\{l_1 \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$ and $s_2' = \langle t, \rho_2, l_2 :: \epsilon, h_2 \oplus\{l_2 \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$, $st_1 = k_1 :: k_1' :: st$, $st_2 = k_2 :: k_2' :: st'$, $st_1' = k_1' \sqcup se(i) :: \epsilon$, and $st_2' = k_2' \sqcup se(i) :: \epsilon$.
  We take $\beta' = \beta \oplus\{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence lemma 40 give us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus\{l_1 \mapsto \mathsf{default}_{\mathbf{np}}\} \sim_{\beta'} h_2 \oplus\{l_2 \mapsto \mathsf{default}_{\mathbf{np}}\}$.
  Local variables are not modified, so by lemma 44, local variable indistinguishability holds.
  We finally have to prove operand stack indistinguishability $l_1 :: \epsilon \sim_{k_1' \sqcup se(i)::\epsilon, k_2' \sqcup se(i)::\epsilon, \beta'} l_2 :: \epsilon$. We make a case analysis on hypothesis $v :: null :: os_1 \sim_{\beta, st_1, st_2} v' :: null :: os_2$ with the help of lemma 34:
  - either $k_1' = k_2'$ and $k_1' \leq k_{\text{obs}}$. In this case, since $l_1 \sim_{\beta'} l_2$ and $k_1' \sqcup se(i) = k_2' \sqcup se(i)$ we are done.
  - or $k_1' \not\leq k_{\text{obs}}$ and $k_2' \not\leq k_{\text{obs}}$ and operand stack indistinguishability trivially holds.

Case: $P_m[i] = \mathsf{getfield}\ f$

By semantics, $s_1 = \langle i_1, \rho_1, l :: os_1, h_1\rangle$ and $s_2 = \langle i_2, \rho_2, l' :: os_2, h_2\rangle$ and there are several options to consider successors of s_1 and s_2,

- There are no exceptions and $s_1' = \langle i_1 + 1, \rho_1, v :: os_1, h_1\rangle$ and $s_2' = \langle i_1 + 1, \rho_2, v' :: os_2, h_2\rangle$. By typability, $st_1 = k_1 :: st$, $st_2 = k_2 :: st'$, $st_1' = \mathrm{lift}_{k_1}((\mathsf{ft}(f) \sqcup se(i)) :: st)$ and $st_2' = \mathrm{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st')$.

We take $\beta' = \beta$. Local variables and heaps are not modified so indistinguishability still hold for them. By hypothesis, $v :: os_1 \sim_{k_1::st,k_2::st',\beta} v' :: os_2$ so $os_1 \sim_{st,st',\beta} os_2$ holds too. We then make a case analysis using lemma 34.

- either $k_1 = k_2$ and $k_1 \leq k_{\text{obs}}$. In this case we conclude operand stack indistinguishability $os_1 \sim_{\mathsf{lift}_{k_1} st,\mathsf{lift}_{k_2} st',\beta} os_2$ by lemma 33,
- or $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. We can hence claim that $\mathsf{high}(os_1, \mathsf{lift}_{k_1} st)$ and $\mathsf{high}(os_2, \mathsf{lift}_{k_2} st')$, and finally conclude about operand stack indistinguishability.

– One execution is normal and there is a null pointer exception in the other execution, there is a handler t in m, and $s'_1 = \langle t, \rho_1, l'' :: \epsilon, h_1 \oplus \{l'' \mapsto \mathbf{default_{np}}\}\rangle$ and s'_2 is equal to $\langle i_1 + 1, \rho_2, v' :: os_2, h_2\rangle$. By typability, $st_1 = k_1 :: st$, $st_2 = k_2 :: st'$, $st'_1 = k_1 \sqcup se(i) :: \epsilon$ and $st'_2 = \mathrm{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st')$.
We take $\beta' = \beta$. Local variables are not modified so indistinguishability still hold for them.
By hypothesis $null :: os_1 \sim_{k_1::st,k_2::st',\beta} l' :: os_2$, we have necessarily $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. It follows that $\mathsf{high}(l'' :: \epsilon, k_1 \sqcup se(i) :: \epsilon)$ and $\mathsf{high}(v' :: os_2, \mathrm{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st'))$.
Heap indistinguishability holds by lemma 41.

– One execution is normal and there is a null pointer exception, but no handler for it in m, $s'_1 = \langle\langle l''\rangle, h_1 \oplus \{l'' \mapsto \mathsf{default_{np}}\}\rangle$ and $s'_2 = \langle i_1 + 1, \rho_2, v :: os_2, h_2\rangle$. By typability, $st_1 = k_1 :: st$, $st_2 = k_2 :: st'$, and $st'_2 = \mathrm{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st')$.
Heap indistinguishability holds by lemma 41.
By hypothesis $null :: os_1 \sim_{k_1::st,k_2::st',\beta} l' :: os_2$, we have necessarily $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. By typability, $k_1 \leq \boldsymbol{k_r}[\mathbf{np}]$. Hence $\boldsymbol{k_r}[\mathbf{np}] \not\leq k_{\text{obs}}$ and $\mathsf{highResult}_{\boldsymbol{k_r}}(\langle l''\rangle, h_1 \oplus \{l'' \mapsto \mathsf{default_{np}}\})$ holds.

– Two null pointer exceptions and no handler for them in i, $s'_1 = \langle\langle l_1\rangle, h_1 \oplus \{l_1 \mapsto \mathsf{default_{np}}\}\rangle$; and $s'_2 = \langle\langle l_2\rangle, h_2 \oplus \{l_2 \mapsto \mathsf{default_{np}}\}\rangle$. By typability, $st_1 = k_1 :: st$ and $st_2 = k_2 :: st'$.
We take $\beta' = \beta \oplus \{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence lemma 40 give us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus \{l_1 \mapsto \mathsf{default_{np}}\} \sim_{\beta'} h_2 \oplus \{l_2 \mapsto \mathsf{default_{np}}\}$.
We then make case analysis on $\boldsymbol{k_r}[\mathbf{np}]$.

- if $\boldsymbol{k_r}[\mathbf{np}] \not\leq k_{\text{obs}}$, we have easily $(\langle l_1\rangle, h'_1) \sim_{\boldsymbol{k_r},\beta'} (\langle l_2\rangle, h'_2)$.
- if $\boldsymbol{k_r}[\mathbf{np}] \leq k_{\text{obs}}$, we have $(\langle l_1\rangle, h'_1) \sim_{\boldsymbol{k_r},\beta'} (\langle l_2\rangle, h'_2)$ thanks to $l_1 \sim_{\beta'} l_2$ (since $\beta'(l_1) = l_2$).

– Two exceptions with handler: $s'_1 = \langle t, \rho_1, l_1 :: \epsilon, h_1 \oplus \{l_1 \mapsto \mathsf{default_{np}}\}\rangle$ and $s'_2 = \langle t, \rho_2, l_2 :: \epsilon, h_2 \oplus \{l_2 \mapsto \mathsf{default_{np}}\}\rangle$.
By typability, $st_1 = k_1 :: st$, $st_2 = k_2 :: st'$, $st'_1 = k_1 :: \epsilon$ and $st'_2 = k_2 :: \epsilon$.
We take $\beta' = \beta \oplus \{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence lemma 40 give us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus \{l_1 \mapsto \mathsf{default_{np}}\} \sim_{\beta'} h_2 \oplus \{l_2 \mapsto \mathsf{default_{np}}\}$.
Local variables are not modified, so by lemma 44, local variable indistinguishability holds.
We finally have to prove operand stack indistinguishability $l_1 :: \epsilon \sim_{k_1::\epsilon,k_2::\epsilon,\beta'} l_2 :: \epsilon$. We make a case analysis on hypothesis $null :: os_1 \sim_{\beta,st_1,st_2} null :: os_2$ with the help of lemma 34:

- either $k_1 = k_2$ and $k_1 \leq k_{\text{obs}}$. In this case, since $l_1 \sim_{\beta'} l_2$ and $k_1 = k_2$ so we are done.
- or $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$ and operand stack indistinguishability trivially holds.

Case: $P_m[i] = \mathsf{throw}$

By semantics, $s_1 = \langle i, \rho_1, v_1 :: os_1, h_1\rangle$ and $s_2 = \langle i, \rho_2, v_2 :: os_2, h_2\rangle$ with v_1 and v_2 in $\mathcal{L} \cup \{null\}$. By typability, $st_1 = k_1 :: st''_1$ and $st_2 = k_2 :: st''_2$. There is a successor state in the current method execution if and only if the thrown exception is caught. In this case st'_j ($j \in \{1, 2\}$) is of the form $st'_j = k_j \sqcup se(i) :: \epsilon$.
By hypothesis, $v_1 :: os_1 \sim_{k_1::st''_1,k_2::st''_2,\beta} v_2 :: os_2$. We then make a case analysis using lemma 34:

– If $k_1 = k_2$, $k_1 \leq k_{\text{obs}}$ and $v_1 \sim_\beta v_2$, there are two cases to consider.

- $v_1 = v_2 = null$: this case is exactly the same as for $\mathsf{getfied}$ instruction when both operand stacks of s_1 and s_2 have a null pointer on top of there stacks.
- $v_1 = l_1$, $v_2 = l_2$ and $l_1 \sim_\beta l_2$: since $h_1 \sim_\beta h_2$ we deduce that $h_1(l_1)$ and $h_2(l_2)$ have the same class. Hence we are either in case 1 of the lemma statement (when the exception is caught) or in case 3 (when the exception is uncaught). In both cases we choose $\beta' = \beta$.
  * if the exception is caught : by semantics, $s'_1 = \langle t, \rho_1, l :: \epsilon, h_1\rangle$ and $s'_2 = \langle t, \rho_2, l :: \epsilon, h_2\rangle$. Heaps and local variables are not modified so there is only something to prove for operand stacks, that is: $l_1 :: \epsilon \sim_{k_1 \sqcup se(i)::\epsilon,k_2 \sqcup se(i)::\epsilon,\beta} l_2 :: \epsilon$. This is true since $k_1 \sqcup se(i) = k_2 \sqcup se(i)$and $v_1 \sim_\beta v_2$.

* if the exception is uncaught, we must establish $(\langle l_1\rangle, h_1) \sim_{\boldsymbol{k_r},\beta} (\langle l_2\rangle, h_2)$. The current hypotheses contains $h_1 \sim_\beta h_2$ and $l_1 \sim_\beta l_2$ so output indistinguishability holds easily.

– If $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. We choose $\beta' = \beta$. In each execution thrown exception is either caught or uncaught. We now examine the different cases (symmetric cases are skipped).

- Both exceptions are caught: each s'_j ($j \in \{1,2\}$) is then of the form $s'_j = \langle t_j, \rho_j, l'_j :: \epsilon, h'_j\rangle$ with $h'_j = h_j$ (if $v_j \neq null$) or $h'_j = h_j \oplus\{l'_j \mapsto \textbf{default}_{\textbf{np}}\}$ (if $v_j \neq null$ and $l'_j = \text{fresh}(h_j)$).
  Local variables are not modified so indistinguishability still hold for them.
  Each st'_j is of the form $k_j \sqcup se(i) :: \epsilon$ and $k_j \not\leq k_{\text{obs}}$ so operand stack indistinguishability since we have $\mathsf{high}(l'_1 :: \epsilon, k_1 \sqcup se(i) :: \epsilon)$ for each j.
  Concerning heaps, $h'_1 \sim_\beta h'_2$ holds even if one of the h'_j is of the form $h_j \oplus\{l'_j \mapsto \mathsf{default}_{\mathbf{np}}\}$ thanks to lemma 37.
- exception is caught in the first execution but not in the second: s'_1 is then of the form $s'_1 = \langle t, \rho_1, l'_1 :: \epsilon, h'_1\rangle$ and s'_2 of the form $(\langle l'_2\rangle, h'_2)$ with $h'_j = h_j$ (if $v_j \neq null$) or $h'_j = h_j \oplus\{l'_j \mapsto \mathsf{default}_{\mathbf{np}}\}$ (if $v_j \neq null$ and $l'_j = \text{fresh}(h_j)$), $j \in \{1,2\}$.
  $h'_1 \sim_\beta h'_2$ holds even if one of the h'_j is of the form $h_j \oplus\{l'_j \mapsto \mathsf{default}_{\mathbf{np}}\}$ thanks to lemma 37.
  We finally have to establish $\mathsf{highResult}_{\boldsymbol{k_r}}(l'_2, h'_2)$. By hypothesis, $\tau_2 = \mathsf{class}(h'_2(l'_2))$, $k_2 \leq \boldsymbol{k_r}[\tau_2]$ and $k_2 \not\leq k_{\text{obs}}$ so we are done.
- exception is uncaught in both executions: each s'_j ($j \in \{1,2\}$) is then of the form $(\langle l'_j\rangle, h'_j)$ with $h'_j = h_j$ (if $v_j \neq null$) or $h'_j = h_j \oplus\{l'_j \mapsto \textbf{default}_{\textbf{np}}\}$ (if $v_j \neq null$ and $l'_j = \text{fresh}(h_j)$).
  We must prove that $(\langle l'_1\rangle, h'_1) \sim_{\beta',\boldsymbol{k'_r}} (\langle l'_2\rangle, h'_2)$. It is sufficient to prove

$$h'_1 \sim_\beta h'_2 \qquad \boldsymbol{k_r}[\mathsf{class}(h'_1(l'_1))] \not\leq k_{\text{obs}} \qquad \boldsymbol{k_r}[\mathsf{class}(h'_2(l'_2))] \not\leq k_{\text{obs}}$$

  $h'_1 \sim_\beta h'_2$ holds even if one of the h'_j is of the form $h_j \oplus\{l'_j \mapsto \mathsf{default}_{\mathbf{np}}\}$ thanks to lemma 37.
  $\boldsymbol{k_r}[\mathsf{class}(h'_j(l'_j))]$ holds for each j since, by semantics $\tau_j = \mathsf{class}(h'_j(l'_j))$ and by typability, $k_j \leq \boldsymbol{k_r}[\tau_j]$ and $k_j \not\leq k_{\text{obs}}$.

Case: $P_m[i] = \mathsf{invokevirtual}\ m_{\text{ID}}$. By semantics, each s_j is of the form $\langle i, \rho_j, os_j :: v_j :: os'_j, h_j\rangle$ with $v_j \in \mathcal{L} \cup \{null\}$. By typability, each st_j is of the form $st''_j :: k_j :: st'''_j$ with $\mathsf{length}(os_j) = \mathsf{length}(st_j)$.
By hypothesis,

$$os_1 :: v_1 :: os'_1 \sim_{st''_1::k_1::st'''_1, st''_2::k_2::st'''_2, \beta} os_2 :: v_2 :: os'_2$$

We then make a case analysis using lemma 34:

– If $k_1 = k_2$, $k_1 \leq k_{\text{obs}}$ and $v_1 \sim_\beta v_2$, there are two cases to consider.

- $v_1 = v_2 = null$: a null pointer exception is thrown. We must then examine if this exception is caught or not.
  * If the exception is caught, $s'_1 = \langle t, \rho_1, l_1 :: \epsilon, h_1 \oplus\{l_1 \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$ and $s'_2 = \langle t, \rho_2, l_2 :: \epsilon, h_2 \oplus\{l_2 \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$. By typability, $st'_1 = st'_2 = (k_1 \sqcup \boldsymbol{k'_r}[\mathbf{np}]) :: \varepsilon$.
    We take $\beta' = \beta \oplus\{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence lemma 40 give us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus\{l_1 \mapsto \mathsf{default}_{\mathbf{np}}\} \sim_{\beta'} h_2 \oplus\{l_2 \mapsto \mathsf{default}_{\mathbf{np}}\}$.
    Local variables are not modified, so by lemma 44, local variable indistinguishability holds.
    Finally, we prove operand stack indistinguishability $l_1 :: \epsilon \sim_{k_1 \sqcup \boldsymbol{k'_r}[\mathbf{np}]::\epsilon, k_2 \sqcup \boldsymbol{k'_r}[\mathbf{np}]::\epsilon, \beta'} l_2 :: \epsilon$ since $l_1 \sim_{\beta'} l_2$ and $k_1 = k_2$.
  * If the exception is uncaught, $s'_1 = (\langle l_1\rangle, h_1)$ and $s'_2 = (\langle l_2\rangle, h_2)$.
    We take $\beta' = \beta \oplus\{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence lemma 40 give us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus\{l_1 \mapsto \mathsf{default}_{\mathbf{np}}\} \sim_{\beta'} h_2 \oplus\{l_2 \mapsto \mathsf{default}_{\mathbf{np}}\}$.
    We conclude on output indistinguishability $(\langle l_1\rangle, h_1) \sim_{\boldsymbol{k_r},\beta'} (\langle l_2\rangle, h_2)$ using previous heap indistinguishability and the fact $l_1 \sim_{\beta'} l_2$.
- $v_1 = l_1$, $v_2 = l_2$ and $l_1 \sim_\beta l_2$: since $h_1 \sim_\beta h_2$ we deduce that $h_1(l_1)$ and $h_2(l_2)$ have the same class. We deduce that the same method m' is called in both executions. As in the proof of the previous lemma we then invoke non-interference at some order k, $k < n$ to establish that outputs of each executions of m' are indistinguishable for some β' such that $\beta \subseteq \beta'$:

$$(r_1, h'_1) \sim_{\beta',\boldsymbol{k'_r}} (r_2, h'_2) \tag{2}$$

There are then two cases to consider for each executions of the called method (normal termination of the called method or termination by an exception). We examine now these different cases, skipping symmetric cases. Heaps indistinguishable is not mentioned since it follows from (2).

* If both executions terminate normally $r_1 = v_1 \in \mathcal{V}$ and $r_2 = v_2 \in \mathcal{V}$. (2) gives $\boldsymbol{k'_r}[n] \leq k_{\text{obs}} \Rightarrow v_1 \sim_{\beta'} v_2$. Indistinguishability of local variables is obtained using lemma 44. Operand stack indistinguishability $v_1 :: os'_1 \sim_{(\boldsymbol{k'_r}[n] \sqcup se(i))::st_1, (\boldsymbol{k'_r}[n] \sqcup se(i))::st_2, \beta'} v_2 :: os'_2$ is finally obtained thanks to $\boldsymbol{k'_r}[n] \leq k_{\text{obs}} \Rightarrow v_1 \sim_{\beta'} v_2$ and $os'_1 \sim_{st_1, st_2, \beta'} os'_2$ (obtained using lemma 45).
* If both executions terminate with an exception e_1 and e_2, $r_1 = l'_1 \in \mathcal{L}$ and $r_2 = l'_2 \in \mathcal{L}$. (2) gives us two more cases:
  In first case $\boldsymbol{k'_r}[e_1] \not\leq k_{\text{obs}}$ and $\boldsymbol{k'_r}[e_2] \not\leq k_{\text{obs}}$. If both exceptions are caught in m, $l'_1 :: \epsilon \sim_{(k_1 \sqcup \boldsymbol{k'_r}[e_1])::\varepsilon, (k_2 \sqcup \boldsymbol{k'_r}[e_2])::\varepsilon, \beta'} l'_2 :: \epsilon$ holds since stacks are high. If e_1 is caught and e_2 is uncaught, $\mathsf{highResult}_{\boldsymbol{k_r}}(l'_2, h'_2)$ holds thanks to typability constraint $k_2 \sqcup se(i) \sqcup \boldsymbol{k'_r}[e_2] \leq \boldsymbol{k_r}[e_2]$. If both exceptions are uncaught in m, $(\langle l'_1 \rangle, h'_1) \sim_{\boldsymbol{k_r}, \beta'} (\langle l'_2 \rangle, h'_2)$ holds thanks to typability constraints $k_1 \sqcup se(i) \sqcup \boldsymbol{k'_r}[e_1] \leq \boldsymbol{k_r}[e_1]$ and $k_2 \sqcup se(i) \sqcup \boldsymbol{k'_r}[e_2] \leq \boldsymbol{k_r}[e_2]$.
  In second case $e_1 = e_2$ and $l'_1 \sim_{\beta'} l'_2$. If e_1 is caught, $l'_1 :: \epsilon \sim_{(k_1 \sqcup \boldsymbol{k'_r}[e_1])::\varepsilon, (k_2 \sqcup \boldsymbol{k'_r}[e_2])::\varepsilon, \beta'} l'_2 :: \epsilon$ holds since $(k_1 \sqcup \boldsymbol{k'_r}[e_1]) = (k_2 \sqcup \boldsymbol{k'_r}[e_2])$ and $l'_1 \sim_{\beta'} l'_2$. If e_1 is uncaught, $(\langle l'_1 \rangle, h'_1) \sim_{\boldsymbol{k_r}, \beta'} (\langle l'_2 \rangle, h'_2)$ holds since $l'_1 \sim_{\beta'} l'_2$.

– $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. We choose $\beta' = \beta$. Each h'_j is either of the form $h_j \oplus \{l'_j \mapsto \mathsf{default_{np}}\}$ (if the virtual call is done on a null pointer) or is equal to the final heap obtained after execution of the called method m'_j. Since methods are supposed side-effect safe we know that $h_j \preceq h'_j$. In every cases $h'_1 \sim_{\beta'} h'_2$ holds thanks to lemma 37 or lemma 39 (side effect levels are high by typability).
  We then make a case analysis according to the nature (final or not) of each execution. Heaps indistinguishability is no more mentioned since already proved.
  - if both executions go on, by typability all elements in st'_1 (respectively st'_2) are greater than k_1 (respectively k_2) and hence are high. Operand stack indistinguishability follows easily. Local variables indistinguishability is immediate since local variables are not modified.
  - if one execution goes on but the other terminates with an uncaught exception e we must prove that $\boldsymbol{k_r}[e] \not\leq k_{\text{obs}}$. This is done using typability constraint $k \sqcup se(i) \sqcup \boldsymbol{k'_r}[e] \leq \boldsymbol{k_r}[e]$ with k equals to k_1 or k_2.
  - If both executions terminate with some uncaught exception e_1 and e_2, output indistinguishability holds using typability constraint $k_1 \sqcup se(i) \sqcup \boldsymbol{k'_r}[e_1] \leq \boldsymbol{k_r}[e_1]$ and $k_2 \sqcup se(i) \sqcup \boldsymbol{k'_r}[e_2] \leq \boldsymbol{k_r}[e_2]$.

Lemma 30 (Non-interference induction step). *Let n an integer such that all method in P are non-interferent at all order k, $k < n$.*

Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and $\langle i_0, \rho_0, os_0, h_0 \rangle, \langle i'_0, \rho'_0, os'_0, h'_0 \rangle \in \text{State}_{\mathcal{G}}$ two $\text{JVM}_{\mathcal{G}}$ statessuch that $\langle i_0, \rho_0, os_0, h_0 \rangle \sim_{S_{i_0}, S_{i'_0}, \beta} \langle i'_0, \rho'_0, os'_0, h'_0 \rangle$, and $i_0 = i'_0$. Suppose we have a derivation

$$\langle i_0, \rho_0, os_0, h_0 \rangle \overset{(n_0)}{\rightsquigarrow}_{m, \tau_0} \cdots \langle i_k, \rho_k, os_k, h_k \rangle \overset{(n_k)}{\rightsquigarrow}_{m, \tau_k} (r, h)$$

with $n_0 + \cdots + n_k \leq n$ and suppose this derivation is typable with respect to S. Suppose we have a derivation

$$\langle i'_0, \rho'_0, os'_0, h'_0 \rangle \overset{(n'_0)}{\rightsquigarrow}_{m, \tau'_0} \cdots \langle i'_k, \rho'_k, os'_k, h'_k \rangle \overset{(n'_k)}{\rightsquigarrow}_{m, \tau'_k} (r', h')$$

with $n'_0 + \cdots + n'_k \leq n$ and suppose this derivation is typable with respect to S. Then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that

$$(r, h) \sim_{\boldsymbol{k_r}, \beta'} (r', h') \quad \textit{and} \quad \beta \subseteq \beta'$$

Proof. By induction on $\max(k, k')$. Suppose the statement is true for derivation of length strictly lower than $\max(k, k')$ and prove it for the current lengths.

There are four cases:

1. $k = k' = 0$: the case 3 of lemma 29 allows us a direct conclusion.

2. $k > 0$ and $k' = 0$: we are in the case 2 of lemma 29 so there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that

$$h_1 \sim_{\beta'} h', \quad \mathsf{highResult}_{\boldsymbol{k_r}}(r', h') \quad \text{and} \quad \beta \subseteq \beta'$$

Thanks to case 2 of lemma 28 and sub-typing lemma 5 we have

$$\mathsf{high}(os_1, S_{i_1}) \quad \text{and} \quad se \text{ is high in } \mathsf{region}(i_0, \tau_1)$$

where τ_1 verifies $i_0 \mapsto^{\tau_1} i_1$. SOAP2 gives $i_1 \in \mathsf{region}(i_0, \tau_1)$ or $i_1 = \mathsf{jun}(i_0, \tau_1)$ but $\mathsf{jun}(i_0, \tau_1)$ is undefined thanks to SOAP3. We can hence apply lemma 27 to conclude that

$$\mathsf{highResult}_{\boldsymbol{k_r}}(r', h') \quad \text{and} \quad h' \sim_{\beta} h'_1$$

Using the other hypotheses

$$h_1 \sim_{\beta'} h'_1, \quad h_1 \sim_{\beta'} h', \quad \mathsf{highResult}_{\boldsymbol{k_r}}(r', h')$$

we can easily conclude.

3. $k = 0$ and $k' > 0$: symmetric version of the previous case.
4. $k > 0$ and $k' > 0$: We make two cases:
Case 1: $i_1 = i'_1$. The induction hypothesis directly applies.
Case 2: $i_1 \neq i'_1$. Let call st_1, st'_1 the stack types such that

$$i_o \vdash^{\tau_0} S_{i_0} \Rightarrow st_1, \quad st_1 \sqsubseteq S_{i_1}, \quad i'_o \vdash^{\tau'_0} S_{i'_0} \Rightarrow st'_1, \quad st'_1 \sqsubseteq S_{i'_1}$$

The case 1 of lemma 28 can be invoked to deduce (with the help of lemma 5)

$$\begin{array}{rl} \mathsf{high}(os_1, st_1) & \text{and} \quad se \text{ is high in } \mathsf{region}(i_0, \tau) \\ \mathsf{high}(os'_1, st'_1) & \text{and} \quad se \text{ is high in } \mathsf{region}(i_0, \tau') \end{array}$$

where τ, τ' verify $i_0 \mapsto^{\tau} i_1$ and $i'_0 \mapsto^{\tau'} i'_1$. Thanks to lemma 5 we have

$$\mathsf{high}(os_1, S_{i_1}) \quad \text{and} \quad \mathsf{high}(os'_1, S'_{i_1})$$

We are furthermore in case 1 of lemma 29 so there exists β', $\beta \subseteq \beta'$ such that

$$\langle i_1, \rho_1, os_1, h_1 \rangle \sim_{st_1, st'_1, \beta'} \langle i'_1, \rho'_1, os'_1, h'_1 \rangle$$

Using lemma 7 two times we have

$$\langle i_1, \rho_1, os_1, h_1 \rangle \sim_{S_{i_1}, S_{i'_1}, \beta'} \langle i'_1, \rho'_1, os'_1, h'_1 \rangle$$

This allows us to invoke lemma 25 which gives us two cases:

(a) There exists j, j' with $1 \leq j \leq k$ and $1 \leq j' \leq k'$ such that $i_j = i'_{j'}$ and $\langle i_j, \rho_j, os_j, h_j \rangle \sim_{S_{i_j}, S_{i'_{j'}}, \beta}$ $\langle i'_{j'}, \rho_{j'}, os_{j'}, h_{j'} \rangle$. We can use the induction hypothesis on $\langle i_j, \rho_j, os_j, h_j \rangle \overset{(n_j)}{\rightsquigarrow}_{m, \tau_j} (r, h)$ and $\langle i_{j'}, \rho_{j'}, os_{j'}, h_{j'} \rangle \overset{(n_{j'})}{\rightsquigarrow}_{m, \tau'_j} (r', h')$ to conclude.

(b) $(r, h) \sim_{\boldsymbol{k_r}, \beta'} (r', h')$ and we can directly conclude.

Proof (of Theorem 4). We show by induction on an integer n that all method in P are non-interferent at order n. We use an induction scheme of the form

$$(\forall n,\ (\forall k, k < n \Rightarrow \mathcal{P}(k)) \ \Rightarrow\ \mathcal{P}(n)) \quad \Longrightarrow \quad \forall n,\ \mathcal{P}(n)$$

The proof is then a direct application of lemma 30.

7 Related work

We refer to the survey article of Sabelfeld and Myers [30] for a more complete account of recent developments in language-based security, and only focus on related work that deals with low-level languages, or develop ideas that are relevant to consider in future work.

For convenience, we separate related work between works that deal with typed assembly languages, and higher-order low-level languages and finally with the JVM and Java. Then, we focus on issues of concurrency and information release, which are not considered in the present work.

7.1 Typed assembly languages

The idea of typing low-level programs and ensuring that compilation preserves typing is not original to information flow, and has been investigated in connection with type-directed compilation. Morrisett, Walker, Crary and Glew [24] develop a typed assembly language (TAL) based on a conventional RISC assembly language, and show that typable programs of System F can be compiled into typable TAL programs.

The study of non-interference for typed assembly languages has been initiated by Medel, Bonelli, and Compagnoni [12], who developed a sound information flow type system for a simple assembly language called SIFTAL. A specificity of SIFTAL is to introduce pseudo-instructions that are used to enforce structured control flow using a stack of continuations; more concretely, the pseudo-instructions are used to push or retrieve linear continuations from the continuation stack. Unlike the stack of call frames that is used in the JVM to handle method calls, the stack of continuations is used for control flow within the body of a method. The use of pseudo-instructions allows to formulate global constraints in the type system, and thus to guarantee non-interference. More recent work by the same authors [**?**] and by Yu and Islam [35] avoids the use of pseudo-instructions. In addition, Yu and Islam consider a richer assembly language and prove type-preserving compilation for an impreative language with procedures.

7.2 Higher-order low-level languages

Zdancewic and Myers [36] develop a sound information flow type system for a CPS calculus that uses linear continuations and prove type-preservation for a linear CPS translation from an imperative higher-order language inspired from SLAM [19] to their CPS language, providing thus one early type-preservation result for information flow. The use of linear continuations in the CPS translation is essential to guarantee type-preserving compilation.

In a similar line of work, Honda and Yoshida [20] develop a sound information flow type system for the π-calculus and prove type-preserving compilation for the Dependency Core Calculus [1] and for an imperative language inspired from Volpano and Smith [34]. Furthermore, they derive soundness of the source type systems from the soundness of the type system for the π-calculus. As in the work of Zdancewic and Myers, linearity is used crucially to ensure that the compilation is type-preserving.

7.3 JVM

Lanet *et al.* [11] provide an early study of information flow the JVM. Their method consists in specifying in the SMV model checker an abstract transition semantics of the JVM that manipulates security levels, and that can be used to verify that an invariant that captures the absence of illicit flows is maintained throughout the (abstract) program execution. Their method is directed towards smart card applications, and thus only covers a sequential fragment of the JVM. While their method has been used successfully to detect information leaks in a case study involving multi-application smartcards, it is not supported by any soundness result. In a series of papers initiating with [10], Bernardeschi and co-workers also propose to use abstract interpretation and model-checking techniques to verify secure information.

In a predecessor to this work, Barthe, Basu and Rezk [5] propose a sound information flow type system for a simple assembly language that closely resembles the $\mathrm{JVM}_{\mathcal{I}}$ fragment of this paper, and show type-preserving

compilation for the imperative language and type system of [34]. Later, Barthe and Rezk [8] extend this work to a language with objects and a simplified treatment of exceptions, and Barthe, Naumann and Rezk [7] show type-preserving compilation for a Java-like language with objects and a simplified treatment of exceptions.

Genaim and Spoto [17] have shown how to represent information flow for Java bytecode through boolean functions; the representation allows checking via binary decision diagrams. Their analysis is fully automatic and does not require that methods are annotated with security signatures, but it is less efficient than type checking.

7.4 Java

Jif is an extension of Java with information flow types developed by Myers and co-workers. Jif builds upon the decentralized label model and offers a flexible and expressive framework to define information flow policies. The rich set of features supported by Jif has proved useful in realistic case studies such as an implementation of mental poker [3], but makes it difficult to prove that the information flow type system is sound.

Banerjee and Naumann [4] develop a sound information flow type system for a fragment of Java with objects and methods. The type system is simpler than Jif:

- due to the absence of certain language features such as exceptions. For example, their return signatures is reduced to a single level, since abnormal termination is not considered.
- by design. For example, there is no mechanism for information release.

The type system has been formally verified in PVS [26], and [33] present a type inference algorithm that dispenses users of writing all security annotations.

More recently, Hammer, Krinke and Snelting [**?**] have developed an information flow analysis based on control dependence regions; they use path conditions to achieve precision in their analysis, and to exhibit security leaks if the program is insecure. Their approach is automatic and flow-sensitive, but less efficient than type-based approach.

Both the type systems of [25] and of [4] rely on the assumption that references are opaque, i.e. the only observations that an attacker can make about a reference are those about the object to which it points. However, Hedin and Sands [**?**] have recently observed that the assumption is unvalidated by methods from the Java API, and exhibited a Jif program that does not use declassification but leaks information through invoking API methods. Their attack relies on the assumption that the function that allocates new objects on the heap is deterministic; however, this assumption is perfectly reasonable and satisfied by many implementations of the JVM. In addition to demonstrating the attack, Hedin and Sands show how a refined information flow type system can thwart such attacks for a language that allows to cast references as integers. Intuitively, their type system tracks the security level of references as well as the security levels of the fields of the object its points to.

7.5 Logical analysis of non-interference for Java

In a different line of work, several authors have investigated the use of program logics to enforce non-interference of Java programs. Darvas and co-workers [14] use dynamic logic to verify information flow policies of Java Card programs. One of their encodings of non-interference is based on the idea of self-composition (see also [6]), where the program is composed with a renaming of itself to ensure properties that involve two executions of a program. The idea of self-composition has also been put in practice by Dufay and co-workers [15], who used an extension of the Krakatoa tool [22] with self-composition primitives to verify that data mining programs from the open source repository WEKA adhere to privacy policies cast in terms of information flow. Both [14, 15] are application-oriented and do not attempt to provide a theoretical study of self-composition for Java. In a recent article, Naumann [27] sets out the details of self-composition in presence of a dynamically allocated heap; in short, one main issue tackled by Naumann is the definition of a meaningful notion of "renaming" for the heap.

Independently, Banerjee and his co-workers [2] develop a logic that allows to verify non-interference without resorting to self-composition. The logic, which is tailored to object-oriented languages, handles the heap using independence assertions inspired from separation logic.

7.6 Concurrency

Extending information flow type systems to concurrent languages is notoriously difficult because the parallel composition of secure sequential programs may itself not be secure [32]. The problem is caused by so-called internal timing leaks that arise when secret information is revealed through the scheduling of threads. In order to avoid internal timing leaks, many works on information flow type systems for concurrent languages focus on a stronger notion of non-interference that considers intermediate execution steps of programs. Thus, information flow type systems for concurrent languages typically enforce bisimulation-based notions of non-interference, at the cost of being very conservative, e.g. by rejecting programs that contain a loop with a high guard, or that perform a low assignment after a high branching statement.

Motivated by the desire to provide flexible and practical enforcement mechanisms for concurrent languages, Russo and Sabelfeld [29] develop a sound information flow type system that enforces termination-insensitive non-interference in a concurrent setting. The originality of their approach resides in the use of pseudo-commands to constrain the behavior of the scheduler so as to avoid internal timing leaks. More precisely, Russo and Sabelfeld introduce two commands to lift a thread for public to secret, or to make a temporarily secret thread public again, and require that the scheduler does not pick any low thread for execution as long as a temporarily high thread is executing.

7.7 Declassification

Information flow type systems have not found substantial applications in practice, in particular because information flow policies based on non-interference are too rigid and do not authorize information release. In contrast, many applications often release deliberately some amount of sensitive information. Typical examples of deliberate information release include sending an encrypted message through an untrusted network, or allowing confidential information to be used in statistics over large databases. In a recent survey [31], A. Sabelfeld and D. Sands provide an overview of relaxed policies that allow for some amount of information release, and a classification along several dimensions, for example who releases the information, and what information is released.

8 Conclusion

8.1 Summary

The paper introduces a sound information flow type system for a fragment of the JVM that includes objects, methods, and exceptions. To our best knowledge, it provides the first soundness proof for such a large fragment of the JVM; furthermore, the work presented in this paper has been complemented by two efforts.

First, we have extended the results of [7] to prove that programs typable in a information-flow typed fragment of Java with method calls and exceptions are typable in our system. As a corollary, we have provided the first sound information-flow type system for a fragment of Java with methods and exceptions, using the observation from [5] that type-preserving compilation and soundness of the target type system imply soundness of the source type system.

Second, we have machine-checked our results in the proof assistant Coq [13] in order to gain increased insurance in the soundness proof. Indeed, we feel it is important to resort to proof assistants for managing the complexity of the definitions and proofs involved in establishing non-interference, in particular because the definition of the type system is intricate and the soundness proof involves some lengthy and error-prone proofs by case analysis, as well as some unusual induction principle on the execution of programs. In addition, we have taken advantage of the formalization for extending the results of the previous sections:

more accurate treatment of exceptions: the type system is inherently imprecise in its treatment of exceptions, because all instructions that can throw instructions are considered as branching statements, and thus many secure programs are rejected, e.g. the compilation of $x_H \cdot f := 1; y_L := 0$ is insecure because the field access may raise an uncaught exception. In order to retain some acceptable degree of precision and not to

reject too many programs, we have formalized a more accurate control dependence region analysis that takes advantage of prior exception analysis like null pointer analysis or array bound analysis. Although the need for auxiliary analyzes has been recognized earlier, in particular by the implementors of Jif [25], we are not aware of any proof of soundness for an information flow type system parametrized by other static analyzes;

extended language coverage: our certified information flow verifier is built on top of Bicolano, which closely follows the official JVM specification and formalizes a large fragment of the JVM—although some features are omitted, e.g. initialization, subroutines (which shall soon disappear), multi-threading, dynamic class loading, garbage collection, 64-bit arithmetic and floats. Thus, our formal proofs cover a larger fragment of Java. For example, the formalization allows void methods, static method calls, etc. More importantly, our formalization deals with arrays. Special care has been taken to allow public arrays handling secret informations as it is done at source level in Jif. In their case study [3], Askarov and Sabelfeld show that such a mechanism is important to type realistic programs.

8.2 Future work

An important goal is to extend our results to multi-threaded Java, in order to broaden the scope of applications of our type system. One realistic objective is to cover mobile phone applications that are based on the MIDP profile, which supports multi-threading. Building upon the proposal of Russo and Sabelfeld [29] to control the interactions between threads and the schedulers, we have given sufficient conditions for extending a sound type system for a sequential assembly language into a sound type system for a concurrent assembly language (unpublished draft). The results do not apply directly to concurrent Java, but we hope the ideas can be lifted without any major difficulty to a concurrent extension of the $\text{JVM}_{\mathcal{E}}$.

Another important goal is to put our type system in practice. To this end, we intend to take advantage of the Coq extraction mechanism to obtain from our formal proofs a certified (lightweight) information flow bytecode for JVM programs, and to subject existing case studies to this verifier. However, we anticipate some difficulties with case studies such as battleship and mental poker, as they make an intensive use of declassification, which is not provisioned by our type system. As a first approximation, one may able to circumvent the problem by introducing declassification methods that are called explicitly to release information. For example, consider the following program fragment that uses encryption with a secret key k:

$$x_L := \{y_H\}_k$$

In order to verify the program fragment, one could treat encryption as an effect-free method $\mathsf{encrypt}_k$ that takes a high input and returns a low output and type check the fragment against the signature $H \to L$ for $\mathsf{encrypt}_k$. However, one could provide a more satisfactory treatment of declassification by adopting recent works on type-based enforcement of declassification policies. Likewise, extending our results to type systems that support dynamic policies and label polymorphism could significantly contribute to the applicability of our type systems on realistic case studies.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proceedings of POPL'99*, pages 147–160. ACM Press, 1999.
2. T. Amtoft and S. Bandhakavi aand A. Banerjee. A logic for information flow in object-oriented programs. In *Proceedings of POPL'06*, pages 91–102. ACM Press, 2006.
3. A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In S. De Capitani di Vimercati, P.F. Syverson, and D. Gollmann, editors, *Proceedings of ESORICS'05*, volume 3679 of *Lecture Notes in Computer Science*, pages 197–221. Springer-Verlag, 2005.
4. A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005. Special Issue on Language-Based Security.
5. G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In B. Steffen and G. Levi, editors, *Proceedings of VMCAI'04*, volume 2934 of *Lecture Notes in Computer Science*, pages 2–15. Springer-Verlag, 2004.

6. G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Proceedings of CSFW'04*, pages 100–114. IEEE Press, 2004.
7. G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for java. In *Symposium on Security and Privacy, 2006*. IEEE Press, 2006.
8. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Proceedings of TLDI'05*, pages 103–112. ACM Press, 2005.
9. G. Barthe and B. Serpette. Partial evaluation and non-interference for object calculi. In A. Middeldorp and T. Sato, editors, *Proceedings of FLOPS'99*, volume 1722 of *Lecture Notes in Computer Science*, pages 53–67. Springer-Verlag, 1999.
10. C. Bernardeschi and N. De Francesco. Combining Abstract Interpretation and Model Checking for analysing Security Properties of Java Bytecode. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume 2294 of *Lecture Notes in Computer Science*, pages 1–15, 2002.
11. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Checking Secure Interactions of Smart Card Applets: Extended version. *Journal of Computer Security*, 10:369–398, 2002.
12. E. Bonelli, A.B. Compagnoni, and R. Medel. Information flow analysis for a typed assembly language with polymorphic stacks. In G. Barthe, B. Grégoire, M. Huisman, and J.-L. Lanet, editors, *Proceedings of CASSIS'05*, volume 3956 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 2005.
13. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.
14. Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proceedings International Conference on Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, 2005. Preliminary version in the informal proceedings of WITS'03.
15. G. Dufay, A.P. Felty, and S. Matwin. Privacy-sensitive information flow with JML. In R. Nieuwenhuis, editor, *Proceedings of CADE'05*, volume 3632 of *Lecture Notes in Computer Science*, pages 116–130. Springer-Verlag, 2005.
16. S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, December 2003.
17. S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In R. Cousot, editor, *Proceedings of VMCAI'05*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362. Springer-Verlag, 2005.
18. P. Girard. Which security policy for multiapplication smart cards? In *Proceedings of Smartcard'99*. USENIX Association, 1999.
19. N. Heintze and J. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of POPL'98*, pages 365–377. ACM Press, 1998.
20. K. Honda and N. Yoshida. A Uniform Type Structure for Secure Information Flow. In *Proceedings of POPL'02*, pages 81–92. ACM Press, 2002.
21. H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, 11(4):615–676, 2003.
22. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard Programs annotated with JML Annotations. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.
23. M. Montgomery and K. Krishna. Secure Object Sharing in Java Card. In *Proceedings of Usenix workshop on Smart Card Technology, (Smartcard'99)*, 1999.
24. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, November 1999.
25. A.C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL'99*, pages 228–241. ACM Press, 1999.
26. D. Naumann. Verifying a secure information flow analyzer. In J. Hurd and T. Melham, editors, *Proceedings of TPHOLs'05*, volume 3603 of *Lecture Notes in Computer Science*, pages 211–226. Springer-Verlag, 2005. Preliminary version appears as Report CS-2004-10, Stevens Institute of Technology, 2003.
27. D. Naumann. From coupling relations to mated invariants for checking information flow (extended abstract). In D. Gollmann and A. Sabelfeld, editors, *Proceedings of ESORICS'06*, volume 3xxx of *Lecture Notes in Computer Science*, pages xxx–xxx. Springer-Verlag, 2006. To appear.
28. F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.
29. A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proceedings of CSFW'06*, 2006.
30. A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.

31. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of CSFW'05*. IEEE Press, 2005.
32. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of POPL'98*, pages 355–364. ACM Press, 1998.
33. Q. Sun, A. Banerjee, and D.A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In R. Giacobazzi, editor, *Proceedings of SAS'04*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99. Springer-Verlag, 2004.
34. D. Volpano and G. Smith. A Type-Based Approach to Program Security. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.
35. D. Yu and N. Islam. A typed assembly language for confidentiality. In P. Sestoft, editor, *Proceedings of ESOP'06*, volume 3924 of *Lecture Notes in Computer Science*, pages 162–179. Springer-Verlag, 2006.
36. S. Zdancewic and A.C. Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2–3):209–234, September 2002.

A Auxiliaries lemmas

All free variables are implicitly universally quantified. Naming convention allows to infer kind of each variables.

A.1 Operand stack

Lemma 31. *If* $os \sim_{st,st',\beta} os'$ *and* $k \leq k_{\text{obs}} \Rightarrow v \sim_\beta v'$ *then* $v :: os \sim_{k::st,k::st',\beta} v' :: os'$

Lemma 32. $os \sim_{st,st',\beta} os'$ *and* $v \sim_\beta v'$ *implies* $v :: os \sim_{k::st,k::st',\beta} v' :: os'$

Lemma 33. $os \sim_{st,st',\beta} os'$ *then* $\mathsf{lift}_k\, os \sim_{st,st',\beta} \mathsf{lift}_k\, os'$.

Lemma 34. $v :: os \sim_{k::st,k'::st',\beta} v' :: os'$
implies $os \sim_{st,st',\beta} os'$ *and* $v \sim_\beta v'$ *and one of the following cases:*

- *either* $k = k'$, $k \leq k_{\text{obs}}$ *and* $v \sim_\beta v'$,
- *or* $k \not\leq k_{\text{obs}}$ *and* $k' \not\leq k_{\text{obs}}$.

Lemma 35. *If* $\mathsf{length}(st_1) = \mathsf{length}(st_2)$ *and*

$$os_1 :: l_1 :: os_1' \sim_{st_1::k_1::st_1',st_2::k_2::st_2',\beta} os_2 :: l_2 :: os_2'$$

If $k_1 \leq \boldsymbol{k'_v}[0]$ *and* $k_2 \leq \boldsymbol{k'_v}[0]$, *if*

$$\forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \boldsymbol{k'_v}[i+1]$$

and

$$\forall i \in [0, \mathsf{length}(st_2) - 1],\ st_1[i] \leq \boldsymbol{k'_v}[i+1]$$

then

$$\{this \mapsto l_1, \boldsymbol{x} \mapsto os_1\} \sim_{\boldsymbol{k'_v},\beta} \{this \mapsto l_2, \boldsymbol{x} \mapsto os_2\}$$

A.2 Heap

Lemma 36. *Let* $k \in \mathcal{S}$ *a security level, for all heap* $h \in \mathsf{Heap}$ *and object* $o \in \mathcal{O}$,

$$h \preceq_k h \oplus \{\text{fresh}(h) \mapsto o\}$$

Lemma 37. $h \sim_\beta h_0$ *and* $l = \text{fresh}(h)$ *implies* $h \oplus \{l \mapsto o\} \sim_\beta h_0$

Lemma 38. *$h \sim_\beta h_0$ and $\mathsf{ft}(f) \not\leq k_{\text{obs}}$ implies $h \oplus \{l \mapsto h(l) \oplus \{f \mapsto v\}\} \sim_\beta h_0$*

Lemma 39. *$h \sim_\beta h_0$, $k \not\leq k_{\text{obs}}$ and $h \preceq_k h'$ implies $h' \sim_\beta h_0$*

Lemma 40. *If $h_1 \sim_\beta h_2$, $l_1 = \text{fresh}(h_1)$ and $l_2 = \text{fresh}(h_2)$, then $\beta' = \beta \oplus \{l_1 \mapsto l_2\}$ verifies:*

- $\beta \subseteq \beta'$,
- $h_1 \oplus \{l_1 \mapsto \mathsf{default}_C\} \sim_{\beta'} h_2 \oplus \{l_2 \mapsto \mathsf{default}_C\}$

Proof. By hypothesis, β is a bijection between $\mathsf{dom}(\beta)$ and $\mathsf{rng}(\beta)$, $\mathsf{dom}(\beta) \subseteq \mathsf{dom}(h_1)$, $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$ and for every $l \in \mathsf{dom}(\beta)$, $h_1(l) \sim_\beta h_2(\beta(l))$.

We remark first that $l_1 \notin \mathsf{dom}(\beta)$, since $l_1 \notin \mathsf{dom}(h_1)$ and $\mathsf{dom}(\beta) \subseteq \mathsf{dom}(h_1)$. Similarly, we have $l_2 \notin \mathsf{rng}(\beta)$, since $l_2 \notin \mathsf{dom}(h_2)$ and $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$.

From $l_1 \notin \mathsf{dom}(\beta)$, we deduce that $\beta \subseteq \beta'$.

Since β is a bijection between $\mathsf{dom}(\beta)$ and $\mathsf{rng}(\beta)$, since $l_1 \notin \mathsf{dom}(\beta)$ and $l_2 \notin \mathsf{rng}(\beta)$, we deduce β' is a bijection between $\mathsf{dom}(\beta')$ and $\mathsf{rng}(\beta')$.

We finally prove that for all $l \in \mathsf{dom}(\beta')$, $h_1 \oplus \{l_1 \mapsto \mathsf{default}_C\}(l) \sim_{\beta'} h_2 \oplus \{l_2 \mapsto \mathsf{default}_C\}(\beta'(l))$. If $l \neq l$ the result trivially holds by hypothesis. If $l = l$ we just have to prove that $\mathsf{default}_C \sim_{\beta'} \mathsf{default}_C$. This is true since for all fields $f \in \mathsf{dom}(\mathsf{default}_C)$, $\mathsf{default}_C.f$ is equal to 0 or *null* (elements on which $\sim_{\beta'}$ is reflexive).

Lemma 41. *If $h_1 \sim_\beta h_2$, if $l_1 = \text{fresh}(h_1)$ and $l_2 = \text{fresh}(h_2)$ then the following properties hold*

- $h_1 \oplus \{l_1 \mapsto \mathsf{default}_C\} \sim_\beta h_2$
- $h_1 \sim_\beta h_2 \oplus \{l_2 \mapsto \mathsf{default}_C\}$
- $h_1 \oplus \{l_1 \mapsto \mathsf{default}_C\} \sim_\beta h_2 \oplus \{l_2 \mapsto \mathsf{default}_C\}$

A.3 Extension of β

Lemma 42. *If $v_1 \sim_\beta v_2$ and $\beta \subseteq \beta'$ then $v_1 \sim_{\beta'} v_2$.*

Lemma 43. *If $v_1 \sim_\beta v_2$ and $\beta \subseteq \beta'$ then $v_1 \sim_{\beta'} v_2$.*

Lemma 44. *If $\rho_1 \sim_\beta \rho_2$ and $\beta \subseteq \beta'$ then $\rho_1 \sim_{\beta'} \rho_2$.*

Lemma 45. *If $os_1 \sim_{st_1, st_2, \beta} os_2$ and $\beta \subseteq \beta'$ then $os_1 \sim_{st_1, st_2, \beta'} os_2$.*

Table of Contents