A sound dependency analysis for secure information flow (extended version)

Dorina Ghindici, Isabelle Simplot-Ryl, Jean-Marc Talbot

To cite this version:

Dorina Ghindici, Isabelle Simplot-Ryl, Jean-Marc Talbot. A sound dependency analysis for secure information flow (extended version). [Research Report] RT-0347, INRIA. 2007. inria-00185263v3

HAL Id: inria-00185263

https://hal.inria.fr/inria-00185263v3

Submitted on 6 Nov 2007

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A sound dependency analysis for secure information flow (extended version)

Dorina Ghindici — Isabelle Simplot-Ryl — Jean-Marc Talbot

N° 0347

October 2007

Thème COM

Rapport technique

ISSN 0249-0803 ISRN INRIA/RT--0347--FR+ENG

A sound dependency analysis for secure information flow (extended version)

Dorina Ghindici[*] , Isabelle Simplot-Ryl[*] , Jean-Marc Talbot[†]

Thème COM —Systèmes communicants
Projet POPS

Rapport technique n° 0347 —October 2007 — 40 pages

Abstract: In this paper we present a flow-sensitive analysis for secure information flow for Java bytecode. Our approach consists in computing, at different program points, a dependency graph which tracks how input values of a method may influence its outputs. This computation subsumes a points-to analysis (reflecting how objects depend on each others) by addressing dependencies arising from data of primitive type and from the control flow of the program. Our graph construction is proved to be sound by establishing a non-interference theorem stating that an output value is unrelated with an input one in the dependency graph if the output remains unchanged when the input is modified. In contrast with many type-based information flow techniques, our approach does not require security levels to be known during the computation of the graph: security aspects of information flow are checked by labeling ”a posteriori” the dependency graph with security levels.

Key-words: Information flow, static analysis, Java bytecode, abstract interpretation, non-interference

[*] IRCICA/LIFL, Univ. Lille 1, INRIA Futurs, France. Email: {Dorina.Ghindici, Isabelle.Ryl}@lifl.fr
[†] LIF, CNRS UMR 6166, Université de Provence, France. Email: {jean-marc.talbot@lif.univ-mrs.fr}

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 —Télécopie : +33 1 60 19 66 08

Une analyse de dépendence pour le flot d’information

Résumé : Dans ce rapport, nous présentons une analyse de flot d’information pour le bytecode Java. Notre approche consiste à calculer, pour différents points du programme, un graphe de dépendances qui représente l’influence que les valeurs en entrée d’une méthode ont sur les sorties. Ce calcul inclut une analyse de pointeurs (illustrant les dépendances entre objets) à laquelle sont ajoutées les dépendances issues des données de type primitif et du flot de contrôle du programme. La construction de notre graphe est prouvée correcte par un théoreme de non-interference qui énonce qu’une valeur de sortie n’est pas liée à une valeur d’entrée dans le graphe de dépendances si la valeur de sortie ne change pas lorsque la valeur d’entrée varie. À l’inverse de beaucoup de techniques basées sur des systèmes de types, notre approche ne nécessite pas de connaître la politique de sécurité lors du calcul du graphe : le respest d’une politique de ”sécurité” en termes de flot d’information sont vérifiés en étiquetant ”a posteriori” le graphe de dépendances avec des niveaux de sécurité.

Mots-clés : Flot d’information, analyse statique, bytecode Java, interpretation abstraite, non-interference

1 Introduction

Information flow analysis [17] detects how data may flow between variables focusing on data manipulations of primitive types. This analysis is used to check data propagation in programs with regards to security requirements and aims to avoid that programs leak confidential information: observable/public outputs of a program must not disclose information about secret/confidential values manipulated by the program. Non-interference defines the absence of illicit information flow by stating that public outputs of a program remain unchanged if its secret values are modified. Data is labeled with security levels, usually $high$ for secret/confidential values and low for observable/public variables. There are information flows arising from assignments (direct flow) or from the control structure of a program (implicit flow). For example, the code `l=h` generates a direct flow from `h` to `l`, while `if(h) then l=1 else l=0` generates an implicit flow. If `h` has security level $high$ and `l` low, then the examples are insecure and generate an illicit information flow, as secret data can be induced by the reader of `l`.

In object oriented languages, like Java [14], the analysis of information flow is related to the analysis of references. *Points-to analysis* [1] is a static analysis which computes for every object the set of objects to which it may point to at runtime. Points-to analysis [18] has been developed for applications such as escape analysis [20] or optimizations, but never used as part of information flow analysis even if they are strongly related.

In this paper we propose a sound general flow-sensitive analysis of Java programs that computes a dependency graph including references and primitive types. The graph is a points-to graph extended with primitive values and dependencies raised by control flow. The dependency graph characterizes on one hand how fields of objects point to other objects, and on the other hand the dependencies between primitive values through direct or indirect flow, in the sense of non-interference. This leads to an analysis which is more general than ”traditional” information flow analysis as it computes a dependency graph abstracting the dependencies between program data: a node a is related to b if the value of b may influence the value of a. These dependencies are not made explicit in ”traditional” information flow analysis and replaced by coarser flows of security levels from a a priori fixed security lattice [8]. This approach has been initiated by the Volpano, Irvine and Smith’s model [19].

In our work, we compute these dependencies between values independently of any *a priori* information like security levels. Therefore, when applied to secure information flow, our approach allows to reuse the same analysis for various security lattices without re-analysing the code.

Note that our dependency graph should not be confused with data dependency graphs such as [7] which represent dependencies among register and memory reads and write and aim to be used for program slicing.

The language we consider is the Java bytecode language: only few papers [4, 13] take into consideration low-level languages, while the Java language is rarely specified. A type-based method for checking secure information flow in Java bytecode is presented in [6], while in [5] Barthe, Pichardie and Rezk define an information flow type system for a sequential JVM-like language and prove that it guarantees non-interference. Some other works [2, 9, 15] use static analysis to treat information flow. The main weakness of these approaches is that they are flow insensitive. Recently, Hunt and Sands proposed a family of flow-sensitive type systems [11] for tracking information flow.

All the previous cited works require the lattice security model of information flow to be known from the very beginning of the analysis. We believe that "traditional" information flow analysis can be recovered as an abstract interpretation of our dependency graphs, this abstract interpretation taking into account the security levels.

The paper is structured as follows: Section 3 presents the concrete model used in the rest of the paper. Section 4 describes the design and construction of our dependency graph during an intra-method analysis of sequential programs. We present the correctness of the construction in Section 5, using a non-interference theorem. Section 6 describes the inter-procedural analysis and adds support for method invocation. Section 7 applies the dependency graph to information flow.

2 Notations

We consider finite graphs whose vertices and edges are both labelled. Let $\mathcal{V}, \mathcal{L}$ two sets used respectively as labels for vertices and edges.

A graph G is given by a triple (V, E, ς) where V is its set of vertices (or nodes), $E \subseteq V \times V \times \mathcal{L}$ is its set of labelled edges and the mapping ς associates a label from $\mathcal{V}$ with each element of V. In a graph G, the edge from vertex u to v, labeled with l is denoted by (u, v, l), and $adj_G(u, l)$ is the set of adjacent vertices of u in G, reached by an edge labeled by l. Further, $G.V$ denotes its set of vertices and $G.E$ its set of edges; the mapping $G.\varsigma(v)$ gives the label of the node v. A node u is a leaf in a graph G if for any node v and label l, $(u, v, l) \notin G.E$.

The union of two graphs $G_1 \cup G_2$ (assuming that for all u in $G_1.V \cap G_2.V$, $G_1.\varsigma(u) = G_2.\varsigma(u)$) is a graph G_3, having $G_3.V = G_1.V \cup G_2.V$, $G_3.E = G_1.E \cup G_2.E$ and $G_3.\varsigma = G_1.\varsigma \cup G_2.\varsigma$. A graph G_1 is included in a graph G_2 (denoted $G_1 \subseteq G_2$) if there exists a subgraph G_2' of G_2 which is isomorphic[1] to G_1.

For two graphs G_1, G_2, we say that G_1 and G_2 are *similar* (denoted $G_1 \equiv G_2$) if there exists a mapping $\varsigma : G_2.V \to \mathcal{V}$ such that for all u in $G_2.V$ which is a leaf $G_2.\varsigma(u) = \varsigma(u)$ and G_1, $(G_2.V, G_2.E, \varsigma)$ are isomorphic.

For a graph G, $G[u \mapsto l]$ designates the graph that agrees with G but changes the label of the vertex u to l, while $G[(u, f) \mapsto v]$ agrees with G except that all the edges of the form (u, u', f) in $G.E$, are replaced by a unique edge (u, v, l).

A vertex v is reachable from u in a graph G if there is a path (a sequence of edges leading) from u to v and we denote by $Reach_G(u)$ the set of vertices reachable from u in G. We define $G\lfloor u \rfloor$ the subgraph of G given by $(Reach_G(u), \{(v, w, l) \mid (v, w, l) \in G.E$ and $v, w \in Reach_G(u)\}, \varsigma_{|Reach_G(u)})$.

Finally, for a function f, by $f[x \mapsto e]$ we denote the function f' such that $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = e$. For D, a subset of the domain of the function f, $f_{|D}$ is the restriction of f on D.

3 The Java Virtual Machine model

This section introduces the syntax and the model of the JVM we consider. Due to limited space, we focus on a general and representative subset of the JVM, depicted in Figure 1. Most of the rest of the JVM instructions are similar to the ones of this set (*e.g.* we give a hint how static variables and arrays can be treated as objects with

[1] The graph isomorphisms we consider take in consideration both vertex and edge labelling.

`prim` op	primitive operation taking two operands, pushing the result on the stack
`pop`	pop the top of the stack
`iconst_` n	push the primitive value n on the stack
`aconst_null`	push null on the stack
`new` C	creates new object of type C in the memory
`goto` a	jump to address a
`ifeq` a	jump to address a if the top of the stack equals to 0
`load` x	push the content of the local variable x on the stack
`store` x	pop the top of the stack and store it into the local variable x
`getfield` $f_{C'}$	load on the stack the field $f_{C'}$ of the object beeing on the top of the stack
`putfield` $f_{C'}$	store the top of the stack in the field $f_{C'}$ of an object on the stack
`invoke` $m_{C'}$	virtual invocation of method $m_{C'}$
`areturn`	return an object and exit the method

Figure 1: Instruction set

some special fields). We do not address exceptions and threads but take overwritting into account. We assume the bytecode programs to be well-typed and deal only with executions that terminate and do not throw exceptions.

We consider a set of class names $Class$, a set of methods $Methods$ and a set $Fields$ of fields names. For two classes B, C, $B \leq C$ if and only if $B = C$ or C is a super class of B. The function $Type : Obj \rightarrow Class$ returns the type of an object. The set of JVM values is defined as $Jv = Val \cup Obj$ where Val is the set of primitive values, and Obj the set of objects, including the special value `null`.

3.1 Memory model

We define the memory as a directed graph. The advantage of this representation is the independence between objects and actual locations where these objects are allocated. However, the graph representing the memory is isomorphic to any address assignment in an execution. This independence property is crucial when comparing memories for two executions of the same method for different input values.

A node designates a location and is labeled with an element from Jv, representing the JVM value stored in the location; edges represent field references and are labeled with field names (from $Fields$). Nodes containing `null` or primitive values (from Val) are leaves.

For a graph G, $G.V^{Obj}$ denotes the nodes labeled by some element from Obj ($G.V^{Obj} = \{v \in G.V \mid G.\varsigma(v) \in Obj\}$) while $G.V^{Val}$ denotes the sets of nodes labelled by some element from Val. We define by $\Pi_{Obj}(G)$ the subgraph of G restricted to objects as $(G.V^{Obj}, \{(u,v,l) \mid u,v \in G.V^{Obj}$ and $(u,v,l) \in G.E\}, G.\varsigma_{|G.V^{Obj}})$.

For a memory graph G, $G.\varsigma$ is an injective function associating with each vertex (location) the JVM value (from Jv) it contains, each object value being associated with a unique location. This allows us to make no distinction between a node of $G.V^{Obj}$ and its label. For l in Obj, we freely use $G.\varsigma^{-1}(l)$ to denote the (unique) vertex labeled by l in G. Moreover, there exists a unique vertex labelled by `null`. For each primitive field of an object, the memory has a location (vertex) containing (labeled with) the value of the field and this vertex is the target of a unique edge. We consider an allocator function $G' = new(o, C)$ which creates a new graph structure for an object named o of type C; o is labelling the root of the graph. The graph G' contains the initial values of the new created object (vartices labeled by 0 for fields of primitive type and edges to `null` for fields of type object).

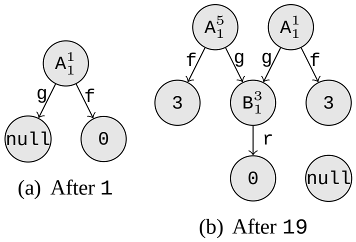
Figure 2(a) shows the result of the allocator function for A_1^1, thus after executing instruction 1.

```
class A {
  int f;
  B g;
}

class B {
  int r;
}
```

```
1 : new A
2 : store x
3 : new B
4 : store y
5 : new A
6 : store z
7 : load  x
8 : load  y
9 : putfield g_A
10: load  z
```

```
11: load  y
12: putfield g_A
13: load  x
14: iconst_3
15: putfield f_A
16: load  z
17: load  x
18: getfield f_A
19: putfield f_A
```



(a) After `1`

(b) After `19`

Figure 2: Memory Example

3.2 Operational semantics

For a method m of class C, n_m denotes the number of its arguments, χ_m its set of local variables, P_m its instruction list; $P_m[i]$ denotes the ith bytecode of the method m. A (concrete) state of our machine is a pair $Q = (fr, G)$, where fr represents a stack of frames corresponding to a dynamic chain of method calls (with the current frame on the top), and G is the memory graph. A frame has the form (pc, ρ, s), where $pc \in P_m$ is the program counter, $\rho : \chi_m \to Jv$ represents a value assignment of local variables and s is a stack with elements in Jv. The operational semantics of Java bytecode is presented in Figure 3. The concatenation $n :: s$ denotes a stack having n on top, while $f_{C'}$ designates the field f of the class C'.

The bytecode `putfield` has two rules depending on the type of the manipulated field: changing the value of a primitive field means changing the label of a vertex, whereas changing an object field consist in changing the edge to the vertex containing the new pointed object.

Figure 2(b) shows the memory graph after executing the entire code from Figure 2. Affecting the object B_1^3 to the field g of A_1^1 (bytecodes `7-9`) consists in creating a new edge labelled by `g` from A_1^1 to B_1^3, while the assignment of the value 3 to the field `f` of A_1^1 (bytecodes `13-15`) changes the label of the node reached from A_1^1 with an `f`-edge.

The intra-method control flow graph CF_m of a method m is defined as usual: CF_m is a graph with P_m as set of vertices and edges from i to each elements of $succ(i)$, with $succ(i)$ given by the bytecode semantics, for each $i \in P_m$ as:

- $succ(i) = \{a\}$ if $P_m[i] =$ `goto` a,
- $succ(i) = \{a, i+1\}$ if $P_m[i] =$ `ifeq` a,
- $succ(i) = \emptyset$ if $P_m[i] =$ `areturn`,
- $succ(i) = \{i+1\}$ otherwise.

We also use the notation $pred(i) = \{j \mid i \in succ(j)\}$. An exit-point in a control flow graph is a node without successor.

For a method m, a *block* B is a subset of the instruction list P_m together with a distinguished instruction, called the *entry-point* of B ($\mathsf{Entry}(B)$) such that the control flow graph CF_B of B is a subgraph of CF_m; all vertices in CF_B are reachable from $\mathsf{Entry}(B)$ and CF_B has a unique exit-point ($\mathsf{Exit}(B)$). We will assume that P_m is itself a block by adding a unique (fake) exit-point to its control flow graph. A state $Q = ((i, \rho, s) :: fr, G)$ is a *good initial* state for a block B, if $B[i]$ is defined and executing the block B starting from the state Q terminates in a state denoted by $instr_B(Q)$.

$$\frac{P_m[i] = \texttt{prim}\, op \qquad n = op(n_1, n_2)}{((i, \rho, n_1 :: n_2 :: s) :: fr, G) \longrightarrow ((i+1, \rho, n :: s) :: fr, G)} \qquad \frac{P_m[i] = \texttt{goto}\ a}{F \longrightarrow ((a, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \texttt{aconst\_null}}{F \longrightarrow ((i+1, \rho, \texttt{null} :: s) :: fr, G)} \qquad \frac{P_m[i] = \texttt{pop}}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((i+1, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \texttt{iconst\_}n}{F \longrightarrow ((i+1, \rho, n :: s) :: fr, G)} \qquad \frac{P_m[i] = \texttt{new}\ C \qquad o = C_i^{\mathsf{fresh}(C_i, G)} \quad G' = new(o, C)}{F \longrightarrow ((i+1, \rho, G'.\varsigma^{-1}(o) :: s) :: fr, G \cup G')}$$

$$\frac{P_m[i] = \texttt{ifeq}\ a \qquad n \neq 0}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((i+1, \rho, s) :: fr, G)} \qquad \frac{P_m[i] = \texttt{ifeq}\ a \qquad n = 0}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((a, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \texttt{load}\ x}{F \longrightarrow ((i+1, \rho, \rho(x) :: s) :: fr, G)} \qquad \frac{P_m[i] = \texttt{store}\ x}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((i+1, \rho[x \mapsto n], s) :: fr, G)}$$

$$\frac{P_m[i] = \texttt{getfield}\ f_{C'} \qquad n \neq \texttt{null} \qquad n' \in adj_G(G.\varsigma^{-1}(n), f_{C'})}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((i+1, \rho, G.\varsigma(n') :: s) :: fr, G)}$$

$$\frac{P_m[i] = \texttt{putfield}\ f_{C'} \qquad n \neq \texttt{null} \qquad v \in Val \qquad n' \in adj_G(G.\varsigma^{-1}(n), f_{C'})}{((i, \rho, v :: n :: s) :: fr, G) \longrightarrow ((i+1, \rho, s) :: fr, G[n' \mapsto v])}$$

$$\frac{P_m[i] = \texttt{putfield}\ f_{C'} \qquad n \neq \texttt{null} \qquad v \notin Val}{((i, \rho, v :: n :: s) :: fr, G) \longrightarrow ((i+1, \rho, s) :: fr, G[(G.\varsigma^{-1}(n), f_{C'}) \mapsto G.\varsigma^{-1}(v)])}$$

$$\frac{P_m[i] = \texttt{invoke}\ m_{C'} \qquad o \neq \texttt{null}}{((i, \rho, p_{n_m} :: \cdots :: p_1 :: o :: s) :: fr, G) \longrightarrow ((0, \{0 \mapsto o, 1 \mapsto p_1 \ldots n_m \mapsto p_{n_m}\}, \epsilon) :: (i, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \texttt{areturn}}{((i, \rho, v :: s) :: (i', \rho', s') :: fr, G) \longrightarrow ((i'+1, \rho', v :: s') :: fr, G)}$$

where $F = ((i, \rho, s) :: fr, G)$. `prim` *op* stands for primitive operations with two parameters. The function $\mathsf{fresh}(c, G)$ returns a natural k such that c^k is not used as a vertex label in G.

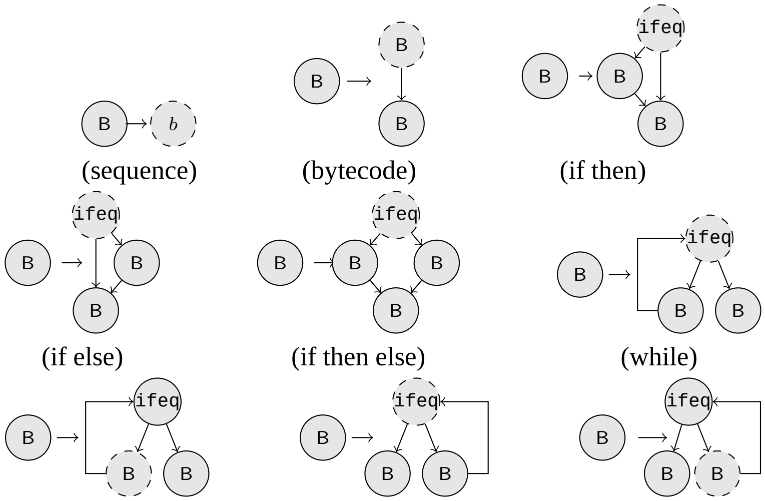
Figure 3: A subset of operational semantics rules

We use the notion of postdominance as defined in [3]: in a control flow graph CF_B of a block B, a node n' post-dominates a node n if n' belongs to any path from n to $\mathsf{Exit}(B)$. We denote $PD(n)$ the set of post-dominators of n. The immediate post-dominator of n, $ipd(n) \in PD(n)$ satisfies that$\forall n'' \in PD(n)$, if $n'' \neq n'$, then $n'' \in PD(n')$. In a block B with control flow graph CF_B, for each conditional instruction $B[i] = \texttt{ifeq}\ a$, we define its *dependency region* as the set of instructions executed under this condition: $reg(i) = Reach_{CF_B}(i) \setminus Reach_{CF_B}(ipd(i))$. We compute a function $cxt : B \rightarrow \wp(B)$ representing the context (the set of conditional bytecodes) under which each instruction is executed: $cxt(i) = \{j \mid i \neq j \wedge i \in reg(j)\}$.

3.3 Assumption about programs : ”Structured” Blocks

We restrict blocks we consider by means of by a graph grammar $\mathcal{G}$. This grammar allows us to rewrite a graph into another one according to some production rules. These rules have a single node in the left part and a graph in their right part. Nodes in the left-hand side of transition rules are non-terminals We assume a unique non-terminal B for blocks wich is also the axiom of $\mathcal{G}$. Nodes corresponding to bytecodes are non-terminals and cannot be rewritten.

The grammar $\mathcal{G}$ is given by the following set of production rules of Figure 4 (the entry-point in the right-side is the dashed one). Note that in these production rules, for the bytecode `ifeq` the two children of this node are distinguished, as according to the concrete semantics, they correspond to two different rules.



(sequence) (bytecode) (if then)

(if else) (if then else) (while)

(repeat) (while not) (repeat not)

Figure 4: Graph grammar

Starting from the axiom B, a derivation is a succession of rewriting steps: a step consists in replacing, according to a production rule $\mathsf{B} \rightarrow G'$, the left-hand side B by the corresponding right-hand side G' such that edges are tranfered as follows: each incoming edge of the node B targets now the entry-point of G'. There is an edge from the exit-point of G' to the successor of B in the original graph (if any). In the resulting graph, the nodes from the right-hand side loose their entry node status (resp. exit node status) is they receive an incoming edge (resp. an outgoing edge).

A graph is produced by the grammar $\mathcal{G}$ if it can be obtained by rewriting from the axiom. Note that all the produced graphs have unique (remaining) entry-point and exit-point. A block is “structured” if its control flow graph is produced by $\mathcal{G}$. From now on, unless specified the blocks we consider are “structured” blocks.

3.4 Non-interference

Our analysis will compute an abstraction on how the value of some objects may depend or not on the value of others. We formalize this notion of dependency through non-interference: roughly, at the method level, the non-interference between an object ω and an input value η (of primitive type) can be stated as “changing of value of η does not affect the value of ω”. The “value” of an object is defined by the values of its fields of primitive types and recursively, by the “values” of its object fields. An input value of a block is a value of primitive type chosen in the concrete state before the execution of the block; formally:

Definition 3.1 (Set of input values) *Let $Q = ((i, \rho, s) :: fr, G)$ be a state. Then, the set of* input values *of Q is $\mathcal{I}(Q) = G.V^{Val} \cup \{x \in \chi_m \mid \rho(x) \in Val\}$.*

Definition 3.2 (Value-change) *The value-change of a state $Q = ((i, \rho, s) :: fr, G)$ and an input value η ($\eta \in \mathcal{I}(Q)$) is a state $Q' = ((i, \rho', s) :: fr, G')$ such that for some value a of primitive type, either*

- *$\eta \in G.V^{Val}$, $\rho = \rho'$ and $G' = G[\eta \mapsto a]$, or*
- *$\eta \in \{x \in \chi_m \mid \rho(x) \in Val\}$, $\rho' = \rho[\eta \mapsto a]$ and $G' = G$.*

Definition 3.3 (Non-interference) *Let B be a block. Let Q_1, Q_2 be two states such that $Q_2 = instr_B(Q_1)$. Let G_1, G_2 be the memory graphs of Q_1 and Q_2 respectively.*

For some input value η from $\mathcal{I}(Q_1)$ and for any object node ω in G_1, η does not interfer in *B with ω if for Q'_1, a value-change of Q_1 and η, and the state Q'_2 (having G'_2 for memory graph) equal to $instr_B(Q'_1)$ it holds that*

$$G_2\lfloor\omega\rfloor \equiv G'_2\lfloor\omega\rfloor.$$

4 Intra-method abstract dependency

In this section, we present the core of our analysis, that is the design of the dependency graph. The dependency graph is an ”abstraction” of the memory graph and contains all possible dependencies between objects and input values. In Section 5, we formalize non-interference: an object and an input value do not interfeer if there is no path between them in the dependency graph.

We present here the intra-method analysis and take in consideration programs without method call: the algorithm consists in computing a dependency graph for each program point of a method. The method invocation and inter-method analysis is discussed in Section 6. Our approach can be context-insensitive or context-sensitive.

4.1 The abstract model: Dependency graph

Let us present the abstract model on an example. We consider the example in Figure 5 and the dependency graph obtained after the execution of method m. To help the reader, we refer to Java source code, but our analysis works on Java bytecode.

The dependency graph G at a program point is a representation of the concrete memory such that, when restricted to objects, G and the memory graph are related by an abstraction relation. Then the abstraction G is prolonged with primitive values and implicit flow dependencies. Nodes in this graph contain abstractions of JVM objects, constants and newly created objects as well as initial values of primitive type of the method. For a dependency graph G, G^{Obj} denotes the set of nodes abstracting JVM objects, while G^{Val} denotes the nodes corresponding to primitive values. As for the memory graph, we can define by $\Pi_{Obj}(G)$ the restriction of a dependency graph G to objects.

Hence, nodes of the graph G are defined to take into account the model:

- $n^n_{pc}, pc \in P_m$: node modeling all the objects created by the execution of the object allocation instruction pc. We use the *object allocation site model* as all objects created at the same program statement have the same abstraction,

- $n^c_{pc}, pc \in P_m$: constant value created at instruction pc. There is a unique node for every constant creation statement in the method,
- $n^{\mathtt{null}}_{-1}$: special node for `null`.

```
0: void m(A p1, A p2, int h){
1:   if(h)
2:     v = p1;
3:   else
4:     v = p2;
5:   v.f = 0;
6: }
```

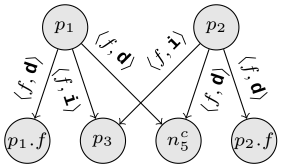

p_1
p_2
$\langle f, \mathbf{d} \rangle$
$\langle f, \mathbf{i} \rangle$
$\langle f, \mathbf{d} \rangle$
$\langle f, \mathbf{i} \rangle$
$\langle f, \mathbf{d} \rangle$
$\langle f, \mathbf{d} \rangle$
$p_1.f$
p_3
n^c_5
$p_2.f$

Figure 5: A Dependency Graph

Static variables can be modeled as fields of a global object, n^s_{-1}, thus instructions dealing with static variables are similar to the ones treating normal objects. *Arrays* are objects having two fields: length and content, holding all the elements of the array.

Edges represent links between nodes; they are labeled with $\langle f, q \rangle$, where $f \in \mathit{Fields}$ is the name of the field ans and $q \in \mathcal{F} = \{\mathbf{d}, \mathbf{i}\}$ is the type of flow: **i** for implicit flow and **d** for direct links, with the order relation $\mathbf{i} \sqsubseteq \mathbf{d}$.

Let us present the abstract model on an example. We consider the example in Figure 5 and the dependency graph obtained after the execution of method m. To help the reader, we refer to Java source code, but our analysis works on Java bytecode. The edge labeled $\langle f, \mathbf{d} \rangle$ between p_1 and n^c_5 shows that there is direct flow, arising from an assignment, from n^c_5 to the field f of p_1. The edge labeled $\langle f, \mathbf{i} \rangle$ between p_1 and p_3 expresses the implicit flow: the value of p_3 might be deduced from the field f of p_1.

To deal with implicit flows, we use the notion of regions. When executing an instruction i which adds an edge $(u, v, \langle f, \mathbf{d} \rangle)$, edges having the form $(u, v', \langle f, \mathbf{i} \rangle)$ are also added, where v' is the value tested by conditional instructions in $cxt(i)$. The intuition behind is that someone who knows the control structure of the program and can observe the field f of u, is able to deduce the value of v'.

To unify the model, we add a special value field ref of primitive type to each object, which stands for the address of the object. When comparing two objects or an object to `null`, the value tested is the ref field. The code `if(o == null) a.x = b` generates an implicit flow from the field x of a to the address ref of o ($(a, o.ref, \langle x, \mathbf{i} \rangle)$).

Implicit flows are from objects modified inside a region to values on which the execution of the region depends. These facts combined with the use of the special field ref, allow us to deduce the properties of a dependency graph:

1. any node $u \in G.V^{Val}$ is a leaf,
2. for any edge of type $(u, u', \langle f, \mathbf{i} \rangle) \in G.E)$, $u \in G.V^{Obj} \setminus \{n^{\mathtt{null}}_{-1}\}$ and $u' \in G.V^{Val}$.

These properties state that edges between two references are always direct edges: if $u_1, u_2 \in G.V^{Obj}$ and $(u_1, u_2, \langle f, t \rangle) \in G.E$ then $t = \mathbf{d}$. Value nodes are always leaves, and implicit edges are always between a reference and a value node. Thus, the edges generated by the information flow and the primitive values are not imprecated in the graph restricted to objects. Our construction is a points-to graph prolonged by edges about primitive values and implicit flows.

$$\frac{(\rho, v_1 :: v_2 :: s, G)}{(\rho, v_1 \cup v_2 \cup TV_\Gamma :: s, G)}\,\texttt{prim}\ op \qquad \frac{(\rho, s, G)}{(\rho, \{\langle n_i^n, \textbf{d}\rangle\} \cup TV_\Gamma :: s, G)}\,\texttt{new\_i}\ C$$

$$\frac{(\rho, s, G)}{(\rho, s, G)}\,\texttt{goto}\ a \qquad \frac{(\rho, s, G)}{(\rho, \{(n_{-1}^{\texttt{null}}, \textbf{d})\} \cup TV_\Gamma :: s, G)}\,\texttt{aconst\_null}$$

$$\frac{(\rho, v :: s, G)}{(\rho, s, G)}\,\texttt{ifeq}\ a \qquad \frac{(\rho, s, G)}{(\rho, \{(n_i^c, \textbf{d})\} \cup TV_\Gamma :: s, G)}\,\texttt{iconst\_i}v$$

$$\frac{(\rho, v :: s, G)}{(\rho, s, G)}\,\texttt{pop} \qquad \frac{(\rho, s, G)}{(\rho, \rho(x) \cup TV_\Gamma :: s, G)}\,\texttt{load}\ x \qquad \frac{(\rho, u :: s, G)}{(\rho[x \mapsto u \cup TV_\Gamma :: s, G)}\,\texttt{store}\ x$$

$$\frac{(\rho, u :: s, G)}{(\rho, \{(e, t) \mid e \in adj_G(e', \langle f_{C'}, t\rangle) \wedge e' \in V_{\textbf{d}}^u\} \cup \{\langle e, \texttt{i}\rangle \mid e \in V_{\texttt{i}}^u\} \cup TV_\Gamma :: s, G)}\,\texttt{getfield}\ f_{C'}$$

$$\frac{(\rho, v :: u :: s, (V, E))}{\left(\rho, s, \left(V, \begin{array}{ll} E & \cup\{(e, e', \langle f_{C'}, t\rangle) | \langle e, \textbf{d}\rangle \in u, e \neq n_{-1}^{\texttt{null}}, \langle e', t\rangle \in v\} \\ & \cup\{(e, e', \langle f_{C'}, \texttt{i}\rangle) | \langle e, \textbf{d}\rangle \in u, e \neq n_{-1}^{\texttt{null}}, e' \in \Gamma \cup V_{\texttt{i}}^u\} \end{array}\right)\right)}\,\texttt{putfield}\ f_{C'}$$

with $V_{\textbf{d}}^u = \{e \mid \langle e, \textbf{d}\rangle \in u\}$ $\quad V_{\texttt{i}}^u = \{e \mid \langle e, \texttt{i}\rangle \in u\}$ $\quad TV_\Gamma = \{\langle e, \texttt{i}\rangle \mid e \in \Gamma\}$

Figure 6: Subset of the abstract transformation rules

4.2 Abstract semantics

Building the dependency graph requires to approximate local variables and stack contents, and to deal with implicit flow, we must know the conditions under which the local variables and the stack are modified. Thus, elements from stack and local variables have the form $\langle u, t\rangle$ where $u \in G.V$ and $t \in \mathcal{F}$. Hence, an abstract state is of the form $Q = (\rho, s, G)$ where G is the dependency graph, ρ is a mapping from χ_m to $\wp(G.V \times \mathcal{F})$ and s is the stack with elements in $\wp(G.V \times \mathcal{F})$.

For every type of bytecodes b, we define a transformation rule $Q' = \overline{instr_b}(Q, \Gamma)$ where Γ is the set of nodes u corresponding to values on which depends the execution of the bytecode, reflecting the impact of the implicit flow. The semantics of transformation rules is presented in Figure 6.

To reflect the impact of control regions on stack and local variables array, every instruction modifying the last two (push for stack or store for local variables array) takes into consideration the values in the context. For example, the instructions `new` and `iconst` push on the stack new abstract values but also the context under which the operation takes place. The `store` bytecode, stores in the local variable array not only the top of the stack, but also the nodes in Γ, as the writing is done under its control.

The instruction `getfield` pushes on the stack the adjacent of u , if u is an object reference, or keeps u on the stack if it is primitive value arising from implicit flow.

The most significant bytecode is `putfield`, as it modifies the dependency graph. Apart edges from direct nodes in u (having the form $(e, \textbf{d})$) to elements in v, implicit flow edges from the nodes e ($(e, \textbf{d}) \in u$) to nodes in Γ, the instruction adds implicit flow edges from nodes e ($(e, \textbf{d}) \in u$) to implicit nodes e' ($(e', \texttt{i}) \in u$). The presence of nodes like $(e', \texttt{i})$ in u signifies that the objects in u depend on e'. Thus, we propagate the implicit dependencies of an object to every field being modified of that object.

4.3 Algorithm

From now on, we assume that a method or a block is represented by its control flow graph.

Our analysis is flow sensitive as it computes a dependency graph at each program point. The analysis is defined in the context of a monotone framework [16] for data flow analysis. To comply with the framework, we define an order relation $\sqsubseteq$ and a join operator $\sqcup$ on the property space $\mathcal{S}$ (the set of pairs (Q, Γ)) and the execution of a method as a set of equations using the transformation rules $\overline{instr}_b$.

Definition 4.1 (Ordering relation on $\mathcal{S}$) *Let $S_1 = ((\rho_1, s_1, G_1), \Gamma_1) \in \mathcal{S}$ and $S_2 = ((\rho_1, s_1, G_1), \Gamma_2) \in \mathcal{S}$ be two states. Then, S_1 is said to be smaller than S_2, i.e. $S_1 \sqsubseteq S_2$ if and only if $s_1 \sqsubseteq s_2$, $\rho_1 \sqsubseteq \rho_2$, $G_1 \subseteq G_2$ and $\Gamma_1 \subseteq \Gamma_2$.*

To order the local variables ρ_1 and ρ_2 we use the classical point-wise ordering relation between functions. Given two stacks s_1 and s_2, s_1 is said to be smaller than s_2, i.e. $s_1 \sqsubseteq s_2$, if both stacks are empty or if $s_1 = v_1 :: s'_1$ and $s_2 = v_2 :: s'_2$ with $v_1 \subseteq v_2$ and $s'_1 \sqsubseteq s'_2$.

The join operator is defined as follows: $(Q_1, \Gamma_1) \sqcup (Q_2, \Gamma_2) = (Q_1 \sqcup Q_2, \Gamma_1 \cup \Gamma_2)$. The join operator $\sqcup$ on two states $(\rho, s, G) = (\rho_1, s_1, G_1) \sqcup (\rho_2, s_2, G_2)$ is a component wise operator ($s = s_1 \sqcup s_2$, etc).

Then, $(\mathcal{S}, \sqsubseteq, \sqcup)$ forms a semi-joint lattice which satisfies the Ascending Chain Property [16] (all increasing sequences in $\mathcal{S}$ become eventually constant) required by the monotone framework.

We can now define the analysis of a block of instructions $B \subseteq P_m$ of a method m as an equation system $\mathcal{E}_B$, starting from a given initial state (Q_0, Γ_0) (remind that we consider terminating executions without exception and we consider here initial states that allow such executions).

For every node i in the control flow graph of B, (Q_i, Γ_i) represent the state and the context (required by the implicit flow) under which the instruction i is executed: It represents, the conditions tested (the top of the stack) by the instructions in $cxt(i)$. Thus, for all nodes i in CF_B:

$$(Q_i, \Gamma_i) = \begin{cases} (Q_0 \sqcup_{j \in pred(i)} \overline{instr}_{B[j]}(Q_j, \Gamma_j), & \text{if } i = \mathsf{Entry}(B) \\ \quad \Gamma_0 \cup \{u \mid \langle u, t\rangle \in v \text{ with } Q_k = (\rho, v :: s, G), k \in cxt(i)\}) & \\ (\sqcup_{j \in pred(i)} \overline{instr}_{B[j]}(Q_j, \Gamma_j), & \text{otherwise} \\ \quad \{u \mid \langle u, t\rangle \in v \text{ with } Q_k = (\rho, v :: s, G), k \in cxt(i)\}) & \end{cases} \tag{4.1}$$

The transformation rules are monotone with respect to the ordering relation $\sqsubseteq$, thus we can solve the system (4.1) using standard methods for monotone dataflow analysis.

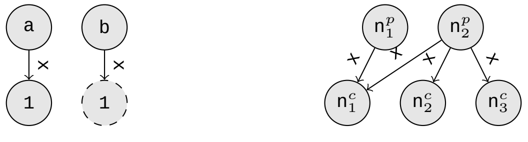
Lemme 4.2 (Monotonicity of the transformation rules) *For every instruction b, the transformation rule $\overline{instr}_b$ is monotone with respect to the ordering relation $\sqsubseteq$.*

The Proof of this Lemma is made by case analysis on each instruction b, based on the transformation rules in Figure 6, it is given in Appendix A.

Our algorithm is a forward dataflow may-analysis [10], as the computed dependency graph for a given program point is the union of graphs created by all the execution paths reaching that point.

For a block B and a pair (Q, Γ), we define $\overline{instr}_B(Q, \Gamma)$ as the state $\overline{instr}_{B[e]}(Q_e, \Gamma_e)$ where $e = \mathsf{Exit}(B)$ and (Q_e, Γ_e) is obtained in the least solution of the system of equations $\mathcal{E}_B$ starting from the initial state (Q, Γ).

```
void m (C a, C b) {
  int i=1;
  int j=0;
  int k=2;
  a.x = i;
  v.f = 0;
  b.x = k-a.x-j;
}
```



(a) memory graph

(b) dependency graph

Figure 7: Relations Example

4.4 Relations between abstract and concrete worlds

We now relate formally abstract and concrete semantics. We consider abstraction functions to relate restrictions of our memory graphs to nodes representing objects and dependency graphs. Because of the allocation site model, it is possible to relate concrete nodes to abstract nodes in a non ambiguous manner. For complete graphs (including nodes representing values), we cannot define an abstract relation since it is not possible to associate uniquely a concrete node value with an abstract one in the model we use.

Let us consider the example of Figure 7. The execution of the method `m` leads to the memory graph 7(a). Anyway, as we aim at capturing the flow between data, our analysis produces the dependency graph 7(b) for `m`. For example, it is not possible to decide which node to associate with the concrete dashed node: n_1^c, n_2^c, or n_3^c since the value `1` is the result of a calculus that implies several values. This is the reason why the abstraction relation is only defined for the restriction of graphs to their object part.

Definition 4.3 (Abstraction of a memory graph) *Let $\mathcal{V}$ and $\overline{\mathcal{V}}$ be two sets of nodes. A relation α from $\mathcal{V}$ to $\overline{\mathcal{V}}$ is an* abstraction function *if it is a mapping that respects the allocation site model: for all k, all the C_i^k are mapped to n_i^n, and for all $u \in \mathcal{V}$ such that $\varsigma(u) = \texttt{null}$, $\alpha(u) = n_{-1}^{\texttt{null}}$. This relation is canonically extended to graphs defined over sets of nodes $\mathcal{V}$ and $\overline{\mathcal{V}}$. Let G be a memory graph with $G.V = G.V^{Obj} \subseteq \mathcal{V}$. Then $\alpha(G)$ is the graph*

$$(\{\alpha(u) \mid u \in G.V^{Obj}\}, \{(\alpha(u), \alpha(v), \langle e, \boldsymbol{d}\rangle) \mid u, v \in G.V^{Obj} \wedge (u, v, e) \in G.E\}).$$

Definition 4.4 (α-abstraction) *Let G be a memory graph and $\overline{G}$ be an abstract graph. Then, $\overline{G}$ is an α-abstraction of G, denoted by $G \lhd^{\alpha} \overline{G}$ if $\alpha(\Pi_{Obj}(G)) \subseteq \Pi_{Obj}(\overline{G})$.*

The abstraction function must be defined between the full state manipulated by the program instructions. The stack abstraction is a component-wise operation. The local variables abstraction is defined similarly.

Definition 4.5 (Stack abstraction) *Let α be an abstraction function. Let G be a memory graph and $\overline{G}$ be an abstract graph such that $G \lhd^{\alpha} \overline{G}$. An abstract execution stack $\overline{s}$ is the α-abstraction of a concrete execution stack s (denoted by $s \lhd^{\alpha} \overline{s}$), if $\overline{s}$,s are both empty or if $s = v :: s_1$, $\overline{s} = \overline{v} :: \overline{s_1}$, $s_1 \lhd^{\alpha} \overline{s_1}$, and $(\alpha(G.\varsigma^{-1}(v)), \boldsymbol{d}) \in \overline{v}$ if $v \in G.V^{Obj}$.*

Definition 4.6 (State abstraction) *Let α be an abstraction function. Given a concrete state $Q = ((pc, \rho, s) :: fr, G)$ and an abstract state $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$, $\overline{Q}$ is an α-abstraction of Q (denoted by $Q \lhd^{\alpha} \overline{Q}$) if $G \lhd^{\alpha} \overline{G}$, $s \lhd^{\alpha} \overline{s}$, and $\rho \lhd^{\alpha} \overline{\rho}$.*

5 Soundness of the intra-method analysis

We now prove the correctness of the construction presented in the previous section with respect to non-interference: if there is no dependency (path in the abstract graph) between an object and an input value, then changing the input value in the concrete graph will not affect the concrete graph of the object. Thus, this section is devoted to the proof of a non-interference theorem.

Definition 5.1 (α-abstraction extension) *Let $Q = ((i, \rho, s) :: fr, G)$ be a concrete state, and $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$ such that $Q \lhd^{\alpha} \overline{Q}$. Then, $\alpha_Q : \mathcal{I}(Q) \longrightarrow \wp(\overline{G}.V^{Val})$ is the unique extension of α in Q if*

$$\begin{array}{ll} \forall \eta \in \mathcal{I}(Q) \cap G.V^{Val}, & \alpha_Q(\eta) = \{\overline{\eta} \mid \exists(\alpha(u), \overline{\eta}, \langle f, \boldsymbol{d}\rangle) \in \overline{G}.E, \mathit{with}(u, \eta, f) \in G.E\} \\ \forall \eta \in \mathcal{I}(Q) \cap \chi_m, & \alpha_Q(\eta) = \{e \mid \langle e, t\rangle \in \overline{\rho}(\eta))\} \end{array}$$

Definition 5.2 *Let α be an abstraction function. Let $Q = ((i, \rho, s) :: fr, G)$ be a concrete state, $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$ such that $Q \lhd^{\alpha} \overline{Q}$. Let $\beta \subseteq \overline{G}.V^{Val}$. For any $v \in G.V^{Obj}$, we have $free_{\alpha}(v, \beta, Q)$ if one of the following conditions holds:*

- *$\exists\ (u, v, f) \in G.E$, with $u \in G'.V^{Obj}$ and $\beta \cap adj_{\overline{G}}(\alpha(u), f) = \emptyset$,*
- *$\exists x$ such that $(\alpha(v), \boldsymbol{d}) \in \overline{\rho}(x)$ and $\nexists(\overline{\eta}, \boldsymbol{i}) \in \overline{\rho}(x)$ with $\overline{\eta} \in \beta$,*
- *$\exists i$ such that $(\alpha(v), \boldsymbol{d}) \in \overline{s}[i]$ and $\nexists(\overline{\eta}, \boldsymbol{i}) \in \overline{s}[i]$ with $\overline{\eta} \in \beta$.*

Theorem 5.3 (Non-interference) *Let B be an instruction block. Let Q_1 be a concrete state, $\overline{Q_1}$ such that $Q_1 \lhd^{\alpha} \overline{Q_1}$, $Q_2 = instr_B(Q_1)$, and $\overline{Q}_2 = \overline{instr}_B(\overline{Q}_1, \Gamma_1)$. Let G_1 be the memory graph of Q_1 and $\overline{G_2}$ be the abstract graph from $\overline{Q_2}$. For an object node ω in $G_1.V^{Obj}$ and an input value η from $\mathcal{I}(Q_1)$ such that $free_{\alpha}(\omega, \alpha_{Q_1}(\eta), Q_1)$, if $Reach_{\overline{G_2}}(\alpha(\omega)) \cap \alpha_{Q_1}(\eta) = \emptyset$ then η does not interfer with ω in B.*

The first step of the proof consists in the soundness of the pointer analysis contained in our analysis. After defining the notion of state-variation which describes how states can be affected by the modification of input values, we complete the proof of soundness for our analysis by the non-interference theorem. As we compare, in this section, concrete and abstract worlds, we denote by e a concrete element and by $\overline{e}$ an abstract element. Note that $\overline{e}$ is just an abstract element, and not necessarily the abstraction of e.

5.1 Abstraction correctness

We show that when the dependency graphs are restricted to references, our analysis is a sound points-to analysis. We rely on the dataflow framework which is slightly adapted to take into account the flow information of the Γ's.

Lemme 5.4 (Local Soundness) *Let α be an abstraction function. Let Q be an execution state and $\overline{Q}$ an abstract state such that $Q \lhd^{\alpha} \overline{Q}$. Then for any bytecode b, for any set of abstract values Γ, $instr_b(Q) \lhd^{\alpha} \overline{instr_b}(\overline{Q}, \Gamma)$.*

The proof is done by case analysis on each bytecode instruction and can be found in Appendix B.1.

We now want to establish the following property.

Proposition 5.5 (Points-to correctness) *Let B be a block, Q be a good initial (concrete) state for B, α be an abstraction function, and $\overline{Q}$ be an abstract state. Then, for any set of abstract values Γ,*

$$Q \lhd^{\alpha} \overline{Q} \textit{ implies } instr_B(Q) \lhd^{\alpha} \overline{instr}_B(\overline{Q}, \Gamma).$$

The proof relies on the general theorem stating the correctness of the global soundness provided that the local correctness holds. However, our system of equations is defined for a pair (Q, Γ) and the computation of the values of Γ's is not performed structurally on the control flow graph but based on regions instead.

Our proof will simply show that the restriction to objects of dependency graphs from the computed abstract states $\overline{Q}$ are independent from the Γ's. Therefore, we will be able to define a new system of equation $\mathcal{E}'_B$ for a block B that differ from $\mathcal{E}_B$ only on the interpretation of the functionals appearing in the system, that becomes independent of Γ. But it turns out that the points-to graphs in the states computed for $\mathcal{E}_B$ and $\mathcal{E}'_B$ are identical.

For two dependency graphs $\overline{G}$, $\overline{G}'$, we write $\overline{G} \doteq \overline{G}'$ iff $\overline{G}$, $\overline{G}'$ are equal when restricted to their object part, ie $\Pi_{Obj}(\overline{G}) = \Pi_{Obj}(\overline{G}')$. This relation $\doteq$ is also defined on sets of typed-flow vertices as : $S_1 \doteq S_2$ if $\{e \mid \langle e, \mathbf{d}\rangle \in S_1 \wedge e \in \mathcal{V}^{Obj}\} = \{e \mid \langle e, \mathbf{d}\rangle \in S_2 \wedge e \in \mathcal{V}^{Obj}\}$, where $\mathcal{V}^{Obj}$ is a set of vertices. This relation is extended to abstract as follows: $Q = (\rho, s, G) \doteq (\rho', s', G') = Q'$ if $G \doteq G'$ and

- for all x in χ_m, $\rho(x) \doteq \rho'(x)$
- either $s = s' = \varepsilon$ or $s = u :: s_t$, $s' = u' :: s'_t$, $u \doteq u'$ and $s_t \doteq s'_t$.

Proposition 5.6 *For any concrete state Q, for any abstract states $\overline{Q}$, $\overline{Q}'$, if $Q \lhd^{\alpha} \overline{Q}$ and $\overline{Q} \doteq \overline{Q}'$ then $Q \lhd^{\alpha} \overline{Q}'$.*

Proof. Let $Q = ((i, \rho, s) :: fr, G)$, $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$ and $\overline{Q}' = (\overline{\rho}', \overline{s}', \overline{G}')$. From the definition 4.6 of state abstraction, $Q \lhd^{\alpha} \overline{Q}'$, if $s \lhd^{\alpha} \overline{s}', \rho \lhd^{\alpha} \overline{\rho}'$ and $G \lhd^{\alpha} \overline{G}'$. By hopthesis, $Q \lhd^{\alpha} \overline{Q}$ thus $s \lhd^{\alpha} \overline{s}, \rho \lhd^{\alpha} \overline{\rho}$ and $G \lhd^{\alpha} \overline{G}$. Moreover $\overline{Q} \doteq \overline{Q}'$ thus $\overline{s} \doteq \overline{s}', \overline{\rho} \doteq \overline{\rho}'$ and $\overline{G} \doteq \overline{G}'$.
Let us first prove that $G \lhd^{\alpha} \overline{G}'$. From $\overline{G} \doteq \overline{G}'$ we have $\Pi_{Obj}(\overline{G}) = \Pi_{Obj}(\overline{G}')$. Since $G \lhd^{\alpha} \overline{G}$, $\alpha(\Pi_{Obj} G) \subseteq \Pi_{Obj}\overline{G} = \Pi_{Obj}\overline{G}'$. Thus $G \lhd^{\alpha} \overline{G}'$.
Let us now prove that $s \lhd^{\alpha} \overline{s}'$. Let $s = u :: s_t$, $\overline{s} = \overline{u} :: \overline{s}_t$ and $\overline{s}' = \overline{u}' :: \overline{s}'_t$. As $s \lhd^{\alpha} \overline{s}$, thus $s_t \lhd^{\alpha} \overline{s}_t$ and $(\alpha(G.\varsigma^{-1}(u)), \mathbf{d}) \in \overline{u}$ if $u \in G.V^{Obj}$. If $u \in G.V^{Obj}$, then $\alpha(G.\varsigma^{-1}(u)) \in \overline{G}.V^{Obj}$.
By hpothesis, $\overline{s} \doteq \overline{s}'$. From the definition of $\overline{s} \doteq \overline{s}'$ if $\overline{s} = \overline{s}' = \epsilon$ or, $\overline{u} \doteq \overline{u}'$ and $\overline{s}_t \doteq \overline{s}'_t$. Moreover, the two sets of typed-flow nodes, $\overline{u}$ and $\overline{u}'$, are equal accoding to $\doteq$ relation if $\{e \mid \langle e, \mathbf{d}\rangle \in \overline{u} \wedge e \in \overline{G}.V^{Obj}\} = \{e \mid \langle e, \mathbf{d}\rangle \in \overline{u}' \wedge e \in \overline{G}'.V^{Obj}\}$. Thus, if $\alpha(G.\varsigma^{-1}(u)) \in \overline{G}.V^{Obj}$ and $(\alpha(G.\varsigma^{-1}(u)), \mathbf{d}) \in \overline{u}$ then $(\alpha(G.\varsigma^{-1}(u)), \mathbf{d}) \in \overline{u}'$. Hence $s \lhd^{\alpha} \overline{s}'$.
The proof for local variables array is similar. □

Proposition 5.7 *For all states Q_1, Q'_1, Q_2, Q'_2,*

- *if $Q_1 \doteq Q_2$ and $Q'_1 \doteq Q'_2$ then $Q_1 \sqcup Q'_1 \doteq Q_2 \sqcup Q'_2$,*
- *for any bytecode b, for any Γ_1, Γ_2, if $Q_1 \doteq Q_2$ then*

$$\overline{instr}_b(Q_1, \Gamma_1) \doteq \overline{instr}_b(Q_2, \Gamma_2)$$

Proof. For the first point, as $\sqcup$ is defined component-wise for stacks and mappings ρ, the only part to be proved is for graphs.
If $Q_1.G \doteq Q_2.G$ then the two graphs are equal when restricted to their object part, thus $\Pi_{Obj}(Q_1.G) = \Pi_{Obj}(Q_2.G)$. Moreover, $\Pi_{Obj}(Q'_1.G) = \Pi_{Obj}(Q'_2.G)$, hence $\Pi_{Obj}(Q_1.G) \cup \Pi_{Obj}(Q'_1.G) = \Pi_{Obj}(Q_2.G) \cup \Pi_{Obj}(Q'_2.G)$.
We must prove that $\Pi_{Obj}(Q_1.G \cup Q'_1.G) = \Pi_{Obj}(Q_2.G \cup Q'_2.G)$. It is enough to prove that, for two dependency graphs G and G, $\Pi_{Obj}(G \cup G') = \Pi_{Obj}(G) \cup \Pi_{Obj}(G')$.
The proof lies on the property of dependency graphs stating that any node $u \in G.V^{Val}$ is a leaf. Thus, a dependency graph contains only edges between two objects or edges from an object to a value: there is no edge from a value to an object node. When making the union of two dependency graphs, the only edges between objects can come from the two dependency graphs. Thus, give restriction to objects part of the union of two graphs is equal to the union of the restrictions of the two graphs to objects: $\Pi_{Obj}(G \cup G') = \Pi_{Obj}(G) \cup \Pi_{Obj}(G')$.

For the second point, we make a case analysis on each bytecode instruction: details can be found in Appendix B.2.

□

Let us consider the system of equations $\mathcal{E}_B$. Each right-hand side can be seen as a functional over (Q_i, Γ_i), the unknowns. Let us define the system $\mathcal{E}'_B$ with $(Q_0, \varnothing)$ for initial state:

$$(Q_i, \Gamma_i) = \begin{cases} (Q_0 \sqcup_{j \in pred(i)} \overline{instr}_{B[j]}(Q_j, \Gamma_j), & \text{if } i = \mathsf{Entry}(B) \\ \quad \{u \mid \langle u, t\rangle \in v \text{ with } Q_k = (\rho, v :: s, G), k \in cxt(i)\}) & \\ (\sqcup_{j \in pred(i)} \overline{instr}_{B[j]}(Q_j, \Gamma_j), & \text{otherwise} \\ \quad \{u \mid \langle u, t\rangle \in v \text{ with } Q_k = (\rho, v :: s, G), k \in cxt(i)\}) & \end{cases}$$

We define for a block B and the system $\mathcal{E}'_B$, the mapping $\overline{instr}'_B(Q, \Gamma)$ as we did for $\overline{instr}_B(Q, \Gamma)$ and $\mathcal{E}_B$.

Obviously, for the least solution of $\mathcal{E}'_B$, we have that $\Gamma_i = \varnothing$ for all i. Therefore, the functionals in the right-hand side and thus, the solution of the system are independent from the Γ'. As Lemma 5.4 establishes the local soundness for all Γ's, in particular for $\varnothing$, we have that the global soundness holds for the system $\mathcal{E}'_B$, that is

$$Q \lhd^\alpha \overline{Q} \text{ implies } instr_B(Q) \lhd^\alpha \overline{instr}'_B(\overline{Q}, \varnothing) \tag{5.1}$$

However, the system $\mathcal{E}_B$ involves the Γ's which influence on the values of the Q's but not on their object part as we show now.

Proposition 5.8 *Provided that their respective initial states satisfies $Q \doteq Q'$, then for the respective least solution of $\mathcal{E}_B$ and $\mathcal{E}'_B$, we have that $Q_i \doteq Q'_i$ for all i.*

Proof. We rely here on the iterative construction of the least solution of equation systems. Starting from the valuation assigning the least value of the lattice to all the unknowns, that is the Q_i, Γ_i's and the Q'_i, Γ'_i's respectively, we compute a new value for these valuations (at the left-hand side of equalities) by using the old values in the right-hand side. Eventually, a fixed point is reached during that computation.
Then, the proof goes by induction on n, the minimal number of iteration required to reach a fixed point in both system using Proposition 5.7 for the induction step. □

Proposition 5.9 *For any block B, for any Γ_1, Γ_2, if $Q_1 \doteq Q_2$ then $\overline{instr}_B(Q_1, \Gamma_1) \doteq \overline{instr}'_B(Q_2, \Gamma_2)$*

Proof. Straightforward from Propositions 5.8 and 5.7. □

By combining the global soundness (5.1), Propositions 5.9 and 5.6, we get Proposition 5.5 as result. This ensures correctness for dependency graph restricted to references. The points-to analysis is a flow-sensitive may-analysis, similar to previous work [18, 20]. In the next subsection, we prove the correctness of the primitive edges (implicit and direct flow) as a non-interference theorem relying on the correctness for the points-to analysis.

5.2 Analysis correctness

To prove the non-interference theorem, according to Definition 3.3, we need to make values vary at some program point according to Definition 3.2 and to check the impact of this variation on objects at another program point. Thus, we first define the notion of *state variation* that captures how a concrete execution state, corresponding to a program point i, might change when a value η has changed in the past of this execution. Definition 5.13 contains an over estimation of the set of states that the JVM can reach after this change: the main impact is on the memory graph, but the local variables and stack can also be affected. The correctness of the definition is proved later in Proposition 5.15.

Definition 5.10 (Graph variation) *Let α be an abstraction function. Let $Q = ((i, \rho, s) :: fr, G)$ be a concrete state, $\overline{Q}$=$(\overline{\rho}, \overline{s}, \overline{G})$ such that $Q \lhd^\alpha \overline{Q}$. Let $\beta \subseteq \overline{G}.V^{Val}$ be a set of nodes of primitive type. Then G' is a **graph variation** of G with respect to $\overline{G}$ and β, denoted $G' = \upsilon(G, \overline{G}, \beta)$, if it is obtained from G applying one of the following rules:*

1. *$G' = G[v \mapsto x]$ with $x \in Val$, if $\exists (u, v, f) \in G.E$, $\exists \overline{\eta} \in \beta$ and $t \in \mathcal{F}$ such that $(\alpha(u), \overline{\eta}, \langle f, t\rangle) \in \overline{G}.E$*

2. *$G' \lhd^\alpha \overline{G}$ st for all $v \in G.V^{Obj}$ if $free_\alpha(v, \beta, Q)$ then $v \in G'.V^{Obj}$ and for all (u, v, f) in $G.E$, with $u \in G'.V^{Obj}$, if $\beta \cap adj_{\overline{G}}(\alpha(u), f) = \emptyset$, then $(u, v, f) \in G'.E$.*

Definition 5.11 (Stack variation) *Let α be an abstraction function. Let $Q = ((i, \rho, s) :: fr, G)$ be a concrete state, $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$ such that $Q \lhd^\alpha \overline{Q}$. Let $\beta \subseteq \overline{G}.V^{Val}$ be a set of nodes of primitive type. Then s' is a **stack variation** of s with respect to $\overline{s}$ and β, denoted by $s' = \upsilon(s, \overline{s}, \beta)$, if it is obtained from s applying one of the following rules:*

1. *$s' = s[i \mapsto x]$ with $x \in$ Val, if $s[i] \in$ Val and $\exists \overline{\eta} \in \beta, t \in \mathcal{F}$ such that $(\overline{\eta}, t) \in \overline{s}[i]$*

2. *$s' = s[i \mapsto \varsigma(o)]$, if $\exists \overline{\eta} \in \beta$ such that $(\overline{\eta}, \boldsymbol{i}) \in \overline{s}[i] \wedge (\overline{o}, \boldsymbol{d}) \in \overline{s}[i] \wedge o \in G'.V^{Obj} \wedge o \in \alpha^{-1}(\overline{o})$*

Definition 5.12 (Local variables array variation) *Let α be an abstraction function. Let $Q = ((i, \rho, s) :: fr, G)$ be a concrete state, $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$ such that $Q \lhd^\alpha \overline{Q}$. Let $\beta \subseteq \overline{G}.V^{Val}$ be a set of nodes of primitive type. Then ρ' is a **local variables array variation** of ρ with respect to $\overline{\rho}$ and β, denoted by $\rho' = \upsilon(\rho, \overline{\rho}, \beta)$, if it is obtained from ρ applying one of the following rules:*

1. *$\rho' = \rho[i \mapsto x]$ with $x \in Val$, if $\rho(i) \in Val$ and $\exists \overline{\eta} \in \beta$, $t \in \mathcal{F}$ such that $(\overline{\eta}, t) \in \overline{\rho}(i)$*
2. *$\rho' = \rho[i \mapsto \varsigma(o))]$, if $\exists \overline{\eta} \in \beta$ such that $(\overline{\eta}, \boldsymbol{i}) \in \overline{\rho}(i) \wedge (\overline{o}, \boldsymbol{d}) \in \overline{\rho}(i) \wedge o \in G'.V^{Obj} \wedge o \in \alpha^{-1}(\overline{o})$*

Based on the definition of stack variation and local variables array variation, we redefine the state variation:

Definition 5.13 (State variation) *Let α be an abstraction function. Let $Q = (\rho, s, G)$ be a concrete state, $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$ an abstract state such that $Q \lhd^{\alpha} \overline{Q}$. Let $\beta \subseteq \overline{G}.V^{Val}$ be a set of nodes of primitive type. Then Q' is a* ***state variation*** *of Q with respect to $\overline{Q}$ and β, denoted by $Q' = ((i, \rho', s') :: fr, G') = \upsilon(Q, \overline{Q}, \beta)$, if $G' = \upsilon(G, \overline{G}, \beta)$, $s' = \upsilon(s, \overline{s}, \beta)$ and $\rho' = \upsilon(\rho, \overline{\rho}, \beta)$.*

A state variation captures the impacts of the modification of a value in the past of the execution on the current state. The main impact we are interested in concerns the changes of the memory graph according to Definition 5.10. Anyway, the modification of a value in the past of a state also impacts the stack and the local variables, according to definitions 5.11 and 5.12. Rule 1 of Definition 5.11 and rule 1 of Definition 5.12 capture possible variations due to immediate use of a value corresponding to β while rule 2 of Definition 5.11 and rule 2 of Definition 5.12 capture the impact of the changes of a value corresponding to β to the objects that depend on it by an implicit flow.

From the definition of state variation, we can state that the variation of a state regarding to a set β, does not impact the objects ω which have no dependence to any node of β.

Lemme 5.14 *Let α be an abstraction function, $Q = ((i, \rho, s) :: fr, G)$, and $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$ with $Q \lhd^{\alpha} \overline{Q}$. Let $\beta \subseteq \overline{G}.V^{Val}$, and $\omega \in G.V^{Obj}$ such that there exists no path from $\alpha(\omega)$ to any node of β in $\overline{G}$. Then, in any state variation $Q' = \upsilon(Q, \overline{Q}, \beta) = ((i, \rho', s') :: fr, G')$, if $\omega \in G'.V^{Obj}$, we have $G\lfloor\omega\rfloor \equiv G'\lfloor\omega\rfloor$.*

Proof. The differences between G and G' are given by rules of Definition 5.10. If $\omega \in G'.V^{Obj}$, then, for any object node v of the subgraph rooted by ω, there exists an edge $(u, v, f) \in G.E$ such that u belongs to the subgraph rooted by ω with:

- $\beta \cap adj_{\overline{G}}(\alpha(u), f) = \emptyset$ because $Reach_{\overline{G}}(\alpha(\omega)) \cap \beta = \emptyset$,
- $u \in G'.V^{Obj}$ because either $u = \omega$, or u belongs to $G'.V^{Obj}$ for the same reason.

Thus, $v \in G'.V^{Obj}$ and $(u, v, f) \in G'.E$
For the leaves of the subgraph rooted by ω, rule 1 randomizes the value of v if there is an edge $(u, v, f) \in G.E$ and a flow from $\alpha(u)$ to a element of β labelled by f in $\overline{G}$. As by hypothesis, there is no path from $\alpha(\omega)$ to any element of β in in $\overline{G}$, it is obvious that this modification does not affect the subgraph rooted by ω.

□

Proposition 5.15 states the state variation correctness, by claiming that changing an input value and executing a block with a state variation leads us to a state variation.

Proposition 5.15 (State variation correctness) *Let B be an instruction block, Q_1 a concrete state, $\overline{Q_1}$ with $Q_1 \lhd^{\alpha} \overline{Q}_1$, $Q_2 = instr_B(Q_1)$, and $\overline{Q_2} = \overline{instr}_B(\overline{Q_1}, \Gamma_1)$.*

If $Q'_1 = \upsilon(Q_1, \overline{Q_1}, \beta)$ and $Q'_2 = instr_B(Q'_1)$ then $Q'_2 = \upsilon(Q_2, \overline{Q_2}, \beta)$.

Proof. *Proof.* We split the proof in two parts: first we prove the theorem holds for blocks containing a single instruction, this is done by case analysis in Appendix B.3. Let us prove now the state variation correctness for "structured" blocks. The grammar $\mathcal{G}$ defines precisely what are "structured" blocks and will be used in this proof. The proof goes by induction on k the number of steps required to produce the control flow graph of the block B by the grammar $\mathcal{G}$. Let G be a CFG (that does not contain the non-terminal B anymore) for some block B such that $\mathsf{B} \rightarrow^k G$ for some natural k. Let us show that G (and thus, B) satisfies the state variation property.

For $k = 1$: Only production rules (bytecode) are concerned. In this case, we rely on the correctness established for each bytecode in the previous section.

For k, the property holding for all $k' < k$: in this case, there exists some graph G' such that $\mathsf{B} \rightarrow G' \rightarrow^{k-1} G$. Depending on the rule used for $\mathsf{B} \rightarrow G'$: we analyse the different cases in Appendix B.4.

□

We can now conclude with the proof of the non-interference theorem.

Proof of Theorem 5.3. Let B be an instruction block. Let $Q_1 = ((i, \rho_1, s_1) :: fr_1, G_1)$ be a concrete state, $\overline{Q_1} = (\rho_1, s_1, G_1)$ such that $Q_1 \lhd^\alpha \overline{Q_1}, Q_2 = instr_B(Q_1)$, and $\overline{Q}_2 = \overline{instr}_B(\overline{Q}_1, \Gamma_1)$. Let $\omega \in G_1.V^{Obj}$ be an object node and $\eta \in \mathcal{I}(Q_1)$ be an input value.
Let us consider a value-change $Q_1' = ((i, \rho_1', s_1) :: fr_1, G_1')$ of Q_1 and η. According to Definition 3.2, either η is a value node of G_1 and then $G_1' = G_1[\eta \mapsto v]$, or η is a local variable and then $\rho_1' = \rho_1[\eta \mapsto v]$ (for some $v \in Val$). Let us denote $\beta = \alpha_{Q_1}(\eta)$, then, the first case corresponds to the rule 1 of Definition 5.10, the second case to the rule 1 of Definition 5.13, thus $Q_1' = v(Q_1, \overline{Q_1}, \beta)$. We denote $Q_2' = instr_B(Q_1')$. According to Proposition 5.15, we have $Q_2' = v(Q_2, \overline{Q_2}, \beta)$. As $free_\alpha(\omega, \alpha_{Q_1}(\eta), Q_1)$, rule 2 of Definition 5.10 says that $\omega \in G_1'.V$, thus $\omega \in G_2'.V$. As by hypothesis, $Reach_{\overline{G_2}}(\alpha(\omega)) \cap \alpha_{Q_1}(\eta) = \emptyset$, by applying Lemma 5.14, we obtain $G_2\lfloor\omega\rfloor \equiv G_2'\lfloor\omega\rfloor$ (for G_2, G_2' the memory graphs of resp. Q_2, Q_2'), ie η does not interfer with ω in B. □

6 Inter-method analysis

In the previous section we show non-interference for methods without invocation. That is to say, for a method `m` that does not contain any invocation, we know that the non-interference theorem holds. In this section, we add a support for a context-insensitive or a context-sensitive analysis with the same semantics rules. The differences between the two ones are the construction of the inter-procedural control flow graph and the node naming strategy.

6.1 Abstract semantics of the method invocation

To deal with method invocation, we add abstract semantics rules for `invoke` and `areturn` bytecodes. In static analysis of inter-procedural programs, one of the main problems is that the size of the call stack in the concrete execution is not bounded: the abstract rule for `invoke` cannot sketch the operational one otherwise, we would have an infinite abstract domain. Some works of the literature propose an abstraction of the call stack [12]. We propose to keep the same abstract domain than in the intra-method analysis, in which we occult the call stack and only consider the current frame of the method: the method invocation can be added to the framework used. Thus, the new abstract semantics rules are given by Figure 8 where ζ is a special value.

$$\frac{(\rho, u_{n_m} :: \cdots :: u_0 :: s, G)}{(\{0 \mapsto u_0, \ldots, n_m \mapsto u_{n_m}\}, \epsilon, G)}\,\texttt{invoke m} \qquad \frac{(\rho, u :: s, G)}{(\zeta, u, G)}\,\texttt{areturn}$$

Figure 8: Abstract semantics rules for method invocation

Using these rules, we do not registered the current context before the call and use instead a bytecode transformation before the analysis. We replace each `invoke` bytecode by an ”`if`” region as described by Figures 9(a) and 9(b).

This transformation is syntactical: we insert some code and rename the following bytecode numbers with a function h. It does not change the semantics of the program since we only add dead code: bytecodes b_0 to b_{n_m+2} will never be executed. We obtain the control flow graph for which it is obvious to see that any concrete execution $Q_0Q_1 \ldots Q_{i-1}Q_iQ_{P_m[0]} \ldots Q_{P_m[j]}Q_{i+1}$ of the original program correspond to an execution $Q_0Q_1 \ldots Q_{i-1}Q_iQ_{a_0}Q_{c_0}Q_{P_m[0]} \ldots Q_{P_m[j]}Q_{h(i+1)}$ with $Q_{i+1} = Q_{h(i+1)}$. Thus, if we obtain an abstract state $\overline{Q}_{h(i+1)}$ that is correct for $Q_{h(i+1)}$, it is also correct for Q_{i+1}. Moreover, we keep the usual Java properties: the stack is well typed and has always the same size at each program point.

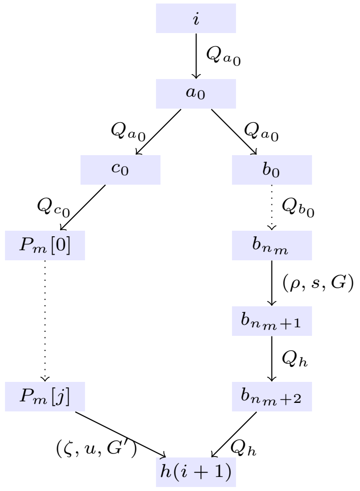
i : `invoke m`
$i+1$: `...`

(a) Original code

i : `iconst_0`
a_0 : `ifeq` c_0
b_0 : `pop`
: `...`
b_{n_m} : `pop`
b_{n_m+1} : `aconst_null`
b_{n_m+2} : `goto` $h(i+1)$
c_0 : `invoke m`
$h(i+1)$: `...`

(b) Transformed code

$$\begin{aligned} Q_{a_0} &= (\rho, u_{n_m} :: \cdots :: u_0 :: s, G) \\ Q_{b_0} &= (\rho, u_{n_m-1} :: \cdots :: u_0 :: s, G) \\ Q_{c_0} &= (\{0 \mapsto u_0, \ldots, n_m \mapsto u_{n_m}\}, \epsilon, G) \\ Q_h &= (\rho, \texttt{null} :: s, G) \end{aligned}$$



(c) Transformed Control Flow

Figure 9: Bytecode transformation

This transformation is only used for technical reasons in the analysis of the abstract model: the path $b_0 \ldots b_{n_m+2}$ (that is never executed in any concrete execution) is used to ”transmit” the current context we had before the invocation to the immediate successor of the invocation, thus sketching the context save and reload that occur the concrete execution of an invocation. The equivalent of a context reload is done by the join operator when computing $\overline{Q}_{h(i+1)}$. We extend the operator to the case where the local variables have the special value ζ: $(\zeta, u, G) \sqcup (\rho, v :: s, G') = (\rho, v :: s, G') \sqcup (\zeta, u, G) = (\rho, u :: s, G \sqcup G')$, thus $\overline{Q}_{h(i+1)} = (\rho, u :: s, G')$ (since $G \sqcup G' = G'$), which contains the local variables of $\overline{Q}_{a_0}$, the stack of $\overline{Q}_{a_0}$ increased with the return value of `m`, and the memory resulting from the execution of `m`.

This is exactly the state we would obtain by saving the calling context and reloading it after the method execution. Note that the extension of the join operator does not affect the previous uses of the operator since we only introduce the special value ζ in the `areturn` rule. Moreover to have a complete definition of $\sqcup$, we can define the last case: $(\zeta, u, G) \sqcup (\zeta, u', G') = (\zeta, u \sqcup u', G \sqcup G')$.

Thus, we perform an analysis *without any abstraction of the call stack*, and keep the same finite abstract domain as previously. We now have to define the inter-procedural control flow graph that we use.

6.2 Inter-procedural control flow graph

We use the same framework and the same equation system than for the intra-method analysis to perform a context insensitive or several context sensitive inter-method analyses. We only have to define the $pred$ function, that is to say the inter-procedural control flow graph (ICFG^i): the parameter i of this graph defines the precision of the analysis.

To built the ICFG, we have to extend our naming strategy to keep disjoint instruction number sets for methods and a unique node name in the dependency graph to respect the allocation site model. Thus, each instruction i of a method `m` is now named `m`^{i} (with `m` the fully qualified name of the method and $m' \leq m$ if m' is a method that overwrites m.), and we use copies of method call graphs including informations about the call sites. For example the node $m_1^i m_2^j m_3^k$ of the ICFG^3 denotes the k^{th} bytecode of method m_3 that has been called in the context of line j of method m_2, that has itself been called at line i of m_1. The larger the parameter of the ICFG is, the more accurate is the analysis but the larger is the number of states of the ICFG. Then, the ICFG^k of a set of methods $\mathcal{M}$ is a graph with:

$$\begin{array}{rcl} ICFG^k_{\mathcal{M}}.V & = & \{m_0^{i_0} m_1^{i_1} \ldots m_k^{i_k} \mid \forall 0 \leq i \leq k, m_i \in \mathcal{M}, i_k \in P_{m_k}, \forall 0 \leq j < k, \\ & & P_{m_j}[i_j] = \texttt{invoke}\, m_{j+1}\} \end{array}$$

$$\begin{array}{rcl} ICFG^k_{\mathcal{M}}.E & = & \{(m_0^{i_0} \ldots m_{k-1}^{i_{k-1}} m_k^i, m_0^{i_0} \ldots m_{k-1}^{i_{k-1}} m_k^j) \mid (i,j) \in CF_{m_k}.E, \\ & & P_{m_k}[i] \neq \texttt{invoke m}\} \\ & \cup & \{(m_0^{i_0} \ldots m_{k-1}^{i_{k-1}} m_k^i, {m'}_0^{j_0} \ldots {m'}_{k-1}^{j_{k-1}} {m'}^0) \mid P_{m_k}[i] = \texttt{invoke m}, m' \leq m\} \\ & \cup & \{((m_0^{i_0} \ldots m_{k-1}^{i_{k-1}} m_k^i, {m'}_0^{j_0} \ldots {m'}_k^{j_k}) \mid P_{m_k}[i] = \texttt{areturn}, \\ & & P_{m'_k}[j_k - 1] = \texttt{invoke m}, m_k \leq m\} \end{array}$$

The ICFG^0 produces a context-insensitive analysis, ICFG^1 a usual context-sensitive analysis, and so on. Note that in node names of the dependency graph like n_{pc}^c and n_{pc}^n, pc now encodes the name of the node of the ICFG in which the constant or new object is created. Note that our construction also takes into account overwritten methods, it can be more precise when the dynamic types of objects can statically be computed by only branching on the first instruction of the exact method.

6.3 Soundness

In order to show that the extended abstract model is sound with respect to the non-interference, we prove that the abstract semantics rules for `invoke` and `areturn` have the same properties as the rest of bytecode. We only have to define an extension of the relation $\lhd^\alpha$ for an abstraction relation α taking into account the new value ζ.

Definition 6.1 (State abstraction (extension)) *Let α be an abstraction function. Given a concrete state $Q = ((pc, \rho, s) :: fr, G)$ and an abstract state $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$, $\overline{Q}$ is an α-abstraction of Q (denoted by $Q \lhd^\alpha \overline{Q}$) if $G \lhd^\alpha \overline{G}$ and:*

- *$v \lhd^\alpha \overline{s}$ with $s = v :: s'$ if $\overline{\rho} = \zeta$*
- *$s \lhd^\alpha \overline{s}$, and $\rho \lhd^\alpha \overline{\rho}$, otherwise.*

The extension is necessary for the `areturn` bytecode, in order to prove the relation $\lhd^\alpha$ for the value returned by the invoked method. This definition is sufficient, since the relation $\lhd^\alpha$ for the rest of the calling context is being insured by the code transformation presented in Subsection 6.1.

For the same reasons, we need to define an extension for the state variation definition: for the `areturn` bytecode, and thus the special value ζ, the state variation must hold only for the returned value.

Definition 6.2 (State variation (extension)) *Let α be an abstraction function. Let $Q = ((pc, \rho, s) :: fr, G)$ be a concrete state, $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$ its S_0-may-abstraction such that $Q \lhd^\alpha \overline{Q}$. Let $\beta \subseteq \overline{G}.V^{Val}$ be a set of nodes of primitive type. Then Q' is a **state variation** of Q with respect to $\overline{Q}$ and β, denoted by $Q' = ((i, \rho', s') :: fr, G') = \upsilon(Q, \overline{Q}, \beta)$, if $G' = \upsilon(G, \overline{G}, \beta)$ and:*

- *$v' = \upsilon(v, \overline{s}, \beta)$ with $s = v :: s''$ and $s' = v' :: s'''$ if $\overline{\rho} = \zeta$*
- *$s' = \upsilon(s, \overline{s}, \beta)$, and $\rho' = \upsilon(\rho, \overline{\rho}, \beta)$, otherwise.*

The detailed proof is in Appendix C.

7 Secure information flow

All the previous works on non-interference require the lattice security model of information flow to be known from the beginning. The dependency graph contains points-to and control flow dependency information. Once computed, the dependency graph can be labeled with security levels and non-interference checked. We can consider any kind of "programs", let us for example consider a method `m` (with n parameters), for which we want to check non-interference. We have to define a initial abstract state $\overline{Q}^{\mathtt{m}}_{init} = (\{0 \mapsto n_0^p, \ldots, n \mapsto n_n^p\}, \epsilon, \overline{G}_{init})$ for the analysis such that $\overline{G}_{init}.V$ contains all the nodes required by the allocation site model, constants and `null`value, nodes representing static variables, and nodes representing formal parameters. For each parameter i, n_i^p denotes the i^{th} parameter of the method. We add to $\overline{G}_{init}$ the minimal subgraph rooted by n_i^p, $\overline{G}_i^p$. This graph represents the parameter and all of its content. To avoid infinite graphs, we use a parameter h, which is called the height of the analysis, and represents how depth we unfold the recursive data structures. The graph G_i^p is the minimal graph that contains n_i^p such that $\forall u \in \overline{G}_i^p.V^{Obj}$, for each field f_1 of $Type(u)$:

- either there exists a path $u^1_1 u^1_2 \ldots u^1_k u^2_1 u^2_2 \ldots u^2_k \ldots u^h_1 u^h_2 \ldots u^h_k u$ labeled by $(\langle f_1, \mathbf{d}\rangle\langle f_2, \mathbf{d}\rangle\langle f_3, \mathbf{d}\rangle \ldots \langle f_k, \mathbf{d}\rangle)^h$ in G^p_i such that $\forall 1 \leq i \leq h, 1 \leq j \leq h$, $Type(u^i_l) = Type(u^j_l)$, then $(u, u^h_1, \langle f_1, \mathbf{d}\rangle) \in G^p_i.E$.
- or $u.f_1 \in G^p_i.V$ and $(u, u.f_1, \langle f, \mathbf{d}\rangle) \in G^p_i.E$.

Then, we use that description of ”formal” initial state. Let $(\mathcal{L}, \sqcup, \sqsubseteq)$ be a lattice of security levels; for example, $\mathcal{L} = \{low, high\}$. Then, we can define security levels on types: let $\lambda_t : \mathit{Fields} \times \mathit{Class} \rightarrow \mathcal{L}$ be a function that associates a security level with a field of a class. A security function is a function that associated security levels to inputs values and to return values (denoted by the set Ret): $\lambda_i : \bigcup_{0 \leq i \leq n} \overline{G}^p_i.V^{\,Val} \cup Ret \rightarrow \mathcal{L}$ such that if v is the field f of a class C, then $\lambda_i(v) = \lambda_t(f, C)$.

We say that a method is secure if values accessible from parameters, return value or static variables on paths contain at least a field of higher security level than their own.

Theorem 7.1 (Secure information flow) *Let m be a method, and $\lambda_{\ i}$ a security function. Let $\overline{Q} = (\overline{\rho}, \overline{u} :: \overline{s}, \overline{G}) = \overline{instr}_{\mathsf{m}}(\overline{Q}_{init}, \epsilon)$, and $Ret = \{v \in \overline{G}.V \mid \exists (v, t) \in \overline{u}\}$. The method has secure information flow with respect to $\mathcal{L}$ if for every node $v \in \overline{G}.V^{\,Val}$ and every path $o_0 \overset{\langle f_1, t_1\rangle}{\rightarrow} o_1 \ldots o_k \overset{\langle f_k, t_k\rangle}{\rightarrow} o_{k+1}$ with $o_0 \in Ret \cup \{v \in \overline{G}.V \mid= n^s_{-1} \vee v = n^p_i\}$, $\exists i$ such that $\lambda_i(v) \sqsubseteq \lambda_t(f_i, \mathit{Type}(o_i))$.*

The advantage of our approach is that security annotations must not be known a priori. Changing a security level does not require a new analysis. Note that we can also have more precise policies on instances: if o and o' have the same type, $o.f$ and $o'.f$ can be given different security levels using a function $\lambda_e : \overline{G}_{init}.E \rightarrow \mathcal{L}$ instead of λ_t. It is more precise but require the user to precise all the policies.

8 Conclusion

Information flow analysis aims to avoid that programs leak confidential information: observable/public outputs of a program must not disclose information about secret/private values manipulated by the program. Motivated by information flow analysis for Java bytecode, we propose an algorithm to compute dependency graphs, which tracks, at different program points, how input values may influence the outputs. Such a graph is a points-to graph extended with primitive values and flows arising from the control structure of a program. We prove the soundness of our construction by giving a non-interference theorem: a node a is related to a node b if the value of b may influence the value of a.

In contrast with usual information flow techniques, our approach is flow-sensitive and the security levels are not required to be known during the graph computation. Changing a security level does not require reanalyzing of the code.

Our main goal is to provide an algorithm which can be use for information flow. In the same time, as the dependency graph includes the points-to graph, our framework can be exploited for any points-to application in program analysis, such as escape analysis and optimization.

Further works includes increasing of accuracy by considering strong update or path-sensitivity in the analysis and studying relationships with existing information flow analysis by considering further abstractions of our analysis algorithm. Actual work concentrates on making our analysis compositional, which will better fit the Java paradigm of dynamic class loading.

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] Marco Avvenuti, Cinzia Bernardeschi, and Nicoletta De Francesco. Java bytecode verification for secure information flow. *ACM SIGPLAN Notices*, 38(12):20–27, 2003.

[3] Thomas Ball. What's in a region?: or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Programming Languages and Systems*, 2(1-4):1–16, 1993.

[4] Gilles Barthe, Amitabh Basu, and Tamara Rezk. Security types preserving compilation: (extended abstract). In Bernhard Steffen and Giorgio Levi, editors, *Proc. 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 2–15, Venice, Italy, 2004. Springer.

[5] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. In *Proc. 16th European Symposium on Programming (ESOP 2007)*, volume 4421 of *Lecture Notes in Computer Science*, pages 125–140, Braga, Portugal, 2007. Springer.

[6] Cinzia Bernardeschi, Nicoletta De Francesco, Giuseppe Lettieri, and Luca Martini. Checking secure information flow in java bytecode by code transformation and standard bytecode verification. *Software – Practice and Experience*, 34(13):1225–1255, 2004.

[7] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, and Harini Srinivasan. Dependence analysis for Java. In *Proc. 12th International Workshop on Languages and Compilers for Parallel Computing (LCPC '99)*, pages 35–52, London, UK, 2000. Springer.

[8] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[9] Samir Genaim and Fausto Spoto. Information Flow Analysis for Java Bytecode. In *Proc. of 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362, Paris, France, 2005. Springer.

[10] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[11] Sebastian Hunt and David Sands. On flow-sensitive security types. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 79–90. ACM Press, 2006.

[12] Bertrand Jeannet and Wendelin Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In Charles Rattray, Savi Maharaj, and Carron

Shankland, editors, *Proc. 10th International Conference of Algebraic Methodology and Software Technology (AMAST 2004)*, volume 3116 of *Lecture Notes in Computer Science*, pages 258–273, Stirling, Scotland, UK, 2004. Springer.

[13] Naoki Kobayashi and Keita Shirane. Type-based information flow analysis for low-level languages. In *Proc. 3rd Asian Workshop on Programming Languages and Systems (APLAS'O2)*, pages 302–316, Shanghai, China, 2002.

[14] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[15] Andrew C. Myers. Jflow: practical mostly-static information flow control. In *Proc. 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'99)*, pages 228–241. ACM Press, 1999.

[16] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[17] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[18] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices*, 36(7):12–23, 2001.

[19] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[20] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proc.ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, volume 34 of *ACM SIGPLAN Notices*, pages 187–206, Denver, Colorado, 1999. ACM Press.

Contents

A Proof of Lemma 4.2 (Monotonicity of the transformation rules)

We consider two pairs (Q_1, Γ_1) and (Q_2, Γ_2) from the property space $\mathcal{S}$ such that $(Q_1, \Gamma_1) \sqsubseteq (Q_2, \Gamma_2)$, and let $Q'_1 = \overline{instr}_b(Q_1, \Gamma_1)$ and $Q'_2 = \overline{instr}_b(Q_2, \Gamma_2)$. We prove that $Q'_1 \sqsubseteq Q'_2$.

We make a proof by case analysis on each instruction b, based on the transformation rules in Figure 6.

Case: $b = \texttt{prim op}$

Let $Q_1 = (\rho_1, v_1 :: v'_1 :: s_1, G_1)$ and $Q_2 = (\rho_2, v_2 :: v'_2 :: s_2, G_2)$. Since $Q_1 \sqsubseteq Q_2$, we also have $\rho_1 \sqsubseteq \rho_2$, $s_1 \sqsubseteq s_2$, $G_1 \subseteq G_2$, $v_1 \subseteq v_2$ and $v'_1 \subseteq v'_2$. Hence, $(v_1 \cup v'_1) \subseteq (v_2 \cup v'_2)$. From the transformation rule, $Q'_1 = (\rho_1, v_1 \cup v'_1 :: s_1, G_1)$ and $Q'_2 = (\rho_2, v_2 \cup v'_2 :: s_2, G_2)$, and as the partial order relation holds on each component, we can conclude with $Q'_1 \sqsubseteq Q'_2$.

Case: $b = \texttt{pop}$ **or** $b = \texttt{ifeq}\ a$

Let $Q_1 = (\rho_1, v_1 :: s_1, G_1)$ and $Q_2 = (\rho_2, v_2 :: s_2, G_2)$, $Q'_1 = (\rho_1, s_1, G_1)$ and $Q'_2 = (\rho_2, s_2, G_2)$. From hypothesis, $(\rho_1, v_1 :: s_1, G_1) \sqsubseteq (\rho_2, v_2 :: s_2, G_2)$, and thus $(\rho_1, s_1, G_1) \sqsubseteq (\rho_2, s_2, G_2)$.

Case: $b = \texttt{goto}\ a$

The abstract semantics rules do not change the input, thus $\overline{instr}_b(Q_1, \Gamma_1) = Q_1$ and $\overline{instr}_b(Q_2, \Gamma_2) = Q_2$. Since $Q_1 \sqsubseteq Q_2$, we can conclude straightforward with

$\overline{instr}_b(Q_1, \Gamma_1) \sqsubseteq \overline{instr}_b(Q_2, \Gamma_2)$.

Case: $b = \texttt{iconst\_}n$ **or** $b = \texttt{new}iC$ **o** $b = \texttt{aconst\_null}$

The instruction adds an abstract element on the stack (the same element for Q_1 and Q_2). Thus, the monotonicity is preserved.

Case: $b = \texttt{load x}$

From hypothesis, $(\rho_1, s_1, G_1) \sqsubseteq (\rho_2, s_2, G_2)$. Thus, $\rho_1(x) \subseteq \rho_2(x)$ and $(\rho_1, \rho_1(x) :: s_1, G_1) \sqsubseteq (\rho_2, \rho_2(x) :: s_2, G_2)$.

Case: $b = \texttt{store x}$

Let $Q_1 = (\rho_1, u_1 :: s_1, G_1)$, $Q_2 = (\rho_2, u_2 :: s_2, G_2)$, $\overline{instr}_b(Q_1, \Gamma_1) = (\rho_1[x \mapsto u'_1], s_1, G_1)$, where $u'_1 = u_1 \cup \{(t, \texttt{i}) \mid t \in \Gamma_1\}]$, and $\overline{instr}_b(Q_2, \Gamma_2) = (\rho_2[x \mapsto u'_2], s_2, G_2)$, where $u'_2 = u_2 \cup \{(t, \texttt{i}) \mid t \in \Gamma_2\}]$. To prove the monotonicity, we must show that $u'_1 \subseteq u'_2$. As $Q_1 \sqsubseteq Q_2$ and $\Gamma_1 \subseteq \Gamma_2$(from hypothesis), we also have $u_1 \subseteq u_2$ and $\{(t, \texttt{i}) \mid t \in \Gamma_1\} \subseteq \{(t, \texttt{i}) \mid t \in \Gamma_2\}$. Hence, $u'_1 \subseteq u'_2$ and $\overline{instr}_b(Q_1, \Gamma_1) \sqsubseteq \overline{instr}_b(Q_2, \Gamma_2)$.

Case: $b = \texttt{getfield}\ f_{C'}$

Let $Q_1 = (\rho_1, u_1 :: s_1, G_1)$, $Q_2 = (\rho_2, u_2 :: s_2, G_2)$, $\overline{instr}_b(Q_1, \Gamma_1) = (\rho_1, u'_1 :: s_1, G_1)$ and $\overline{instr}_b(Q_2, \Gamma_2) = (\rho_2, u'_2 :: s_2, G_2)$. To prove the monotonicity, we must show that $u'_1 \subseteq u'_2$, where $u'_1 = \{(e, \texttt{d}) \mid e \in adj_{G_1}(e', \langle f_{C'}, \texttt{d}\rangle) \wedge (e', \texttt{d}) \in u_1\} \cup \{(e, \texttt{i}) \mid (e, \texttt{i}) \in u_1\}$. From $u_1 \subseteq u_2$, obviously, $\{(e, \texttt{i}) \mid (e, \texttt{i}) \in u_1\} \subseteq$

$\{(e, \mathtt{i}) \mid (e, \mathtt{i}) \in u_2\}$. Let us denote by u_1'' the first set in u_1'. We now show that $u_1'' \subseteq u_2''$, where

$$u_1'' = \{(e, \mathbf{d}) \mid e \in adj_{G_1}(e_1', \langle f_{C'}, \mathbf{d}\rangle) \wedge (e_1', \mathbf{d}) \in u_1\}$$

$$u_2'' = \{(e, \mathbf{d}) \mid e \in adj_{G_2}(e_2', \langle f_{C'}, \mathbf{d}\rangle) \wedge (e_2', \mathbf{d}) \in u_2\}.$$

Since $G_1 \subseteq G_2$ and $u_1 \subseteq u_2$, all the adjacents labeled by $f_{C'}$ of e_1' (where $(e_1', \mathbf{d}) \in u_1$) are also adjacents of e_2' (where $(e_2', \mathbf{d}) \in u_2$). Thus $u_1'' \subseteq u_2''$ and $u_1' \subseteq u_2'$.

Case: b = **`putfield`** $f_{C'}$

Let $Q_1 = (\rho_1, v_1 :: u_1 :: s_1, (V_1, E_1))$, $Q_2 = (\rho_2, v_2 :: u_2 :: s_2, (V_2, E_2))$. Moreover, let $\overline{instr}_b(Q_1, \Gamma_1) = (\rho_1, s_1, (V_1, E_1'))$ and $\overline{instr}_b(Q_2, \Gamma_2) = (\rho_2, s_2, (V_2, E_2'))$. To prove the monotonicity, we must show that $E_1' \subseteq E_2'$. From `putfield` abstraction transformation rule in Figure 6, $E_1' = E_1 \cup E_1'' \cup E_1'''$, where

$$E_1'' = \{(e, e', \langle f_{C'}, t\rangle) | (e, \mathbf{d}) \in u, (e', t) \in v_1\},$$

$$E_1''' = \{(e, e', \langle f_{C'}, \mathtt{i}\rangle) | (e, \mathbf{d}) \in u_1, e' \in \Gamma \cup V_{\mathtt{i}}^{u_1}\},$$

$$V_{\mathtt{i}}^{u_1} = \{e \mid \langle e, \mathtt{i}\rangle \in u_1\}.$$

The definition of E_2' is similar. By hypothesis, $u_1 \subseteq u_2$ and $v_1 \subseteq v_2$, thus $E_1'' \subseteq E_2''$. Moreover, as $\Gamma_1 \subseteq \Gamma_2$, the inclusion $E_1''' \subseteq E_2'''$ also holds. Thus, $E_1' \subseteq E_2'$.

B Soundness of the intra-method analysis

B.1 Proof of Lemma 5.4 (Local soundness)

Lemme B.1 (Local Soundness) *Let α be an abstraction function. Let Q be an execution state and $\overline{Q}$ an abstract state such that $Q \lhd^\alpha \overline{Q}$. Then for any bytecode b, for any set of abstract values Γ, $instr_b(Q) \lhd^\alpha \overline{instr}_b(\overline{Q}, \Gamma)$.*

Proof.
By case analysis on each bytecode instruction. Except for the `new` bytecode, the other intructions do not add new nodes to memory graph and do not change the location of abjects. Thus, for a state Q and $Q' = instr_b Q$, $Q.G.V^{Obj} = Q'.G.V^{Obj}$ and hence $Q.G.\varsigma(u) = Q'.G.\varsigma(u)$, for all $u \in Q.G.V^{Obj}$. For simplicity, we denote by $\varsigma(u)$ the value of $Q.G.\varsigma(u)$ and $Q'.G.\varsigma(u)$.

Case: b = **`prim op`**

From the operational semantics of `prim` op (Figure 3), $Q = ((i, \rho, v_1 :: v_2 :: s) :: fr, G)$ and $instr_b(Q) = ((i+1, \rho, op(v_1, v_2) :: s) :: fr, G)$. From the abstract rule of `prim`, $\overline{Q} = (\overline{\rho}, \overline{v_1} :: \overline{v_2} :: \overline{s}, \overline{G})$ and $\overline{instr}_b(\overline{Q}, \Gamma) = (\overline{\rho}, \overline{v_1} \cup \overline{v_2} :: \overline{s}, \overline{G})$. From hypothesis, we have $G \lhd^\alpha \overline{G}$, $\rho \lhd^\alpha \overline{\rho}$, $s \lhd^\alpha \overline{s}$ and since v_1, v_2 are not objects we also have $(op(v_1, v_2) :: s) \lhd^\alpha (\overline{v_1} \cup \overline{v_2} :: \overline{s})$, and hence we can conclude with $instr_b(Q) \lhd^\alpha \overline{instr}_b(\overline{Q}, \Gamma)$.

Case: b = **`pop`** **or** b = **`ifeq`** a

From operational semantics, $Q = ((i, \rho, n :: s) :: fr, G)$ and $instr_b(Q) = ((j, \rho, s) :: fr, G)$, where j is $i+1$ or a. From the abstract rules, $\overline{Q} = (\overline{\rho}, \overline{v} :: \overline{s}, \overline{G})$ and

$\overline{instr}_b(\overline{Q}, \Gamma) = (\overline{\rho}, \overline{s}, \overline{G})$. As we have $Q \lhd^\alpha \overline{Q}$ from hypothesis, it is obvious that $instr_b(Q) \lhd^\alpha \overline{instr}_b(\overline{Q}, \Gamma)$.

Case: b = **goto** a

Let $Q = ((i,\rho,s) :: fr, G)$ and $instr_b(Q) = ((a,\rho,s) :: fr, G)$. The abstract semantics of goto instruction does not change the input state $\overline{Q}$, and thus $\overline{instr}_b(\overline{Q},\Gamma) = \overline{Q}$. The concrete semantics rule makes a jump to adress a and leaves unchanged the rest of the frame Q. Hence the α-abstraction relation is preserved, and $instr_b(Q) \lhd^{\alpha} \overline{instr}_b(\overline{Q},\Gamma)$.

Case: b = **iconst**_n

The bytecode adds a primitive value on top of the stack, thus since primitive values are not concerned by α-abstraction, we have $instr_b(Q) \lhd^{\alpha} \overline{instr}_b(\overline{Q},\Gamma)$.

Case: b = **aconst_null**

The bytecode adds pushes $null$ on the concrete stack and $n_{-1}^{\mathtt{null}}$ on the abstract stack. By definition, $\alpha(\varsigma^{-1}(null)) = n_{-1}^{\mathtt{null}}$.

Case: b = **new** C

The instruction pushes a new object on the stack. By construction, due to the allocation site model, the value pushed by the abstract rule (n_i^n) is the abstraction of the concrete object ($C_i^{\mathsf{fresh}(C_i,G)}$) pushed on the concrete stack. The concrete graph G is enlarged with the new object, while the dependency graph already containts the abstract node (we remind that we suppose that all abstract values are computed from the begining).

Case: b = **load x**

Let $Q = ((i,\rho,s) :: fr, G)$ and $\overline{Q} = (\overline{\rho},\overline{s},\overline{G})$. From the operational semantics rule of load x (Figure 3), we have $instr_b(Q) = ((i+1,\rho,\rho(x) :: s) :: fr, G)$. From the abstract transformation rule of load x (Figure 6), we have $\overline{instr}_b(\overline{Q},\Gamma) = (\overline{\rho},\overline{\rho}(x) :: \overline{s},\overline{G})$. By hypothesis, we have $Q \lhd^{\alpha} \overline{Q}$, thus by definition, we have $s \lhd^{\alpha} \overline{s}$, $\rho \lhd^{\alpha} \overline{\rho}$ and $G \lhd^{\alpha} \overline{G}$

The local variables array abstraction relation $\rho \lhd^{\alpha} \overline{\rho}$ implies that for all x such that $\rho(x) \in Obj$, for all $\overline{n} \in \alpha(\varsigma^{-1}(\rho(x)))$ we have $(\overline{n},\mathbf{d}) \in \overline{\rho}(x)$. Since $s \lhd^{\alpha} \overline{s}$, the definition of stack abstraction gives us $(\rho(x) :: s) \lhd^{\alpha} (\overline{\rho}(x) :: \overline{s})$. We can conclude with $instr_b(Q) \lhd^{\alpha} \overline{instr_b}(\overline{Q})$.

Case: b = **store** x

From the concrete semantics rule, if $Q = ((i,\rho,n :: s) :: fr, G)$, then $instr_b(Q) = ((i+1,\rho \oplus \{x \mapsto n\}, s) :: fr, G)$. From the abstract rules, if $\overline{Q} = (\overline{\rho},\overline{u} :: \overline{s},\overline{G})$, then $\overline{instr}_b(\overline{Q},\Gamma) = (\overline{\rho}[x \mapsto \overline{u} \cup \{(t,\mathtt{i}) \mid t \in \Gamma\}],\overline{s},\overline{G})$. The α-abstraction on stack and graphs is straightforward. From hypothesis and the α-abstraction definition on stack, we have that $\alpha(\varsigma^{-1}(n)) \in \overline{u}$, thus $\rho[x] \in \overline{\rho}[x]$, and, as the other elements of the local variable array do not change, $\rho \lhd^{\alpha} \overline{\rho}$. Moreover, by hypothesis $s \lhd^{\alpha} \overline{s}$ and $G \lhd^{\alpha} \overline{G}$, thus the local soundness holds.

Case: b = **putfield** $f_{C'}$

Let $Q = ((i,\rho,v :: u :: s) :: fr, G)$ and $\overline{Q} = (\overline{\rho},\overline{v} :: \overline{u} :: \overline{s},\overline{G})$. We consider the case when $v \notin Val$ (the case $v \in Val$ is straightforward, as the edges between references do not change and neither does the stack and local variables array).

From the operational semantics rule of putfield (Figure 3), we have $instr_b(Q) = ((i+1,\rho,s) :: fr, G')$. From the abstract transformation rule of putfield (Figure 6), we have $\overline{instr}_b(\overline{Q},\Gamma) = (\overline{\rho},\overline{s},\overline{G'})$. From the hypothesis, $s \lhd^{\alpha} \overline{s}$, $\rho \lhd^{\alpha} \overline{\rho}$.

We still need to prove that $G' \lhd^{\alpha} \overline{G}'$. Since $G \lhd^{\alpha} \overline{G}$ and because we only add and never delete edges to/from $\overline{G}$, all we need to prove is that the edge added by $instr_b$ respects the abstraction property *i.e.* the abstraction of the edge added by $instr_b$ in G must be in $\overline{G}$.

As, by definition, $G' = G[(\varsigma^{-1}(u), f_{C'}) \mapsto \varsigma^{-1}(v)]$, the only edge added to G' is $(\varsigma^{-1}(u), \varsigma^{-1}(v), f_{C'})$. From $Q \lhd^{\alpha} \overline{Q}$ and the stack abstraction definition, we have:

$$(\alpha(\varsigma^{-1}(v)), \mathbf{d}) \in \overline{v}$$
$$(\alpha(\varsigma^{-1}(u)), \mathbf{d}) \in \overline{u}$$

The abstract transformation rule of `putfield` adds to $\overline{G'}$ edges having the form $(\overline{n'}, \overline{n}, \langle f_{C'}, \mathbf{d} \rangle)$, $\forall \overline{n'} \in \overline{u}$ and $\forall \overline{n} \in \overline{v}$.

Thus, it adds also the edge $(\alpha(\varsigma^{-1}(u)), \alpha(\varsigma^{-1}(v)), \langle f_{C'}, \mathbf{d} \rangle)$. Thus $G' \lhd^{\alpha} \overline{G}'$.

Case: b = **`getfield`** $f_{C'}$

From the semantics rules of `getfield`, the local variables array and the memory do not change. Thus, we have to prove the α-abstraction for the stack. Let $Q = ((i, \rho, n :: s) :: fr, G)$ and $\overline{Q} = (\overline{\rho}, \overline{u} :: \overline{s}, \overline{G})$.

By concrete and abstract semantics rules, $instr_b(Q) = ((i+1, \rho, \varsigma(n') :: s) :: fr, G)$ and $\overline{instr}_b(\overline{Q}, \Gamma) = (\overline{\rho}, \overline{u'} :: \overline{s}, \overline{G})$. By hypothesis, we have $s \lhd^{\alpha} \overline{s}$, hence we still need to show that the α-abstraction is preserved for the top of the resulted stack (between $\varsigma(n')$ and $\overline{u}$).

We have two cases: the field $f_{C'}$ is of primitive type or is an object. For the first case, $\varsigma(n') \in \mathit{Val}$ and thus the α-abstraction is preserved, as it does not apply on primitive values. We consider the second case, in which $\varsigma(n') \in \mathit{Obj}$. From Definition 4.5 of stack abstraction, we must prove that $(\alpha(n'), \mathbf{d}) \in \overline{u}$, where $n' \in adj_G(\varsigma^{-1}(n), f_{C'})$.

By hypothesis, we also have $G \lhd^{\alpha} \overline{G}$, which means that $\alpha(\Pi_{Obj}(G)) \subseteq \overline{G}$. By definition, $\alpha(G).E = \{(\alpha(v), \alpha(v'), \langle e, \mathbf{d} \rangle) \mid v, v' \in G.V^{Obj} \wedge (v, v', e) \in G.E\}$. As $\alpha(G).E \subseteq \overline{G}.E$ and $(n, n', f_{C'}) \in G.E$, we have $(\alpha(n), \alpha(n'), \langle f_{C'}, \mathbf{d} \rangle) \in \overline{G}.E$, and from the value of $\overline{u'}$ from the abstract transformation rule of $getfield$, we can conclude with $(\alpha(n'), \mathbf{d}) \in \overline{u'}$.

□

B.2 Proof of Proposition 5.7

Details of the case analysis for the proof of the Proposition 5.7.

Case: b = **`prim`** **op**

The instruction pops two operands from the stack and pushes back the result of *op* on these operands. The operands are of primitive types. Thus the relation $\doteq$ holds for the resulted stack. The local variables array and the graph are not changed.

Case: b = **`pop`** **or** b = **`ifeq`** a

From the abstract semantics in Figure 6, $Q_1 = (\rho_1, v_1 :: s_1, G_1)$ and $instr_b(Q_1, \Gamma_1) = (\rho_1, s_1, G_1)$. As the only operation performed by these bytecodes is a pop from the stack, obviously $\overline{instr}_b(Q_1, \Gamma_1) \doteq \overline{instr}_b(Q_2, \Gamma_2)$.

Case: b = **goto** a

The transformation rules do not change the input, thus $\overline{instr}_b(Q_1, \Gamma_1) = Q_1$ and $\overline{instr}_b(Q_2, \Gamma_2) = Q_2$. Since $Q_1 \doteq Q_2$, we can conclude straightforward with

$\overline{instr}_b(Q_1, \Gamma_1) \doteq \overline{instr}_b(Q_2, \Gamma_2)$.

Case: b = **iconst**_iv **or** b = **aconst_null or** b = **new**_iC

The instruction adds an abstract element on the stack, either $\langle n_i^c, \mathbf{d}\rangle$ or $\langle n_{-1}^{\mathtt{null}}, \mathbf{d}\rangle$ or $\langle n_i^n, \mathbf{d}\rangle$, which is always the same for Q_1 and Q_2. Thus, the $\doteq$ relation is preserved.

Case: b = **load x**

From hypothesis, $(\rho_1, s_1, G_1) \doteq (\rho_2, s_2, G_2)$. Thus, $\rho_1(x) \doteq \rho_2(x)$ and thus $(\rho_1, \rho_1(x) :: s_1, G_1) \doteq (\rho_2, \rho_2(x) :: s_2, G_2)$.

Case: b = **store x**

Let $Q_1 = (\rho_1, u_1 :: s_1, G_1)$, $Q_2 = (\rho_2, u_2 :: s_2, G_2)$, $\overline{instr}_b(Q_1, \Gamma_1) = (\rho_1[x \mapsto u_1'], s_1, G_1)$, where $u_1' = u_1 \cup \{(t, \mathtt{i}) \mid t \in \Gamma_1\}]$, and $\overline{instr}_b(Q_2, \Gamma_2) = (\rho_2[x \mapsto u_2'], s_2, G_2)$, where $u_2' = u_2 \cup \{(t, \mathtt{i}) \mid t \in \Gamma_2\}]$. To prove the relation $\doteq$, we must show that $u_1' \doteq u_2'$, which holds because , from hypothesis, we have $u_1 \doteq u_2$ and the restriction of set $\{(t, \mathtt{i}) \mid t \in \Gamma_1\}$ to objects in $\emptyset$. Hence, $\overline{instr}_b(Q_1, \Gamma_1) \doteq \overline{instr}_b(Q_2, \Gamma_2)$.

Case: b = **getfield** $f_{C'}$

Let $Q_1 = (\rho_1, u_1 :: s_1, G_1)$, $Q_2 = (\rho_2, u_2 :: s_2, G_2)$, $\overline{instr}_b(Q_1, \Gamma_1) = (\rho_1, u_1' :: s_1, G_1)$ and $\overline{instr}_b(Q_2, \Gamma_2) = (\rho_2, u_2' :: s_2, G_2)$. To prove the relation $\doteq$, we must show that $u_1' \doteq u_2'$, where $u_1' = \{(e, \mathbf{d}) \mid e \in adj_{G_1}(e', \langle f_{C'}, \mathbf{d}\rangle) \wedge (e', \mathbf{d}) \in u_1\} \cup \{(e, \mathtt{i}) \mid (e, \mathtt{i}) \in u_1\}$. The elements in $\{(e, \mathtt{i}) \mid (e, \mathtt{i}) \in u_1\}$ refer to value nodes, thus they do no present interest. Let us denote by u_1'' the first set in u_1'. We now show that $u_1'' \doteq u_2''$, where

$$u_1'' = \{(e, \mathbf{d}) \mid e \in adj_{G_1}(e_1', \langle f_{C'}, \mathbf{d}\rangle) \wedge (e_1', \mathbf{d}) \in u_1\}$$

$$u_2'' = \{(e, \mathbf{d}) \mid e \in adj_{G_2}(e_2', \langle f_{C'}, \mathbf{d}\rangle) \wedge (e_2', \mathbf{d}) \in u_2\}.$$

Since $G_1 \doteq G_2$ and $u_1 \doteq u_2$, all the adjacents labeled by $f_{C'}$ of e_1' (where $(e_1', \mathbf{d}) \in u_1$) are also adjacents of e_2' (where $(e_2', \mathbf{d}) \in u_2$). Thus $u_1'' \doteq u_2''$ and $u_1' \doteq u_2'$.

Case: b = **putfield** $f_{C'}$

Let $Q_1 = (\rho_1, v_1 :: u_1 :: s_1, (V_1, E_1))$, $Q_2 = (\rho_2, v_2 :: u_2 :: s_2, (V_2, E_2))$,

$\overline{instr}_b(Q_1, \Gamma_1) = (\rho_1, s_1, (V_1, E_1'))$ and $\overline{instr}_b(Q_2, \Gamma_2) = (\rho_2, s_2, (V_2, E_2'))$. To prove the relation $\doteq$, we must show that $(V_1, E_1') \doteq (V_2, E_2')$, thus the `putfield` rule must add the same edges btween objects in E_1' and in E_2'.

From abstraction transformation rule in Figure 6, $E_1' = E_1 \cup E_1'' \cup E_1'''$, where $E_1'' = \{(e, e', \langle f_{C'}, t\rangle) | (e, \mathbf{d}) \in u, (e', t) \in v_1\}$ and $E_1''' = \{(e, e', \langle f_{C'}, \mathtt{i}\rangle) | (e, \mathbf{d}) \in u_1, e' \in \Gamma \cup V_{\mathtt{i}}^{u}\}$, $V_{\mathtt{i}}^{u_1} = \{e \mid \langle e, \mathtt{i}\rangle \in u\}$. The definition of E_2' is similar. Edges in E_1''' are between objects and values, thus are not considered by $\doteq$. We must refer only to E_1'' and E_2''. From hypothesis, $u_1 \doteq u_2$ and $v_1 \doteq v_2$, thus $E_1'' \doteq E_2''$. Thus, $(V_1, E_1') \doteq (V_2, E_2')$.

B.3 Proof of Proposition 5.15 (State variation correctness for single instruction block)

By case analysis on each instruction.

Case: b = **`prim op`**

The local variables array and the memory / dependency graphs are not affected by the execution of this bytecode, thus we still need to prove the state variation for the stack.

Let $Q_1.s = n_1 :: n_2 :: s$, $\overline{Q_1}.s = \overline{n_1} :: \overline{n_2} :: \overline{s}$, $Q'_1.s = n'_1 :: n'_1 :: s'$. From the operational semantics rule of `prim op` in Figure 3, we have $Q_2.s = op(n_1, n_2)$, $Q'_2 = op(n'_1, n'_2)$ and, from the abstract transformation rule, $\overline{Q_2}.s = (\overline{n_1} \cup \overline{n_2}) :: \overline{s}$. As n_1 and n_2 must be primitive values ($n_1, n_2 \in Val$), only the rule 1 of Definition 5.11 (stack variation) can be applied to them.

If there exists $\eta \in \beta$, $t \in \mathcal{F}$ such that $(\eta, t) \in \overline{n_1}$, then the value on the top of the stack is randomize and $Q'_1.s = Q_1.s[0 \mapsto x]$ with $x \in Val$. Thus, $Q'_2.s = Q_2.s[0 \mapsto op(x, n_2)]$, which corresponds to rule 1 of Definition 5.11, as (η, t) belongs to the top of the stack in $\overline{Q_2}.s$ ($(\eta, t) \in n_1$ and obviously $(\eta, t) \in (n_1 \cup n_2)$). Thus $Q'_2.s = \upsilon(Q'_1.s, \overline{Q_1}.s, \beta)$.

Case: b = **`pop`**

In both concrete and abstract semantics, the instructions pop a value from the stack. The rest of the stack, the local variables array and the memory/dependency graph remain unchanged. Thus, if Q'_1 is the state variation of Q_1 with respect to $\overline{Q_1}$ and β, and $Q'_2 = instr_b(Q'_1)$ and $\overline{Q_2} = \overline{instr_b}(\overline{Q_1}, \Gamma)$, then Q'_2 is a state variation of Q_2 with respect to $\overline{Q_2}$ and β.

Case: b = **`goto`** a

The proof is similar with the one for `pop` bytecode, as the instruction does not change local the variables and memory / dependency graph and moreover, does not change the stack.

Case: b = **`iconst_`**n **or** b = **`new`**iC **or** b = **`aconst_null`**

A constant value is pushed on the stack (for both states Q_1 and the state variation Q'_1). The rest of the frame remains unchanged, thus the proof is straightforward.

Case: b = **`load`** x

Let $\overline{Q_1} = (\overline{\rho}, \overline{s}, \overline{G})$, $\overline{Q_2} = (\overline{\rho}, \overline{\rho}(x) :: \overline{s}, \overline{G})$ (from the abstract tranformation rules in Figure 6), $Q_1 = ((i, \rho, s) :: fr, G)$ and $Q_2 = ((i+1, \rho, \rho(x) :: s) :: fr, G)$ (from the semantics rule in Figure 3).

Let $Q'_1 = ((i, \rho', s') :: fr, G')$ be the state variation , and $Q'_2 = ((i+1, \rho', \rho'(x) :: s') :: fr, G')$.

The bytecode only pushes an element on the stack, thus the graph and local variables array do not change. From hypothesis, we have $G' = \upsilon(G, \overline{G}, \beta)$, thus the dependency graph in Q'_2 is a state variation of the graph in Q_2 with respect to the graph in $\overline{Q_2}$ and β. The same reasonement applies to the local variables. We still need to prove the state variation for stacks. q We know that s' is a state variation of s with respect to $\overline{s}$ and β. The top of the stack in Q'_2 ($\rho'(x)$) respects the rules 1 and 2 of Definition 5.12 of local variables array variation, and thus is a state variation of the top of the stack in Q_2 ($\rho(x)$) with respect to the top of the stack in $\overline{Q}_2$ ($\overline{\rho}_2$) and β.

Case: b = `store` x

The instruction stores the top of the stack in the local variable x. The proof is similar to the `load` x bytecode: the top of the stack in the state variation Q'_1 respects the rules 1 and 2 of stack variation definition, thus the local variable x in Q'_2 respects the rules 1 and 2 of local variables array variation.

Case: b = `putfield` $f_{C'}$

Let $Q_1.s = v :: u :: s$, $\overline{Q_1}.s = \overline{v} :: \overline{u} :: \overline{s}$, $Q'_1.s = v'_1 :: u'_1 :: s'_1$.

Since `putfield` does not affect the local variables and only pops elements from the stack, we have: $Q_1.\rho = Q_2.\rho$ and $Q'_1.\rho = Q'_2.\rho$.

By hypotesis, $Q'_1.\rho = \upsilon(Q_1.\rho, \overline{Q_1}.\rho, \beta)$, thus replacing $Q_1.\rho$ by $Q_2.\rho$ and $Q'_1.\rho$ by $Q'_2.\rho$, we obtain the variation for local variables: $Q'_2.\rho = \upsilon(Q_2.\rho, \overline{Q_2}.\rho, \beta)$. We apply the same resonement for the stack: $s = Q_2.s$ and $s'_1 = Q'_2.s$. From the hypotesis, $Q'_1.s = \upsilon(Q_1.s, \overline{Q_1}.s, \beta)$, thus $s'_1 = \upsilon(\overline{Q_2}.s, s, \beta)$. By replacing s by $Q_2.s$ and s'_1 by $Q'_2.s$, we obtain the variation for the stack: $Q'_2.s = \upsilon(Q_2.s, \overline{Q_2}.s, \beta)$.

To completely prove the state variation correctness, we still need to prove the graph variation: $Q'_2.G = \upsilon(Q_2.G, \overline{Q_2}.G, \beta)$.

Depending on the type of the top of the stack v, there are two semantics rules for the `putfield` instruction (as depicted in Figure 3). We consider each case: v is a primitive value or an object.

Let us first consider that the field manipulated is of primitive type ($v \in Val$).

We will make a case study based on the values of v and u, depending on β. We distinguish the following possibilities:

- for $v \in Val$:
  - either $\exists \overline{\eta} \in \beta, t \in \mathcal{F}$ such that $(\overline{\eta}, t) \in \overline{v}$
  - or $\forall \overline{\eta} \in \beta, t \in \mathcal{F}(\overline{\eta}, t) \notin \overline{v}$.
- for $u \in Obj$:
  - either $\exists \overline{\eta'} \in \beta$ such that $(\overline{\eta'}, \mathbf{i}) \in \overline{u}$
  - or $\forall \overline{\eta'} \in \beta, (\overline{\eta'}, \mathbf{i}) \notin \overline{u}$.

Based on the combination of cases presented above, we distinguish the following cases for the abstract stack, thus the following possibilities to compute the state Q'_1.

Case I: $\forall \overline{\eta} \in \beta, t \in \mathcal{F}, (\overline{\eta}, t) \notin \overline{v}$ **and** $\forall \overline{\eta'} \in \beta, (\overline{\eta'}, \mathbf{i}) \notin \overline{u}$

In this case, the rules of the state variation do not apply on v and u, thus $v' = v$ and $u' = u$. Since `putfield` is executed on the same objects, the same edges are added and since $Q'_1.G = \upsilon(Q_1.G, \overline{Q_1}.G, \beta)$, the conclusion is obvious $Q'_2.G = \upsilon(Q_2.G, \overline{Q_2}.G, \beta)$.

Case II: $\exists \overline{\eta} \in \beta, t \in \mathcal{F}$ **such that** $(\overline{\eta}, t) \in \overline{v}$ **and** $\forall \overline{\eta'} \in \beta, (\overline{\eta'}, \mathbf{i}) \notin \overline{u}$

We have a new value for v' and u is not affected by the state variation ($u = u'$). According to the concrete semantics, $Q_2.G = Q_1.G[n' \mapsto v]$ and $Q'_2.G = Q'_1.G[n' \mapsto v']$, where $n' = adj_G(Q_1.G.\varsigma^{-1}(u), f_{C'})$. Applying the abstract semantics of `putfield`, an edge $(\alpha(Q_1.G.\varsigma^{-1}(u)), \overline{\eta}, \langle f_{C'}, t\rangle)$ is added to $Q_2.G$. Thus, the graphs $Q'_2.G$ differs from $Q_2.G$ according to rule 1 of the definition 5.10 of Graph variation. Thus, $Q'_2.G$ is a graph variation of $Q_2.G$ with respect to $\overline{Q_2}.G$ and β.

Case III: $\exists\overline{\eta'} \in \beta$ **such that** $(\overline{\eta'}, \mathtt{i}) \in \overline{u}$

According to the rule 2 of Definition 5.11 of stack variation, $u \neq u'$ and $u' = \varsigma(\alpha^{-1}(\overline{o})$, with $(\overline{o}, \mathbf{d}) \in \overline{u}$. The transformation rule of `putfield` creates the links $(\alpha(Q_1.G.\varsigma^{-1}(u)), \overline{\eta}, \langle f_{C'}, \mathtt{i}\rangle)$ and $(\overline{o}, \overline{\eta}, \langle f_{C'}, \mathtt{i}\rangle)$ in $\overline{Q_2}.G$. At the concrete level, the semantics will change the value of adjacent of $\alpha^{-1}(\overline{o})$ and not $Q_1.G.\varsigma^{-1}(u)$, which respects the rule 1 of graph variation definition.

We consider now the case when the field $f_{C'}$ is of type reference ($v \in Obj$).

The idea is the same as the previous case, when $v \in Val$: we will make a case study based on the values of v and u, depending on β. We distinguish the following possibilities:

- for $v \in Obj$:
  - either $\exists\overline{\eta} \in \beta$ such that $(\overline{\eta}, \mathtt{i}) \in \overline{v}$
  - or $\forall\overline{\eta} \in \beta, (\overline{\eta}, \mathtt{i}) \notin \overline{v}$.
- for $u \in Obj$:
  - either $\exists\overline{\eta'} \in \beta$ such that $(\overline{\eta'}, \mathtt{i}) \in \overline{u}$
  - or $\forall\overline{\eta'} \in \beta, (\overline{\eta'}, \mathtt{i}) \notin \overline{u}$.

Case I: $\forall\overline{\eta} \in \beta, (\overline{\eta}, \mathtt{i}) \notin \overline{v}$ **and** $\forall\overline{\eta'} \in \beta, (\overline{\eta'}, \mathtt{i}) \notin \overline{u}$

In this case, the rules of the state variation do not apply on v and u, thus $v' = v$ and $u' = u$. Since $putfield$ is executed on the same objects, the same edges are added and since $Q'_1.G = \upsilon(Q_1.G, \overline{Q_1}.G, \beta)$, the conclusion is obvious $Q'_2.G = \upsilon(Q_2.G, \overline{Q_2}.G, \beta)$.

Case II: $\exists\overline{\eta} \in \beta$ **such that** $(\overline{\eta}, \mathtt{i}) \in \overline{v}$ **and** $\forall\overline{\eta'} \in \beta, (\overline{\eta'}, \mathtt{i}) \notin \overline{u}$

According to the rule 2 of Definition 5.11 of stack variation, $v \neq v'$ and $v' = \varsigma(\alpha^{-1}(\overline{o}))$, with $(\overline{o}, \mathbf{d}) \in \overline{v}$. The transformation rule of `putfield` creates the links $(\alpha(Q_1.G.\varsigma^{-1}(u)), \overline{\eta}, \langle f_{C'}, \mathbf{d}\rangle)$ and $(\alpha(Q_1.G.\varsigma^{-1}(u)), \overline{o}, \langle f_{C'}, \mathbf{d}\rangle)$ in $\overline{Q_2}.G$. At the concrete level, the semantics will add the edge $(Q_1.G.\varsigma^{-1}(u), \alpha^{-1}(\overline{o}), f_{C'})$ in $Q'_2.G$ and not $(Q_1.G.\varsigma^{-1}(u), v, f_{C'})$, as in $Q_2.G$. The α-abstraction is still preserved for $Q'_2.G$ and $\overline{Q_2}.G$, as the $(\alpha(u), \overline{o}, \langle f_{C'}, \mathbf{d}\rangle) \in \overline{Q_2}.G.E$. Thus $Q'_2.G \lhd^\alpha \overline{Q_2}.G$, and, according to rule 2 of Definition 5.10, we obtain the graphe variation: $Q'_2.G = \upsilon(Q_2.G, \overline{Q_2}.G, \beta)$.

Case III: $\exists\overline{\eta} \in \beta$ **such that** $(\overline{\eta}, \mathtt{i}) \in \overline{v}$ **and** $\exists\overline{\eta'} \in \beta$ **such that** $(\overline{\eta'}, \mathtt{i}) \in \overline{u}$

According to the rule 2 of Definition 5.11 of stack variation, $u \neq u'$ and $u' = \varsigma(\alpha^{-1}(\overline{o}))$, with $(\overline{o}, \mathbf{d}) \in \overline{u}$. In the same way, $v \neq v'$ and $v' = \varsigma(\alpha^{-1}(\overline{o'}))$, with $(\overline{o'}, \mathbf{d}) \in \overline{v}$.

The transformation rule of `putfield` creates the links $(\alpha(u), \overline{\eta}, \langle f_{C'}, \mathbf{d}\rangle)$, $(\overline{o}, \overline{\eta}, \langle f_{C'}, \mathbf{d}\rangle)$ and $(\alpha(u), \overline{o'}, \langle f_{C'}, \mathbf{d}\rangle)$ and $(\overline{o}, \overline{o'}, \langle f_{C'}, \mathbf{d}\rangle)$ in $\overline{Q_2}.G$. At the concrete level, the semantics will add the edge $(\alpha^{-1}(\overline{o}), \alpha^{-1}(\overline{o'}), f_{C'})$ in $Q'_2.G$ and not $(u, v, f_{C'})$, as in $Q_2.G$. The α-abstraction is still preserved for $Q'_2.G$ and $\overline{Q_2}.G$, as the $(\overline{o}, \overline{o'}, \langle f_{C'}, \mathbf{d}\rangle) \in \overline{Q_2}.G.E$. Thus $Q'_2.G \lhd^\alpha \overline{Q_2}.G$, and, according to rule 2 of Definition 5.10, $Q'_2.G = \upsilon(Q_2.G, \overline{Q_2}.G, \beta)$.

Case IV: $\forall\overline{\eta} \in \beta, (\overline{\eta}, \mathtt{i}) \notin \overline{v}$ **and** $\exists\overline{\eta'} \in \beta$ **such that** $(\overline{\eta'}, \mathtt{i}) \in \overline{u}$

This case is a simplification of the previous one, thus the proof is similar.

Case: b = **getfield** $f_{C'}$

The intruction pops an object and pushes the field $f_{C'}$ on the stack. Thus, to prove the state variation correctness, we must refer only to the new object/value pushed on the stack (the field $f_{C'}$) and prove that it can change only accordingly to rules 1 or 2 of Defition 5.11.

Let $Q_1.s = u :: s$, $\overline{Q_1}.s = \overline{u} :: \overline{s}$, and $Q'_1.s = u' :: s'$ be the state variation, with $u \in Obj$. Moreover, let $Q_2.s = v :: s$, $\overline{Q_2}.s = \overline{v} :: \overline{s}$, and $Q'_1.s = v' :: s'$.

We can have two cases: 1) for all $\eta \in \beta$, $(\eta, \mathtt{i}) \notin \overline{u}$ or 2) there exists $\eta \in \beta$ such that $(\eta, \mathtt{i}) \in \overline{u}$.

1) Let us consider the first case: rule 2 does not apply to the top of the stack in Q_1, thus $u' = u$. The `getfield` bytecode pushes on the stack the adjacent of u in $Q_1.G$. Since $Q'_1.G$ is a state variation of $Q_1.G$ with respect to $\overline{Q_1}.G$ and β, $\overline{Q_1}.G$ is an α-abstraction of $Q'_1.G$ ($Q'_1.G \lhd^\alpha \overline{Q_1}.G$). We have again two cases (we denote by u'' the node containing the object u; $u'' = Q_1.G.\varsigma^{-1}(u)$):

- either $\beta \cap adj_{\overline{Q_1}.G}(\alpha(u''), f_{C'}) \neq \emptyset$
- or $\beta \cap adj_{\overline{Q_1}.G}(\alpha(u''), f_{C'}) = \emptyset$

In the second case, the adjacent labeled $f_{C'}$ of u does not change (conform to rule 2 of graph variation definition), thus the instruction pushes the same object/value on the stack for both $Q'_1.G$ and $Q_1.G$ ($v = v'$).

We consider now the first case ($\alpha(u'')$ has an adjacent in β in $\overline{Q_1}.G$). Than there exists $\overline{\eta} \in \beta$ and $t \in \mathcal{F}$ and an edge $(\alpha(u''), \overline{\eta}, \langle f_{C'}, t\rangle)$ in $\overline{Q_1}.G$, We have two possibilities: $f_{C'}$ is of primitive type or is of type reference.

If the field $f_{C'}$ is of primitive type, the value of $adj_{Q_1.G}(u'', f_{C'})$ might change according with the rule 1 of graph variation definition (Definition 5.10). Thus, the value pushed on the stack in $Q'_1.G$ will differ from the one in $Q_1.G$, but it will respect the rule 1 of Definition 5.13, since the abstract value $(\overline{\eta}, t)$ will be pushed on the abstract stack in Q_2 ($(\overline{\eta}, t) \in \overline{v'}$ since $(\alpha(u''), \overline{\eta}, \langle f_{C'}, t\rangle) \in \overline{Q_1}.G.E \subseteq \overline{Q_2}.G.E$).

If the field $f_{C'}$ is of type reference, the adjacent of u'' might change according to the rule 2 of graph variation definition. Since $Q'_1.G \lhd^\alpha \overline{Q_1}.G$ and from the points-to correctness (Proposition 5.5), we have $instr_b(Q'_1) \lhd^\alpha \overline{instr_b}(\overline{Q_1}, \Gamma)$ or $Q'_2 \lhd^\alpha \overline{Q_2}$. Thus $Q'_2.s \lhd^\alpha \overline{Q_2}.s$, $(\overline{\eta}, \mathtt{i}) \in \overline{v'}$ and $\alpha(\varsigma^{-1}(v')) \in \overline{v'}$. Hence the stack respects the rule 2 of stack variation definition.

2) Let us consider now the case when there exists $\eta \in \beta$ such that $(\eta, \mathtt{i}) \in \overline{u}$. Thus the rule 2 of Defition 5.11 applies to u, and $u' = \alpha^{-1}(\overline{o})$, with $(\overline{o}, \mathtt{i}) \in \overline{u}$. From the abstract semantics of `getfield`, $(\overline{\eta}, \mathtt{i})$ is kept on the stack ($(\overline{\eta}, \mathtt{i}) \in \overline{v}$). Thus, the value of v' will change, but it will respect the rules 1 or 2 of stack variation definition (the proof is similar with the previous case, depending on the type of $f_{C'}$, primitive or reference).

B.4 Proof of Proposition 5.15 (State variation correctness for blocks)

- (sequence) : let us name respectively B_1, B_2 the non-terminals in the right-hand side from top to bottom. Let G_1, G_2 two graphs without non-terminals such that in the derivation producing G, $\mathsf{B}_1 \to^{k_1} G_1$ and $\mathsf{B}_2 \to^{k_2} G_2$. Let B_1, B_2 be the two blocks corresponding to G_1, G_2. Let e, e_1, e_2 be the exit points of B, B_1, B_2 respectively.

Remind that $\overline{instr}_B$ and Γ_e is computed from $\mathcal{E}_B$ and $\overline{instr}_{B_1}$, Γ_{e_1} and $\overline{instr}_{B_2}$, Γ_{e_2} are computed from $\mathcal{E}_{B_1}, \mathcal{E}_{B_2}$ respectively. From these definitions and the fact that due to the definition of contexts, Γ_{e_1} is computed independently of $\mathcal{E}_{B_2}$, one can see that $\overline{instr}_B(\overline{Q}, \Gamma)$ is $\overline{instr}_{B_2}(\overline{instr}_{B_1}(\overline{Q_1}, \Gamma_1), \Gamma_{e_1})$ where Γ_{e_1} is computed for $\mathcal{E}_{B_1}$ with $(\overline{Q_1}, \Gamma_1)$ as initial state.

- As k_1 is strictly smaller than k, by induction hypothesis, we have for G_1 (and thus, B_1) that for any concrete state Q_1, any abstract state $\overline{Q_1}$ such that $Q_1 \lhd^{\alpha} \overline{Q_1}$, $Q_2 = instr_{B_1}(Q_1)$, and $\overline{Q_2} = \overline{instr}_{B_1}(\overline{Q_1}, \Gamma_1)$.
  If $Q'_1 = \upsilon(Q_1, \overline{Q_1}, \beta)$ and $Q'_2 = instr_{B_2}(Q'_1)$ then $Q'_2 = \upsilon(Q_2, \overline{Q_2}, \beta)$.
- As k_2 is strictly smaller than k, by induction hypothesis, we have for G_2 (and thus, B_2) that for any concrete state Q_2, any abstract state $\overline{Q_2}$ such that $Q_2 \lhd^{\alpha} \overline{Q_2}$, $Q_3 = instr_{B_2}(Q_2)$, and $\overline{Q_3} = \overline{instr}_{B_2}(\overline{Q_1}, \Gamma_1)$.
  If $Q'_1 = \upsilon(Q_1, \overline{Q_1}, \beta)$ and $Q'_2 = instr_{B_2}(Q'_1)$ then $Q'_2 = \upsilon(Q_2, \overline{Q_2}, \beta)$.

Hence, by combining these two results, one has that for any concrete state Q_1, any abstract state $\overline{Q_1}$ such that $Q_1 \lhd^{\alpha} \overline{Q_1}$, any Γ_1, denoting Q_3 the abstract state $\overline{instr}_{B_2}(\overline{instr}_{B_1}(\overline{Q_1}, \Gamma_1), \Gamma_{e_1})$,

if $Q'_1 = \upsilon(Q_1, \overline{Q_1}, \beta)$ and $\overline{instr}_{B_2}(\overline{instr}_{B_1}(\overline{Q'_1}, \Gamma_1), \Gamma_{e_1})$, then

$Q'_3 = \upsilon(Q_3, \overline{Q_3}, \beta)$. This allows us to conclude this case due to the definition of $\overline{instr}_B$.

- (if then else) : let us name respectively B_1, B_2, B_3 the non-terminals in the right-hand side, B_1 being the "then" block, B_2 the "else" and B_3 the junction block. Let G_1, G_2, G_3 the graphs without non-terminals such that in the derivation producing G, $\mathsf{B}_1 \rightarrow^{k_1} G_1$, $\mathsf{B}_2 \rightarrow^{k_2} G_2$ and $\mathsf{B}_3 \rightarrow^{k_3} G_3$. Let B_1, B_2, B_3 be the blocks corresponding to G_1, G_2, G_3 respectively. Let e, e_1, e_2, e_3 be the exit points of B, B_1, B_2, B_3 respectively.

  For any concrete state Q_1, any abstract state $\overline{Q_1}$ such that $Q_1 \lhd^{\alpha} \overline{Q_1}$, we consider $Q'_1 = \upsilon(Q_1, \overline{Q_1}, \beta)$. We assume that the execution of B on Q_1 leads to execute the block B_1, that is the top of the stack s_1 in Q_1 is equal to 0 (the other case is dual can be be treated similarly). Let $Q_2 = instr_{\texttt{ifeq a}}(Q_1)$, $Q_3 = instr_{B_1}(Q_2)$ and $Q_4 = instr_{B_3}(Q_3)$. Now depending on the top of the stack $\overline{s}_1$ in the abstract state $\overline{Q_1}$:

  - there is no $\overline{\eta}$ from β such that $\langle \overline{\eta}, \texttt{i} \rangle \in \overline{s}_1[0]$: by definition of state variation, the top of the stack s'_1 in Q'_1 is equal to 0. Let us consider $Q'_2 = instr_{\texttt{ifeq a}}(Q'_1)$; as $s'_1[0] = 0$, it is easy to see due to the semantics of the bytecode `ifeq a`, $Q'_2 = \upsilon(Q_2, \overline{Q_2}, \beta)$ where $\overline{Q_2} = \overline{instr}_{\texttt{ifeq a}}(\overline{Q_1}, \Gamma_1)$. Now, by induction hypothesis for the block B_1, we have $Q'_3 = \upsilon(Q_3, \overline{Q_3}, \beta)$ where $Q'_3 = instr_{B_1}(Q'_2)$ and $\overline{Q_3} = \overline{instr}_{B_1}(\overline{Q_2}, \Gamma'_1)$. Let us denote $\overline{Q_4} = \overline{instr}_{B_2}(\overline{Q_2})$. By weakening the state variation, one has $Q'_3 = \upsilon(Q_3, \overline{Q_3} \sqcup \overline{Q_4}, \beta)$. By induction hypothesis for the block B_3, denoting $Q'_4 = instr_{B_3}(Q'_3)$, we have
  $$Q'_4 = \upsilon(Q_4, \overline{instr}_{B_3}(\overline{Q_3} \sqcup \overline{Q_4}, \Gamma_{e_1}), \beta)$$
  We conclude simply by using the definition of $\overline{instr}_B$.

  - there exists some $\overline{\eta}$ from β such that $\langle\overline{\eta}, \mathtt{i}\rangle \in \overline{s}_1[0]$: assume $Q_1 = ((i, \rho, v :: s) :: fr, G)$, the state variation Q'_1 is of the form $((i, \rho', v' :: s') :: fr, G')$. Let us consider $Q_2 = instr_{\mathtt{ifeq\ a}}(Q_1) = ((k, \rho, s) :: fr, G)$ and $Q'_2 = instr_{\mathtt{ifeq\ a}}(Q'_1) = ((k', \rho', s') :: fr, G')$. If $k = k'$ then this case is similar to the previous case. So, we assume that $k \neq k'$;

    Let $Q_3 = instr_{B_1}(Q_2)$ and $Q'_3 = instr_{B_2}(Q'_2)$; we are going to show first that $Q_3 = \upsilon(Q'_3, \overline{instr}_{B_1}(\overline{Q_2}, \Gamma_{e_1}) \sqcup \overline{instr}_{B_2}(\overline{Q_2}, \Gamma_{e_1}), \beta)$.

    Let $\overline{u}$ be the top of the stack in $\overline{Q_1}$. Then, for all i in both B_1 and B_2, we have that all e such that $\langle e, \mathtt{i}\rangle \in u$ belong to Γ_i. Intuitively, due to the abstract semantics, this implies that all the modified position (memory, local variables, stack) in concrete states can be identified in abstract states, and are concern by state variation. Thus, they can vary to show that $Q'_3 = \upsilon(Q_3, \overline{Q_3} \sqcup \overline{Q'_3}, \beta)$. Now, we can use the induction hypothesis for the block B_3 to conclude using the definition of $instr_B$ and of $\overline{instr}_B$.

- (if then) - (if else) : these cases are treated similarly to the (if then else) case.
- (while) : let us name respectively B_1, B_2, the non-terminals in the right-hand side, B_1 being the block within the loop and B_2 the end block of the loop. Let $\mathsf{G}_1, \mathsf{G}_2$ the graphs without non-terminals such that in the derivation producing G, $\mathsf{B}_1 \rightarrow^{k_1} \mathsf{G}_1$ and $\mathsf{B}_2 \rightarrow^{k_2} \mathsf{G}_2$ Let B_1, B_2 be the blocks corresponding to $\mathsf{G}_1, \mathsf{G}_2$ respectively. Let B be the block composed by the $ifeq$ node and B_1.

  For a concrete state $Q_1 = ((i_1, \rho_1, s_1) :: fr, G_1)$, let us denote $Q_2 = instr_B(Q_1) = ((i_2, \rho_2, s_2) :: fr, G_2)$ for some Γ_1, and let us consider an abstract state $\overline{Q_1}$ such that $Q_1 \lhd^{\alpha} \overline{Q_1}$, we consider $Q'_1 = \upsilon(Q_1, \overline{Q_1}, \beta)$ and $Q'_2 = instr_B(Q'_1)$ with $Q'_1 = ((i_1, \rho'_1, s'_1) :: fr, G'_1)$ and $Q'_2 = ((i_2, \rho'_2, s'_2) :: fr, G'_2)$. Let us denote $\overline{Q_2}$ such that $\overline{Q_2} = \overline{instr}_B(\overline{Q_1}, \Gamma_1)$ for some Γ_1 and $Q_2 \lhd^{\alpha} \overline{Q_2}$.

  If the state variation does not impact the top of the stack in $\overline{Q}_1$ then the control flow in the concrete execution is not modified and we are in the same situation than for the (if then else).

  If the top of the stack is impacted by the variation. Due to the abstract semantics that adds elements from Γ_1 to any manipulation of the state in B_1, any node or edge of G_2 that differs from G_1 is such that is mapping in $\overline{G_2}$ is linked by implicit flow to elements of Γ_1. The same goes for G'_1 and G'_2. Thus, everything in the ntersection of G_2 and G'_2 comes from G_1 and the rest is linked to elements of β either because it has been modified in the loop or because it has been modifed by the initial variation. In both cases it can vary under β and then $G'_2 = \upsilon(G'_2, \overline{G_2}, \beta)$. We can treat similarly local variables and the stack: the specification of Java bytecode tells us that s_2 and s'_2 have the same size, even if the block B may have been executed different numbers of times to lead to Q_2 and Q'_2.
- (repeat) - (while not) - (repeat not) : these cases are treated similarly to the (while) case.

C Soundness of inter-procedural analysis

We have to add the cases of the two new bytecodes (`invoke` and `areturn`), to the proofs that have been done by case analysis on each bytecode.

C.1 Addition to proof of Lemma 4.2

We must show that the abstract semantics rules are monotone. We consider two pairs (Q_1, Γ_1) and (Q_2, Γ_2) from the property space $\mathcal{S}$ such that $(Q_1, \Gamma_1) \sqsubseteq (Q_2, \Gamma_2)$, and let $Q'_1 = \overline{instr}_b(Q_1, \Gamma_1)$ and $Q'_2 = \overline{instr}_b(Q_2, \Gamma_2)$. We prove that $Q'_1 \sqsubseteq Q'_2$.

By case analysis on each instruction b, based on the transformation rules in Figure 8.

Case: b = **`invoke m`**

Let $Q_1 = (\rho, u_{n_m} :: \cdots :: u_0 :: s, G)$ and $Q_2 = (\rho', u'_{n_m} :: \cdots :: u'_0 :: s', G')$. From the semantics of `invoke`,

$$\overline{instr}_b(Q_1, \Gamma_1) = (\{0 \mapsto u_0, \ldots, n_m \mapsto u_{n_m}\}, \epsilon, G)$$

$$\overline{instr}_b(Q_2, \Gamma_2) = (\{0 \mapsto u'_0, \ldots, n_m \mapsto u_{n'_m}\}, \epsilon, G').$$

As $Q_1 \sqsubseteq Q_2$ and $\Gamma_1 \subseteq \Gamma_2$, we have $u_0 \subseteq u'_0 \ldots u_{n_m} \subseteq u'_{n_m}$, $s \sqsubseteq s'$ and $G \subseteq G'$, thus $\overline{instr}_b(Q_1, \Gamma_1) \sqsubseteq \overline{instr}_b(Q_2, \Gamma_2)$.

Case: b = **`areturn`**

Let $Q_1 = (\rho, u :: s, G)$ and $Q_2 = (\rho', u' :: s', G')$. From the semantics of `areturn`,

$$\overline{instr}_b(Q_1, \Gamma_1) = (\zeta, u, G)$$

$$\overline{instr}_b(Q_2, \Gamma_2) = (\zeta, u', G').$$

As $Q_1 \sqsubseteq Q_2$ and $\Gamma_1 \subseteq \Gamma_2$, we have $u \subseteq u'$ and $G \subseteq G'$, thus $\overline{instr}_b(Q_1, \Gamma_1) \sqsubseteq \overline{instr}_b(Q_2, \Gamma_2)$.

C.2 Addition to proof of Lemma 5.4

We show here the local soundness of the extended model, that is the abstract semantics rules for `invoke` and `areturn` (defined in Figure 8) respect the local soundness, as defined in Theorem 5.4, in respect with the state abstraction extension (Definition 6.1).

Case: b = **`invoke m`**

From the operational semantics of `invoke m` (Figure 3), we have

$$Q = ((i, \rho, p_{n_m} :: \cdots :: p_1 :: p_0 :: s) :: fr, G)$$

$$instr_b(Q) = ((0, \{0 \mapsto p'_0, 1 \mapsto p_1 \ldots n_m \mapsto p_{n_m}\}, \epsilon) :: (i, \rho, s) :: fr, G).$$

From the abstract rule of `invoke m` (Figure 8),

$$(\overline{\rho}, \overline{u_{n_m}} :: \cdots :: \overline{u_0} : \overline{s}, \overline{G})$$

$$\overline{instr}_b(\overline{Q}, \Gamma) = (\{0 \mapsto \overline{u_0}, \ldots, n_m \mapsto \overline{u_{n_m}}\}, \epsilon, \overline{G}).$$

From hypothesis, we have $G \lhd^\alpha \overline{G}$, $p_0 \lhd^\alpha \overline{u_0} \ldots p_{n_m} \lhd^\alpha u_{n_m}$. Moreover, the stacks are empty and $\lhd^\alpha$. Thus, $instr_b(Q) \lhd^\alpha \overline{instr}_b(\overline{Q}, \Gamma)$ according to definition 6.1 .

Case: b = **areturn** From the operational semantics of areturn in Figure 3, we have

$$Q = ((i, \rho, v :: s) :: (i', \rho', s') :: fr, G)$$
$$instr_b Q = ((i'+1, \rho', v :: s') :: fr, G).$$

From the abstract rule of areturn:

$$\overline{Q} = (\overline{\rho}, \overline{u} :: \overline{s}, \overline{G})$$

$$\overline{instr}_b(\overline{Q} = (\zeta, \overline{u}, \overline{G}).$$

From hypothesis, $Q \lhd^\alpha \overline{Q}$, hence $G \lhd^\alpha \overline{G}$, $\rho \lhd^\alpha myabs\rho$, $v \lhd^\alpha myabsu$ and $s \lhd^\alpha \overline{s}$. Since $\overline{instr}_b(\overline{Q}).\rho = \zeta$, $instr_b(Q) \lhd^\alpha \overline{instr}_b(\overline{Q}, \Gamma)$ according to definition 6.1 if $G \lhd^\alpha \overline{G}$ and $v \lhd^\alpha \overline{u}$ which holds by hypothesis.

C.3 Addition to proof of Proposition 5.7

We prove the second point of the Proposition for both bytecodes, invoke m and areturn: for any Γ_1, Γ_2, if $Q_1 \doteq Q_2$ then $\overline{instr}_b(Q_1, \Gamma_1) \doteq \overline{instr}_b(Q_2, \Gamma_2)$.

Case: b = **invoke m**

Let $Q_1 = (\rho, u_{n_m} :: \cdots :: u_0 :: s, G)$ and $Q_2 = (\rho', u'_{n_m} :: \cdots :: u'_0 :: s', G')$. From the semantics of invoke in Figure 8,

$$\overline{instr}_b(Q_1, \Gamma_1) = (\{0 \mapsto u_0, \dots, n_m \mapsto u_{n_m}\}, \epsilon, G)$$

$$\overline{instr}_b(Q_2, \Gamma_2) = (\{0 \mapsto u'_0, \dots, n_m \mapsto u_{n'_m}\}, \epsilon, G').$$

As $Q_1 \doteq Q_2$, we have $u_0 \doteq u'_0 \dots u_{n_m} \doteq u'_{n_m}$ and $G \doteq G'$, thus $\overline{instr}_b(Q_1, \Gamma_1) \doteq \overline{instr}_b(Q_2, \Gamma_2)$.

Case: b = **areturn**

Let $Q_1 = (\rho, u :: s, G)$ and $Q_2 = (\rho', u' :: s', G')$. From the semantics of areturn in Figure 8,

$$\overline{instr}_b(Q_1, \Gamma_1) = (\zeta, u, G)$$
$$\overline{instr}_b(Q_2, \Gamma_2) = (\zeta, u', G').$$

As $Q_1 \doteq Q_2$, we have $u \doteq u'$ and $G \doteq G'$, thus $\overline{instr}_b(Q_1, \Gamma_1) \doteq \overline{instr}_b(Q_2, \Gamma_2)$.

C.4 Addition to proof of Proposition 5.15

Case: b = **invoke m**

From the operational semantics of invoke m (Figure 3), we have

$$Q = ((i, \rho, p_{n_m} :: \cdots :: p_1 :: p_0 :: s) :: fr, G)$$

$$instr_b(Q) = ((0, \{0 \mapsto p_0, 1 \mapsto p_1 \dots n_m \mapsto p_{n_m}\}, \epsilon) :: (i, \rho, s) :: fr, G).$$

From the abstract rule of invoke m (Figure 8),

$$\overline{Q} = (\overline{\rho}, \overline{u_{n_m}} :: \cdots :: \overline{u_0} : \overline{s}, \overline{G})$$

$$\overline{instr}_b(\overline{Q},\Gamma) = (\{0 \mapsto \overline{u_0},\ldots,n_m \mapsto \overline{u_{n_m}}\},\epsilon,\overline{G}).$$

Let Q' be the state variation and

$$Q' = \upsilon(Q,\overline{Q},\beta) = ((i,\rho',p'_{n_m} :: \cdots :: p'_1 :: p'_0 :: s') :: fr, G')$$
$$instr_b(Q') = ((0,\{0 \mapsto p'_0, 1 \mapsto p'_1 \ldots n_m \mapsto p'_{n_m}\},\epsilon) :: (i,\rho',s') :: fr, G').$$

From hypothesis, we have $Q'.G = \upsilon(Q.G,\overline{Q.G},\beta)$. The stack are empty, thus the state variation relation holds. We need to prove the state variation for the local variables array. In Q', p'_0 varies according to rules 1 and 2 of Definition 5.11 in function of β and $\overline{u_0}$ (the corresponding value on the abstract stack). In $instr_b(Q')$, the value of local variable 0 (which is p'_0) is a variation of the local variable 0 in the abstract state $\overline{instr}_b(\overline{Q},\Gamma)$ (which is $\overline{u_0}$) and of β, thus it respects the rules 1 and 2 of Definition 5.12 which define the definition of local variables. We apply the same reasoment on the rest of the local variables, and we obtain the state variation for local variables array. Thus $instr_b(Q') = \upsilon(instr_b(Q),\overline{instr}_b(\overline{Q},\Gamma),\beta)$.

Case: b = **`areturn`**

From the operational semantics of `areturn` (Figure 3), we have

$$Q = ((i,\rho,v :: s) :: (i_0,\rho_0,s_0) :: fr, G)$$
$$instr_b(Q) = ((i_0+1,\rho_0,v :: s_0) :: fr, G).$$

From the abstract rule of `areturn`:

$$\overline{Q} = (\overline{\rho},\overline{u} :: \overline{s},\overline{G})$$
$$\overline{instr}_b(\overline{Q}) = (\zeta,\overline{u},\overline{G}).$$

Let Q' be the state variation and

$$Q' = \upsilon(Q,\overline{Q},\beta) = ((i,\rho',v' :: s') :: (i_0,\rho_0,s_0) :: fr, G')$$
$$instr_b(Q') = ((i_0+1,\rho_0,v' :: s_0) :: fr, G').$$

Since $\overline{instr}_b(\overline{Q}).\rho = \zeta$, $instr_b(Q) = \upsilon(instr_b(Q'),\overline{instr}_b(\overline{Q}),\beta)$ if $v' = \upsilon(v,\overline{u},\beta)$ and $G = \upsilon(G',\overline{G},\beta)$. By hypothesis, $Q' = \upsilon(Q,\overline{Q},\beta)$ and thus the two relations required hold.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803