Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Tesi di laurea magistrale

Field-Sensitive Unreachability and Non-Cyclicity Analysis

Candidato
Enrico Scapin

Relatore
Prof. Dr. Nicola Fausto Spoto

Anno Accademico 2011-2012

To Nobody[1]
...my favorite classical Hero

[1] a.k.a. πολύτροπος i.e., able to adapt to different situations.

Acknowledgements

Sebbene dalla dedica si possa pensare il contrario, debbo ringraziare alcune persone. In primo luogo vorrei ringraziare il Prof. Nicola Fausto Spoto, per avermi pazientemente seguito durante questo lavoro di tesi.

Vorrei poi ringraziare i compagni con cui ho condiviso il laboratorio SPY durante quest'ultimo anno, Flavio A., Giulio B., Federico D. M., Fabio M., Oscar M., Riccardo R. e Fabio S., per l'atmosfera (fin troppo ☺) gioviale che si è instaurata tra quelle mura. In particolare, vorrei ringraziare Federico D. M. e Fabio S. per le numerose discussioni e congetture sugli argomenti studiati, permettendomi così di superare gli esami più facilmente, e Fabio M. per aver corretto il mio pessimo inglese nella tesi.

Un grazie ai miei compagni di scarpinate, Marco B., Giulio B., Marco C. e Matteo R., per le salutistiche giornate in montagna che mi hanno permesso di staccare un po' dall'informatica in quest'ultimo anno di studi. Ovviamente ringrazio anche i "butei" della compagnia, Francesca B., Manuel B., Giulia C., Lucia C., Diego C., Andrea C., Enrico C., Francesca G., Simone P. e Luca Z., per i sabati sera passati assieme a scorrazzare in tutta la provincia.

Un caloroso ringaziamento va anche alla mia famiglia, Stella Claudio e Nicolò, per il sostegno fornitomi in questi anni di università e per avermi dato la possibilità di esser stato uno studente "full-time".

Infine, un grazie anche a Sergey Brin, Larry Page e Jimmy Wales perchè, ammettiamolo, senza Google e Wikipedia sarebbe stato tutto più difficile!

Verona, Enrico Scapin
Ottobre 2012

[...] the purpose of abstraction is not to be vague,
but to create a new semantic level
in which one can be absolutely precise.

Edsger W. Dijkstra,
The Humble Programmer, 1972

Abstract

Field-sensitive static analyses of object-oriented code use approximations of the computational states where fields are taken into account, for better precision. This thesis presents a novel and sound *definite* analysis of Java bytecode that states two strictly related properties: field-sensitive unreachability between program variables and field-sensitive non-cyclicity of program variables. The latter exploits the former for better precision. We use abstract interpretation to approximate the concrete semantics and an analysis framework based on *constraint graphs*, whose nodes are program points and whose arcs propagate information according to the semantics of each bytecode instruction. Our analysis has been designed with the goal of improving client analyses such as *termination analysis*, asserting the non-cyclicity of variables w.r.t. specific fields.

Contents

List of Tables

Chapter 1

Introduction

Software and hardware are becoming ever more ubiquitous and hence ever more source of failures. Therefore in the past decades several theories and techniques, mainly based on computational logic and algebra, have been developed in order to accomplish the task of verification i.e., a formal specification of what the system should do and a formal proof that it meets this specification.

The topics of Temporal Logic and Model Checking are respectively used to specify complex behaviors of reactive and concurrent systems, such as embedded devices or web servers, and to check whether their satisfy given temporal properties [3]. On the other hand, Decision Procedures and Automated Theorem Proving involve algorithms that assert the satisfiability/validity of particular types of formulae, such as quantifier-free fragments of theories with equality, and also have the ability to produce counterexamples. In this way, for instance, it is possible to enrich a program text with annotations which are automatically validate to ensure that it obeys its specification [2].

The third major topic in verification area is *static analysis*, mainly specialized in analyzing the whole correctness of a piece of software. Actually the process of static analysis consists in building compile-time techniques in order to predict the set of values (such as states, duble precision real etc.) or behaviors arising dynamically at run-time i.e., during the computer program execution [12]. In this way it is possible to increase software quality by detecting illegal operations, such as a division by zero or a dereference of `null`, erroneous executions, such as infinite loops, or security flaws, such as unwanted disclosure of information.

Unfortunately most of these problems are formalized with properties that are, by Rice's Theorem[1], undecidable. For this reason, we need a method to approximate our analyses in order to be rough enough to be computable: we follow *abstract interpretation*, a semantics based framework defined by Cousot and Cousot [5] that formalizes the notion of approximation and abstraction in a mathematical setting, independent of specific language or application.

In modern object-oriented languages such as Java, a persistent problem regarding verification for real, large software programs is how the dynamic allocation of objects shapes the heap: namely they can be instantiated on demand and contain references to other objects through *fields* also changing during execution. Thus there are several papers in literature describing memory related properties. *Shape analysis* builds a general description of possible shapes that data structure might assume at run-time, *aliasing analysis* determines which variables point to the same heap location, whereas *sharing analysis* infers which variables are bound to overlap-

[1] The Rice's Theorem asserts that every non trivial property i.e., neither empty nor the description of all the Turing machines, of a Turing-complete programming languages is undecidable, [9].

ping data structures. Furthermore there are also more specific analysis such as *reachability*, that states whether it exists a path between pairs of locations pointed by two variables and *non-cyclicity*, that determines which variables are bound to non-cyclical data structures.

We note that all of these analyses can provide an over or an under approximation of the real behavior of the program with respect to the property that we want to state: in the former case we refer to them as a possible analyses, whereas in the latter as definite analyses.

In this context we present our definite analysis that introduces the notion of field-sensitive unreachability and non-cyclicity properties. Intuitively, we are interested in building an under approximation of program fields with respect to which a variable can not reach another one or, respectively, it can not reach a cyclic data structure. In other words, regarding the unreachability, we maintain, for each pair of variables $\langle v, w \rangle$, a set of fields F such that for each path linking the location pointed by v to the location pointed by w, this path does not go through any field in F. To analyze the non-cyclicity on the other hand, each variable is enriched with a set F representing the program fields that are not part of any cyclic path (i.e., cyclic data structure) reachable from the locations pointed by v.

We have chosen to create an analysis with a domain that takes into account both properties since we need information about unreachability to determine non-cyclicity for the corresponding variable: every time a cycle between the locations is created inside the heap, in order to compute the set of fields not involved in that cycle, we intersect the two sets of fields that contain the unreachability information between the two variables bounded to these locations. That is, we intersect the unreachability information of the former with respect to the latter and vice-versa.

This analysis has been designed to increase the precision of Julia[2], a Java and Android bytecode static analyzer that employs it as building blocks of two larger tools: a *nullness checker*, that tries to state in which points a program throws a null-pointer exception, and a *termination checker*, that determines which method calls might diverge at rut-time. For instance, if we are able to assert that a variable v is non-cyclical with respect to a field $\kappa.f : \mathsf{t}$, the Java instruction $v.f$ does not allow to reach v itself. In particular, the cyclicity property is crucial for the termination analyzer since it is based on the property of *path-length* that determines the maximal length of a path of pointers that can be followed from each program variable [22].

For example, given the Java instruction x = x.next, up to now we have asserted the path--length decreases (when accessing next field) only whether it was possible to assert the non-cyclicity of x. Contrariwise, by exploiting this novel analysis, we are now able to attest the decrease by distinguishing on the accessing field: the path-length surely decreases only if next is inside the non-cyclical set of fields F_x related to the variable x. Hence, we can argue that iterations over variables bound to cyclic data structures might diverge, unless the fields involved are in the set with respect to which the variable is non-cyclical.

Finally, the path-length and hence non-cyclicity properties are useful to build *resource usage analyzer*: an estimation of execution time and memory consumption could be performed by inferring a bound to the number of iterations of a loop which traverses the heap by its depth.

1.1 Related Work

Unreachabilty and non-cyclicity analysis belongs to the group of *shape analyses* for discovering how a program manipulates heap memory. In [19] the goal is to determine the shape invariants describing the program's data structures through the use of decision procedures: properties are encoded as first order-formulae and theorem proving is used to determine their validity.

[2] http://www.juliasoft.com

Although it provides a very precise approximation of the run-time heap memory, from which the particular properties can be extracted, this technique is computationally expensive (from the semi-decidability of first order logic starts all the troubles) and sometimes limited to some particular data structure (called First-Order Theories, such as the theories of lists or arrays).

On the other hand, our analysis can be viewed as a specific *pointer analysis* which statically determines the possible run-time values of a pointer respect to the specific property that we want to infer. There are several works in literature dealing with this kind of analyses [8] not only by providing a formal framework but also by introducing efficient tools.

First of all, we point out that a *field-sensitive pointer analysis* has already be developed by Pearce, Kelly, and Hankin [17]. Their approach is constraint based as ours but their results show how the field-sensitive pointer analysis, although more precise, is expensive to compute. Nevertheless, it is worth noting that this analysis does not work in an object-oriented framework with dynamically allocated instances, but rather than in the C imperative language, which only supports fields of structures. Let hence discuss more related pointer analyses i.e., based on object-oriented languages.

Sharing analysis [21] determines whether two variables might ever be bound to overlapping data structures i.e., two variables share if they might reach the same location at run-time. It is worth noting that two variables x and y can share in four different manners: 1) they are aliases, 2) x reaches y, 3) y reaches x, 4) they both reach a common location. For this reason other finer, more concrete abstraction of the computational states are introduce in order to state how two variables may or must share.

Definite aliasing analysis between a variable and an expression has been studied in [15] and we have heavily used to achieve precision. Although whenever two variables are alias, they are also reachable from each other, stating the relation between aliasing and unreachability with respect to a set of fields seems to be more complex. We have proved that a sufficient condition for aliasing is that two variables must be both reachable from each other and unreachable with respect to all the program fields.

A possible reachability analysis [13, 14] is even more relevant for our formalization since it is, in some sense, the dual of the unreachability property. Furthermore, if a variable is not reachable from another one, we can state that is also unreachable with respect to all the program fields. On the contrary we have proved that a necessary condition for reachability is that two variables must either have at least a program field with respect to which they are not unreachable or be alias.

However, the two most related formalizations to our analysis are *Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions* [18] and *Reachability-based Acyclicity Analysis by Abstract Interpretation* [6]. They have the same aim, discerning which variables are related with cyclic data structure and which not. The former introduces the non-cyclicity definition that we have extended: a variable v is non-cyclical if there is no cycle in its reachable locations. Therefore field-sensitive non-cyclicity is a finer abstraction of the simple non-cyclicity: as well as it is worth understanding how two variables share, here we want to assert with respect to which fields a variable is cyclical or not. On the other hand, the latter introduces an acyclicity analysis as a reduced product of two abstract domains: reachability and cyclicity, defined with the same semantics of our properties. It is highlighted that cyclicity is strictly related to reachability and when a data structure is modified by means of field update, `x.f = y`, a cyclic data structure is created only if `x` and `y` are aliases or `y` reaches `x`.

Comparing the static analyses introduced here and in [6], it is worth noting that:

- our target language is the low-level Java bytecode, whereas they use a high-level Java object-oriented language; pros and cons of this choice are explained in [11] and we discuss our choice in Chapter 3;

- we explicitly handle methods side effects without using shallow variables;
- our evaluation of expressions does not use any special variable;
- we provide a more detailed explanation of the propagation rules and formally prove them correct;
- our formalization is expressed in such a way that it is straightaway implementable.

Regarding more theoretical aspects related to our formalization, the under-approximation static analysis in the context of Abstract Interpretation has been studied by Schmidt [20] through the predicate transformers where the abstraction transition function is a sound postcondition transformer of the state-transition function. The paper investigates how to soundly under-approximate precondition and postcondition transformers, by showing why the expected abstraction of the postcondition transformer is unsound and by providing the correct one. An underapproximation analysis formalized in a similar way of ours can instead be found in [15].

Finally, the theoretical framework to refine abstract domains in order to enhance their precision has been provided by Giacobazzi and Ranzato [7]. They define the refinement operators as a monotonic and reductive maps i.e., preserving the greatest lower bound, in such a way that the resulting domain satisfies the same property but produces more concrete statements. Furthermore, they underline that the most familiar example of abstract domain refinement is the reduced product introduced by Cousot and Cousot [4] and exploited by Genaim and Zanardini [6] to combine their reachability and cyclicity domains.

1.2 Overview of the Thesis

In this thesis we introduce an abstract domain to state a set of program fields with respect to which a variable can not reach another one or can not reach a cyclic data structure. The rest of the thesis is organized as follows.

Chapter 2 introduces mathematical notations of the main mathematical structures (from sets to complete partial orders) and operators working on them. Furthermore fix-point and abstract interpretation theories are presented in an uniform and consistent way.

Chapter 3, after a discussion on the decision of a low-level analysis, describes the syntax and the semantics of the Java Bytecode language according to the Java Virtual Machine Specification [10]. Namely, it has been introduced six inference rules defining a small-step operational semantics between configurations (code block, state pairs), with a partial map showing how each instruction transforms the state.

The choice of the operational semantics allows to give the meaning of the program in terms of which steps of computation it makes when it runs. Moreover, this kind of programming language formalization allows a relatively simple way to prove the soundness of the assertion that we made through inductive techniques.

Chapter 4 introduces, inside this formalization, the notion of path between locations as a tuple of program fields such that starting from the object in the first location and going through the objects linked by the ordered fields in the tuple, we reach the object in the last. In this way, it has been possible to formalize the definition of unreachability and non-cyclicity with respect to a set of fields and to explain its relation to the reachability and aliasing analyses.

Chapter 5 provides an Abstract Interpretation of these definitions through an abstract powerset domain A_τ composed of both unreachability and non-cyclicity elements and a map γ_τ from elements inside A_τ to sets of states that are part of the corresponding concrete powerset domain C_τ. Furthermore it has been proved that the map is the concretization function of a Galois Connection between these two domains and it entails a consistent abstract interpretation i.e., it correctly approximates the concrete states of the program.

Chapter 6 is the main chapter since it is explained how this analysis works and on which others it depends: analyses of definite aliasing and of possible sharing and reachability.

The abstract transition function has been built through a well-know computable representation, a *constraint graph* of the program control flow graph, where each node corresponds to a bytecode instruction and each arc has a propagation rules associated i.e., a function that defines how the unreachability and non-cyclicity information inside the abstract element I at its source is propagated to establish the correspondent at its sink.

A generic fix-point algorithm is hence used to calculate the minimal solution of the constrain graph i.e., the least fixed point: the result is the field-sensitive unreachability and non-cyclicity analysis.

Chapter 7 provides a formal proof of the propagation rules soundness. Namely, it has been proved that the states produced by the application of an instruction to an abstraction set concretization $\mathsf{ins}(\gamma(I))$, are contained into the set of states deriving from the application of this concretization map to the correspondent propagation rule application, $\gamma(\Pi(I)$.

Finally, it has been proved the soundness of the whole analysis i.e., given an execution of the operational semantics, the final state is inside the set of states produced by the application of the concretization map to the final abstract set resulting by the analysis execution. In particular, to achieve the last proof it has been used mathematical induction on the length of the execution distinguishing on the basis of the transition rule of the operational semantics that is applied. It is worth noting that during this inductive proof, it has been heavily used the results of the Lemmas related to the propagation rules soundness.

Chapter 8 summarizes the main topics of the thesis, also linking it with a more theoretical result obtained by Logozzo and Fähndrich [11]. Furthermore it provides a possible direction on future works in order to state the effectiveness and the efficiency of the analysis.

Chapter 2

Preliminaries

We introduce here most of the notations and mathematical results which will be used throughout this thesis. Nevertheless, a deeper mathematical background related to the static analysis can be found in Nielson, Nielson, and Hankin [12].

2.1 Sets and tuples

A set is an unordered collection of elements. Given two sets S_1 and S_2, their Cartesian product is the set of pairs $S_1 \times S_2 = \{(s_1, s_2) \mid s_1 \in S_1 \wedge s_2 \in S_2\}$. The *powerset* of a set S is $\wp(S) = \{S' \mid S' \subseteq S\}$. The cardinality of a set S is denoted by $\mathsf{card}\, S$.

A tuple $\tilde{x} = \langle s_1, s_2, \ldots, s_n \rangle$ is an ordered collection of elements of S, so we can write $\tilde{x} \subseteq S$. In this way we use the common set relations and operations also between tuples and sets: the former will not change the nature of the tuples as ordered list of elements, contrariwise the latter will create a new set. Set operations between tuples are not allowed. Furthermore we denote the symbol '::' as concatenation between tuples: $\langle \hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n \rangle :: \langle \check{s}_1, \check{s}_2, \ldots, \check{s}_m \rangle = \langle \hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n, \check{s}_1, \check{s}_2, \ldots, \check{s}_m \rangle$.

2.2 Relations and functions

A *binary relation* between S and S' ($R\colon S \times S'$) is an element of $\wp(S \times S')$. We write $x\ R\ y$ for $(x, y) \in R$. A *partial function* from S to S' is a relation $f\colon S \times S'$ such that for any $x \in S$ and $\{y, y'\} \subseteq S'$ we have that $(x, y) \in f$ and $(x, y') \in f$ entails $y = y'$. By $f\colon S \rightharpoonup S'$ we denote a partial function of the set S (the *domain*) into the set S' (the *range*). We use the notation $f(x) = y$ when there exists y such that $(x, y) \in f$. In such a case, we say that $f(x)$ is defined. Otherwise, we say that $f(x)$ is undefined.

A (total) function f from S to S' is a partial function from S to S' such that, for all $x \in S$, there is some $y \in S'$ such that $f(x) = y$. Although total functions are a special kind of partial function, it is traditional to understand something described as simply a function to be a total function. So we will always say explicitly when a function is partial. To indicate that a function f from S to S' is total, we write $f\colon S \to S'$.

2.2.1 Lambda notation

It is sometimes useful to use the lambda notation to describe functions. It provides a way of referring to functions without having to name them. Suppose $f\colon S \to S'$ is a function which, for any element $x \in S$, gives a value $f(x)$ which is exactly described by the expression E, possibly involving x. Then we can write $\lambda x \in S.\, E$ for the function f. Thus, $(\lambda x \in S.\, E) = \{(x, E[x]) | x \in S\}$ and so $\lambda x \in S.\, E$ is just an abbreviation for the set of input-output values determined by the expression $E[x]$.

2.2.2 Composing relations and functions

We compose relations, and so partial and total functions, $R\colon S \times S'$ and $Q\colon S' \times S''$ by defining their *composition* (a relation between S and S'') as $Q \circ R = \{(x, z) \in S \times S'' \mid$ there exists $y \in S'$ such that $(x, y) \in R$ and $(y, z) \in Q\}$. We denote by R^n the relation

$$\underbrace{R \circ \cdots \circ R}_{n},$$

i.e., $R^1 = R$ and $R^{n+1} = R \circ R^n$. Each set S is associated with an identity function $Id_S = \{(x, x) \mid x \in S\}$, which is the neutral element of $\circ$. We define $R^0 = Id_S$.

The *function composition* of $g\colon S \rightharpoonup S'$ and $f\colon S' \rightharpoonup S''$ is the partial function $f \circ g\colon S \rightharpoonup S''$, where $(f \circ g)(x) = f(g(x))$, if $g(x)$ (first) and $f(g(x))$ (then) are defined, and is undefined otherwise. When it is clear from the context, $\circ$ will be omitted.

A function $f\colon S \to S'$ is *injective* (called also *one-to-one*) if and only if for each $\{x, y\} \subseteq S$ if $f(x) = f(y)$ then $x = y$; f is *surjective* (callled also *onto*) if and only if for each $x' \in S'$ there exists $x \in S$ such that $f(x) = x'$; f is *bijective* if it has an *inverse* $g\colon S' \to S$, i.e., if and only if there exists a function g such that $g \circ f = Id_S$ and $f \circ g = Id_{S'}$. Then the sets S and S' are said to be in one-to-one correspondence. Any set in one-to-one correspondence with a subset of natural numbers is said to be countable. Note that a function f is bijective if and only if it is injective and surjective.

2.2.3 Direct and inverse image of a relation

We extend relations, and thus partial and total functions, $R\colon S \times S'$ to functions on subsets by taking $R(X) = \{y \in S' \mid$ there exists $x \in X.\, (x, y) \in R\}$ for $X \subseteq S$. The set $R(X)$ is called the *direct image* of X under R. Therefore, if $f\colon S \rightharpoonup S'$ is a partial function and $X \subseteq S$, we denote by $f(X)$ the *image* of X under f, i.e., the set $f(X) = \{f(x) \mid x \in X\}$.

2.2.4 Equivalence Relations and Congruences

An *equivalence relation* $\approx$ on a set S is a binary relation on S ($\approx\colon S \times S$) such that, for each $\{x, y, z\} \subseteq S$, we have

$x\ R\ x$	(reflexivity)
$x\ R\ y$ entails $y\ R\ x$	(symmetricity)
$x\ R\ y$ and $y\ R\ z$ entails $x\ R\ z$	(transitivity).

The *equivalence class* of an element $x \in S$ w.r.t. $\approx$ is the subset $[x]_\approx = \{y \mid x \approx y\}$. When clear from the context, we abbreviate $[x]_\approx$ by $[x]$ and often abuse notation by letting the elements of a set denote their correspondent equivalence classes. The *quotient set* $S/_\approx$ of S modulo $\approx$ is the set of equivalence classes of elements in S w.r.t. $\approx$.

An equivalence relation $\approx$ on S is a *congruence* w.r.t. a partial function $f : S^n \rightharpoonup S$ if and only if, given $\{a_i, b_i\} \subseteq S$ with $i = 1, \ldots, n$, if $f(a_1, \ldots, a_n)$ is defined then also $f(b_1, \ldots, b_n)$ is defined and $f(a_1, \ldots, a_n) \approx f(b_1, \ldots, b_n)$. Then, we can define the partial function $f_{\approx} : (S/_{\approx})^n \rightharpoonup S/_{\approx}$ as

$$f_{\approx}([a_1]_{\approx}, \ldots, [a_n]_{\approx}) = [f(a_1, \ldots, a_n)]_{\approx}$$

since, given $[a_1]_{\approx}, \ldots, [a_n]_{\approx}$, the class $[f(a_1, \ldots, a_n)]_{\approx}$ is uniquely determined independently from the choice of the representatives $a_1, \ldots, a_n$.

2.3 Terms and Substitutions

Given a set of variables V, a set of function symbols Σ with associated arity and $k \in \mathbb{N}$, we define the set of terms as

$$\mathsf{terms}(\Sigma, V) = \bigcup_{k \geq 0} \mathsf{terms}^k(\Sigma, V)$$

where $\mathsf{terms}^k(\Sigma, V)$ is an inductive definition:

$$\mathsf{terms}^k(\Sigma, V) = \begin{cases} V & \text{if k=0} \\ \mathsf{terms}^{k-1}(\Sigma, V) \cup \left\{ \mathtt{f}(t_1, \ldots, t_n) \;\middle|\; \begin{array}{l} \mathtt{f}^n \in \Sigma \wedge \\ \{t_1, \ldots, t_n\} \subseteq \mathsf{terms}^k(\Sigma, V) \end{array} \right\} & \text{if k>0} \end{cases}$$

We assume that Σ contains at least a symbol of arity 0. We denote by $\mathsf{vars}(t)$ the set of variables which occur in a term t. When $\mathsf{vars}(t) = \emptyset$ we say that the term t is ground. Given a set of variables V and a variable x, $V \cup x$ means $V \cup \{x\}$ and $V \setminus x$ means $V \setminus \{x\}$.

A substitution θ is a map from variables into terms. We define the sets $\mathsf{dom}(\theta) = \{x \mid \theta(x) \neq x\}$ and $\mathsf{rng}(\theta) = \cup_{x \in \mathsf{dom}(\theta)} \mathsf{vars}(\theta(x))$. We require $\mathsf{dom}(\theta)$ to be finite. This allows us to represent a substitution θ in extensional way as $\theta = \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$, meaning that $\theta(v_i) = t_i$ for all $i = 1, \ldots, n$ and $\theta(v) = v$ for every $v \in V \setminus \{v_i \mid i = 1, \ldots, n\}$.

We define $\Theta^Z_{V,W}$, with $Z \subseteq V \cap W$, as the set of substitutions θ such that $\mathsf{dom}(\theta) \subseteq V$, $\theta(x) \in \mathsf{terms}(\Sigma, W)$ for every $x \in V$ and $\mathsf{dom}(\theta) \cap \mathsf{rng}(\theta) \subseteq Z$. If $Z = \emptyset$ we omit the superscript. The elements of $\Theta_{V,W}$ are called idempotent substitutions. We write Θ_V for $\Theta_{V,V}$ and Θ^Z_V for $\Theta^Z_{V,V}$. A substitution θ is called grounding for a set of variables G if $\theta(x)$ is ground for every $x \in G$. Given θ and a set of variables R, we define $\theta|_R(x) = \theta(x)$ if $x \in R$ and $\theta|_R(x) = x$ otherwise.

Given a term $t \in \mathsf{terms}(\Sigma, V)$ and $\theta \in \Theta^Z_{V,W}$, $t\theta \in \mathsf{terms}(\Sigma, W)$ is the term obtained with parallel substitution of every variable x in t with $\theta(x)$. We often write $t[t'/x]$ for $t\{x \mapsto t'\}$ for any $t' \in \mathsf{terms}(\Sigma, V)$.

Given a substitution σ and $\{x, n\} \subseteq V$, we define the substitution $\sigma[n/x]$ as $\sigma[n/x](x) = x$, $\sigma[n/x](n) = \sigma(x)[n/x]$ and $\sigma[n/x](y) = \sigma(y)[n/x]$ if $y \neq x$ and $y \neq n$. Composition of substitutions $\theta \in \Theta_{V,W}$ and $\sigma \in \Theta_{W,Z}$ is defined as $(\theta\sigma)(x) = \theta(x)\sigma$ for every $x \in V \cup W$. We recall that composition of substitutions is associative, the empty substitution ε is the neutral element and, for each term t, $t(\theta\sigma) = (t\theta)\sigma$. A *renaming* is a substitution ρ for which there exists ρ^{-1}, such that $\rho\rho^{-1} = \rho^{-1}\rho = \varepsilon$.

2.4 Complete Partial Orders and Lattices

A binary relation $\leq$ on S ($\leq\colon S \times S$) is a *partial order* if, for each $\{x, y\} \subseteq S$,

$x \leq x$	(reflexivity)
$x \leq y$ and $y \leq x$ entails $x = y$	(antisymmetry)
$x \leq y$ and $y \leq z$ entails $x \leq z$	(transitivity).

A *partially ordered set* (poset) $(S, \leq)$ is a set S equipped with a partial order $\leq$. A set S is *totally ordered* if it is partially ordered and, for each $\{x, y\} \subseteq S$, we have $x \leq y$ or $y \leq x$. A *chain* is a (possibly empty) totally ordered subset of S.

A *preorder* is a binary relation which is reflexive and transitive. A preorder $\leq$ on a set S induces on S an equivalence relation $\approx$ defined as follows: for each $\{x, y\} \subseteq S$, $x \approx y$ if and only if $x \leq y$ and $y \leq x$. Moreover, $\leq$ induces on $S/_{\approx}$ the partial order $\leq_{\approx}$ such that, for each $\{[x]_{\approx}, [y]_{\approx}\} \subseteq S/_{\approx}$, we have $[x]_{\approx} \leq_{\approx} [y]_{\approx}$ if and only if $x \leq y$.

If $(S, \leq)$ is a preorder and $S' \subseteq S$, S' is *downward closed* if and only if from $s_1 \in S'$ and $s_2 \leq s_1$ it follows that $s_2 \in S'$.

A binary relation $<$ on S is *strict* if and only if it is anti-reflexive (i.e., $x < x$ does not hold for every $x \in S$) and transitive. Given a poset $(S, \leq)$ and $X \subseteq S$, $y \in S$ is an *upper bound* for X if and only if for each $x \in X$ we have $x \leq y$. Moreover, $y \in S$ is the *least upper bound* (called also join) of X, if y is an upper bound of X and, for every upper bound y' of X, $y \leq y'$. A least upper bound of X is often denoted by $\mathsf{lub}_S X$ or by $\bigsqcup_S X$. We also write $\bigsqcup_S \{d_1, \ldots, d_n\}$ as $d_1 \sqcup_S \cdots \sqcup_S d_n$. Dually, an element $y \in S$ is a *lower bound* for X if and only if for each $x \in X$ we have $y \leq x$. Moreover, $y \in S$ is the *greatest lower bound* (called also meet) of X, if y is a lower bound of X and for every lower bound y' of X we have $y' \leq y$. A greatest lower bound of X is often denoted by $\mathit{glb}_S X$ or by $\bigsqcap_S X$. We also write $\bigsqcap_S \{d_1, \ldots, d_n\}$ as $d_1 \sqcap_S \cdots \sqcap_S d_n$. When it is clear from the context, the subscript S will be omitted. It is easy to check that if lub and *glb* exist, then they are unique.

A *direct set* is a poset in which any subset of two elements (and hence any finite subset) has an upper bound in the set. A *complete partial order* (*CPO*) S is a poset such that every chain D has a least upper bound (i.e., $\bigsqcup_S D$ exists).

A *complete lattice* is a poset $(S, \leq)$ such that for every subset X of S there exists $\bigsqcup X$ and $\bigsqcap X$. We let $\top$ denote the *top element* $\bigsqcup S = \bigsqcap \emptyset$ and $\bot$ denote the *bottom element* $\bigsqcap S = \bigsqcup \emptyset$ of S.

Let $(L, \leq)$ and $(M, \sqsubseteq)$ be two posets. A function $f\colon L \to M$ is *monotonic* if and only if for every $\{x, y\} \subseteq L$ such that $x \leq y$ we have $f(x) \sqsubseteq f(y)$. Moreover, f is *continuous* if and only if for each non empty chain $D \subseteq L$ we have $f(\bigsqcup_L D) = \bigsqcup_M f(D)$. Every continuous function is also monotonic. A function $f\colon S \to S'$ is *additive* if and only if the previous continuity condition is satisfied for each non empty set. Hence, every additive function is also continuous. Co-continuity and co-additivity are defined dually.

It can be proved that composition of monotonic, continuous or additive functions is monotonic, continuous or additive, respectively.

The mathematical way of expressing that structures are "essentially the same" is through the concept of isomorphism. A continuous function $f\colon D \to E$ between two *CPO*s D and E is said to be an *isomorphism* if there is a continuous function $g\colon E \to D$ such that $g \circ f = Id_D$ and $f \circ g = Id_E$. Thus f and g are mutual inverses. It follows from the definition that isomorphic *CPO*s are essentially the same but for a renaming of elements. A function $f\colon D \to E$ is an isomorphism if and only if f is bijective and continuous.

2.5 Fixpoint Theory

Given a poset $(S, \leq)$ and a function $f\colon S \to S$, a *fixpoint* of f is an element $x \in S$ such that $f(x) = x$. A *pre-fixpoint* of f is an element $x \in S$ such that $f(x) \leq x$ and dually a *post-fixpoint* of f is an element $x \in S$ such that $x \leq f(x)$. Moreover, we say that $x \in S$ is the *least fixpoint of* f (denoted by $lfpf$ or $\mu t.f(t)$) if and only if x is a fixpoint of f and for all fixpoints y of f we have $x \leq y$. Dually, we define the *greatest fixpoint* (denoted by $gfpf$).

A fundamental theorem by Knaster-Tarski states that the set $\mathsf{fp}(f)$ of fixpoints of a monotonic function f is a complete lattice.

Theorem 2.1 (Fixpoint theorem [26])**.** *A monotonic function f on a complete lattice $(L, \leq)$ has a least and a greatest fixpoint. Moreover, we have*

$$lfp(f) = \bigsqcap\{x \mid f(x) \leq x\} = \bigsqcap\{x \mid x = f(x)\}$$
$$gfp(f) = \bigsqcup\{x \mid x \leq f(x)\} = \bigsqcup\{x \mid x = f(x)\}\ .$$

The Knaster-Tarski Theorem is important because it applies to any monotone function on a complete lattice. However, most of the time we will be concerned with least fixpoints of continuous functions. Therefore, it is useful to state some more notations and results on fixpoints of functions defined on (complete) lattices.

2.6 Abstract Interpretation and Galois Connections

Abstract interpretation is a theory developed to reason about the abstraction relation between two different semantics (the *concrete* and the *abstract* semantics). The idea of approximating program properties by evaluating a program on a simpler domain of descriptions of *concrete* program states goes back to the early 70's. The inspiration was that of approximating properties from the exact (concrete) semantics into an approximate (abstract) semantics, that explicitly exhibits a structure (e.g. ordering) which is somehow present in the richer concrete structure associated to program execution.

The guiding idea is to relate the concrete and the abstract interpretation of the calculus by a pair of functions, *abstraction* α and *concretisation* γ, which form a Galois connection. Galois connections are used to formalise this relation between abstract and concrete meaning of a computation.

Let $(\mathsf{C}, \sqsubseteq_{\mathsf{C}})$ (concrete domain) be the domain of the concrete semantics, while $(\mathsf{A}, \sqsubseteq_{\mathsf{A}})$ (abstract domain) be the domain of the abstract semantics. The partial order relations reflect an approximation relation. Since in approximation theory a partial order specifies the precision degree of any element in a poset, it is obvious to assume that if α maps every concrete element in $(\mathsf{C}, \sqsubseteq_{\mathsf{C}})$ into an abstract element in $(\mathsf{A}, \sqsubseteq_{\mathsf{A}})$ then the following holds: if $\alpha(x) \sqsubseteq_{\mathsf{A}} y$, then y is also a correct, although less precise, approximation of x. The same argument holds if $x \sqsubseteq_{\mathsf{C}} \gamma(y)$. Then y is also a correct approximation of x, although x provides more accurate information than $\gamma(y)$. This is formally defined below.

Definition 2.1 (Galois connection and insertion)**.** *Let $(\mathsf{C}, \sqsubseteq_{\mathsf{C}})$ and $(A, \sqsubseteq_{\mathsf{A}})$ be two posets (the concrete and the abstract domain). A* Galois connection *$\langle \alpha, \gamma \rangle\colon (\mathsf{C}, \sqsubseteq_{\mathsf{C}}) \rightleftharpoons (\mathsf{A}, \sqsubseteq_{\mathsf{A}})$ is a pair of maps $\alpha\colon C \to A$ and $\gamma\colon A \to C$ such that*

1. *α and γ are monotonic,*
2. *for each $x \in \mathsf{C}$ we have $x \sqsubseteq_{\mathsf{C}} (\gamma \circ \alpha)(x)$, and*

3. for each $y \in \mathsf{A}$ *we have* $(\alpha \circ \gamma)(y) \sqsubseteq_{\mathsf{A}} y$.

Moreover, a Galois insertion *(of* $(\mathsf{C}, \sqsubseteq_{\mathsf{C}})$ *into* $(\mathsf{A}, \sqsubseteq_{\mathsf{A}})$*)* $\langle \alpha, \gamma \rangle \colon (\mathsf{C}, \sqsubseteq_{\mathsf{C}}) \rightleftharpoons (\mathsf{A}, \sqsubseteq_{\mathsf{A}})$ *is a Galois connection where* $\alpha \circ \gamma = Id_A$*. In other words,* γ *is injective and hence point 3 becomes:*

$$\textit{for each } y \in \mathsf{A} \textit{ we have } (\alpha \circ \gamma)(y) = y$$

Property 2 is called *extensivity* of $\gamma\alpha$. The map α (γ) is called the *abstraction* (*concretisation*) function in the context of abstract interpretation.

In view of the compositional design of abstract interpretations we have that the composition of Galois insertions is a Galois insertion.

The following basic properties are satisfied by any Galois connection.

1. γ is injective if and only if α is surjective if and only if $\alpha \circ \gamma = Id_A$.
2. α is additive and γ is co-additive.
3. The abstraction map uniquely determines the concretisation map and vice versa. Namely,

$$\gamma(y) = \bigsqcup\nolimits_{\mathsf{C}} \{x \in \mathsf{C} \mid \alpha(x) \sqsubseteq_{\mathsf{A}} y\}\,, \qquad \alpha(x) = \bigsqcap\nolimits_{\mathsf{A}} \{y \in \mathsf{A} \mid x \sqsubseteq_{\mathsf{C}} \gamma(y)\}\,.$$

Conversely, if C and A are complete lattices and $\alpha \colon C \to A$ is additive or $\gamma \colon A \to C$ is co--additive (also called multipicative), then $\langle \alpha, \gamma \rangle$ is a Galois connection from C to A.

When, in a Galois connection $\langle \alpha, \gamma \rangle$, γ is not one-to-one, several distinct elements of the abstract domain $(A, \leq)$ have the same meaning (by γ). Therefore, the abstract domain contains redundancy [4]. Therefore, a Galois insertion can always be forced by collapsing abstract elements denoting the same concrete element into a single element, and the result is an abstract domain containing no redundant elements. This process is known as *reduction* of the abstract domain. It ensures that any abstract element is the image of some concrete element or, equivalently, that the abstraction function is surjective.

A reduction of a Galois connection $\langle \alpha, \gamma \rangle \colon (C, \leq) \rightleftharpoons (A, \leq)$ is $\langle \alpha, \gamma \rangle \colon (C, \leq) \rightleftharpoons (\alpha(C), \leq)$. If $\langle \alpha, \gamma \rangle \colon (C, \leq) \rightleftharpoons (\alpha(C), \leq)$ is a Galois insertion then $\alpha(C)$ is isomorphic to $\gamma(\alpha(C))$ and $\langle \gamma\alpha, id \rangle \colon (C, \leq) \rightleftharpoons (\gamma(\alpha(C)), \leq)$ is also a Galois insertion. This allows us to consider Galois insertions of type $\langle \rho, id \rangle \colon (C, \leq) \leftrightharpoons (\rho(C), \leq)$ only, with $\rho \colon C \to C$. The abstraction function and the abstract domain uniquely determine each other. Hence, to specify a Galois insertion $\langle \rho, id \rangle \colon (C, \leq) \leftrightharpoons (A, \leq)$, it suffices to give either the abstraction map ρ (and the abstract domain will be $\rho(C)$) or an abstract domain $A \subseteq C$ (and the abstraction function will be the unique function $\rho \colon C \to C$ such that $\langle \rho, id \rangle \colon (C, \leq) \rightleftharpoons (\rho(C), \leq)$ is a Galois insertion and $\rho(C) = A$).

Chapter 3

Syntax and Semantics

Before starting to describe the syntax and the semantics of our language, it is worth discussing the decision to perform a low-level analysis, i.e., on a low-level program language as Java Bytecode.

Java Bytecode instructions work over states, by affecting their operand stack, local variables, or memory. There are more than 100 Java bytecode instructions. However, many of them are similar and differ only in the type of their operands. Others are not relevant in this article, such as those that perform tedious but useful stack manipulations. Hence we concentrate here on a very restricted set of 14 instructions only, that exemplify the operations that the Java Virtual Machine performs: stack manipulation, arithmetics, interaction between the stack and the local variables set, object creation and access, and method call.

The choice of a low-level analysis was made because it provides several advantages compared to the high-level version as explained in [11]. First of all it is more faithful, as it analyzes the code that is actually executed and it enables the analysis of libraries when source code is not available. Moreover the analyzer avoids redundant work that the compiler performs, such as name resolution and type checking, and, since the semantics of high-level constructs are expanded by the compiler, it also needs to deal with many fewer constructs, reducing its complexity. Finally the analyzer can be high-level language independent; e.g., analyzing the common Java bytecode allows to handle all the programming languages that run on the Java Virtual Machine.

On the contrary, the low-level nature of the bytecode presents others issues, such as the unstructuredness of the code and the presence of an operand stack of variable height. Furthermore it was shown that the analysis of the low-level languages is typically less precise than the high-level equivalent, unless we use a refined abstract domains and pre-processing [11]. In order to overcome these troubles and hence improve the precision, we have heavily exploited other analyses, in particular *possible reachability*, *definite aliasing* and *possible sharing*, as we will describe in Chapter 6.

The following sections present a formalization of Java Bytecode introduced by [25].

3.1 Control Flow Graph

First of all, we introduce the syntax that we are going to use to formalize our analysis. We assume that the only primitive type is int and that reference types are *classes* containing *instance fields* and *instant method* only. The implementation handles all Java types and bytecodes, as well as classes with static fields and static methods.

```
1  class Element{
2          public Object value;
3          public Element prec, next;
4
5          public Element(Object value){
6                  this.value=value;
7          }
8          public Element(Object value, Element prec){
9                  this.value=value;
10                 this.prec=prec;
11                 prec.next=this;
12         }
13 }
14 public class MWexample{
15         public static void main(String[] args){
16                 Element top = new Element(new Integer(0));
17                 for(int i=1;i<=3;i++)
18                         top = new Element(new Integer(i),top);
19         }
20 }
```

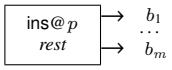
Code 1: Our Minimal Working Example.

```
1  aload_0
2  invokespecial #1 <java/lang/Object/<init>()V>
3  aload_0
4  aload_1
5  putfield #2 <Element/value Ljava/lang/Object;>
6  aload_0
7  aload_2
8  putfield #3 <Element/prec LElement;>
9  aload_2
10 aload_0
11 putfield #4 <Element/next LElement;>
12 return
```

Code 2: Bytecode Instructions of the Element second constructor, Code 1

In order to analyze the bytecode instructions we prepocess each method into a data structure, widely used in compilers [1]: the *control flow graph* (CFG), a directed graph where nodes represent basic blocks and edges represent possible transfer of control flow from one basic block to another. It is built on top of the intermediate code (bytecode) representation abstracting the control flow behavior of each method or constructor that is been compiled.

A simple method to build the control flow graph is provided by Aho, Sethi, and Ullman [1]. Namely, they determine the *leader* of each basic block (i.e., their first statement) as any statement that is the target or immediately follow a goto, either conditional or unconditional. Furthermore all the bytecode instructions that can throw an exception are handled as goto. Therefore, a basic block consists in a *leader* and all the other instruction before the next one. We graphically write:



to denote a block of code starting with a bytecode instruction ins at program point p, possibly followed by more bytecode instructions *rest* and linked to m subsequent block $b_1, \ldots, b_m$. The program point p is often irrelevant, so we write just ins instead of ins@p.

An exception handler starts with a catch bytecode. A conditional, virtual method call, or selection of an exception handler is translated into a block linked to many subsequent blocks. Each of these subsequent blocks starts with a filtering bytecode, such as excp_is K for exceptional handlers. We note that bytecode is a stack-based programming language i.e., it operates directly on a stack machine for passing parameters. For this reason, we can group the *variables* in stack elements $\{s_0, \ldots, s_n\}$ and local variables $\{l_0, \ldots, l_m\}$.

Example 3.1. *Figure 3.1 shows the control flow graph of bytecode instructions in Code 2, correspondent to the second constructor of the* Element *class (Code 1, lines 8-12). We note how the bytecode instructions' names are slightly differences in Figure 3.1 respect to 2. That is because, as already explained at the beginning of the Chapter, we "abstract" from their specific features, by collapsing them depending on the state manipulation.*

There is a branch at the call to the constructor of java.lang.Object*, that might throw an exception (like every call). If this happens, the exception is first caught and then re-thrown to the caller f the constructor. Otherwise, the execution continues with* 2 *blocks storing the formal parameters (locals* 1 *and* 2*) into the fields of* this *(local* 0*) and then returns.*

Note that each bytecode instruction except return *and* throw *has always one or more immediate successor bytecodes, while* return *and* throw *are placed at the end of a method or constructor and typically have no successors.*

3.2 The Java Virtual Machine Formalization

We introduce our formalization in order to mathematically handle the the Java Virtual Machine (JVM) structure and since one of our goals is to prove our analysis correct. This formalization, already introduced in [13, 14], seeks to be as consistent as possible to the specification provided by Lindholm and Yellin [10]. In particular, as we are going to explain in Sections 3.3 and 3.4, our semantics keeps a *state* that maps program variables to values, whereas an *activation stack* of states models the method call mechanism, exactly in the actual implementation of the JVM.

First of all, we introduce the concept of *class*. In the object-oriented programming a class is a constructor used to create instances of itself called *objects* in such a way that they have state and behavior. In other words, a class is the blueprint from which individual objects are created.

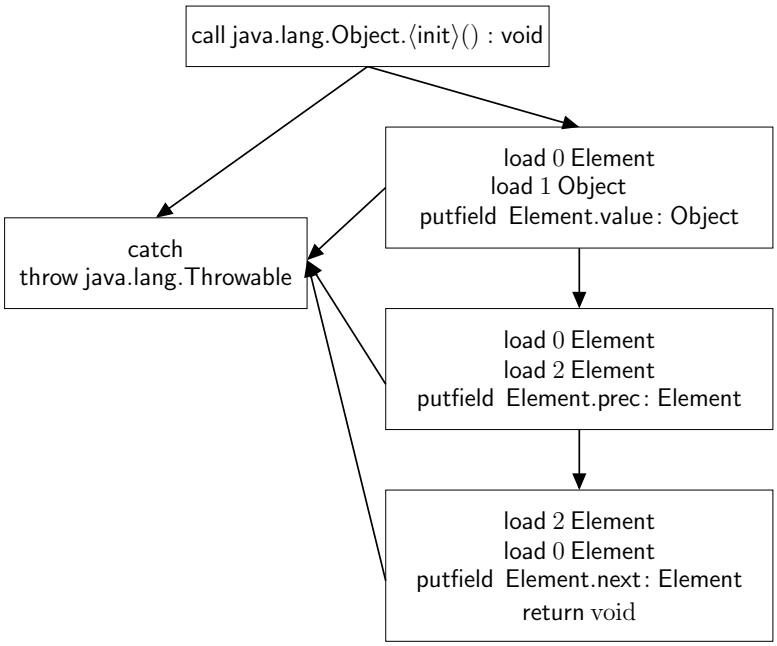


Figure 3.1: Control Flow Graph of Java Bytecode Instructions in Code 2.

Furthermore, classes define the *data type* of their instances i.e., the possible values that they may assume at run-time.

Definition 3.1 (Classes)**.** *The set of* classes $\mathbb{K}$ *of a program is partially ordered w.r.t. the* subclass relation $\leq$*:* $\mathsf{t} \leq \mathsf{t}'$ *if* t *(respectively* t'*) is a subclass (respectively superclass) of* t' *(respectively* t*). Every class has at most one* direct superclass *and an arbitrary number of direct subclasses.*
A type *is an element of* $\mathbb{T} = \{\mathsf{int}\} \cup \mathbb{K}$*, ordered by the extension of* $\leq$ *with* $\mathsf{int} \leq \mathsf{int}$*.*
A class $\kappa \in \mathbb{K}$ *has* fields $\kappa.f : \mathsf{t}$ *(field* f *of type* $\mathsf{t} \in \mathbb{T}$ *defined in* κ*), where* κ *and* t *are often omitted. We let* $\mathbb{F}(\kappa) = \{\kappa'.f : \mathsf{t}' \mid \kappa \leq \kappa'\}$ *be the fields defined in* κ *or in any of its superclasses. Furthermore, we define the set of program fields as* $\mathcal{F} = \bigcup_{\kappa \in \mathbb{K}} \mathbb{F}(\kappa)$*.*
A class κ *has* methods $\kappa.m(\vec{\mathsf{t}}) : \mathsf{t}$ *(method* m*, defined in* κ*, with arguments of type* $\vec{\mathsf{t}}$*, returning a value of type* $\mathsf{t} \in \mathbb{T} \cup \{\mathsf{void}\}$*), where* κ*,* $\vec{\mathsf{t}}$*, and* t *are often omitted.*
Constructors are methods named init *that return* void*.*

Lemma 3.1. *Consider a type* $\mathsf{t} \in \mathbb{T}$ *and let* t' *and* t'' *be two distinct supertypes of* t*, i.e.,* $\mathsf{t} \leq \mathsf{t}', \mathsf{t} \leq \mathsf{t}'' and \mathsf{t}' \neq \mathsf{t}''$*. Then one of the following relations holds:* $\mathsf{t}' \leq \mathsf{t}''$ *or* $\mathsf{t}'' \leq \mathsf{t}'$*.*

Proof. If $\mathsf{t} = \mathsf{int}$ then $\mathsf{t}' = \mathsf{t}'' = \mathsf{int}$ and both $\mathsf{t}' \leq \mathsf{t}''$ and $\mathsf{t}'' \leq \mathsf{t}'$ trivially hold. Otherwise t, t' and t'' are classes. Since every class has at most one direct superclass (Definition 3.1), by starting at t and going up through the superclass chain, one must find t' and then t'' or t'' and then t'. In the former case $\mathsf{t}'' \leq \mathsf{t}'$, in the latter $\mathsf{t}' \leq \mathsf{t}''$. □

Definition 3.2 (Compatible types)**.** *We define a function* $\mathsf{compatible} : \mathbb{T} \to \wp(\mathbb{T})$*, mapping every type* $\mathsf{t} \in \mathbb{T}$ *to the set of its* compatible *types:*

$$\mathsf{compatible}(\mathsf{t}) = \{\mathsf{t}' \mid \mathsf{t} \leq \mathsf{t}' \text{ or } \mathsf{t}' \leq \mathsf{t}\}$$

The following lemma shows that if a type is compatible with another one, then every superclass of the former is compatible with the latter as well.

Lemma 3.2. *Let* $\mathsf{t}, \mathsf{t}', \mathsf{t}'' \in \mathbb{T}$ *with* $\mathsf{t}' \leq \mathsf{t}''$. *If* $\mathsf{t}' \in \mathsf{compatible}(\mathsf{t})$, *then* $\mathsf{t}'' \in \mathsf{compatible}(\mathsf{t})$.

Proof. Since $\mathsf{t}' \in \mathsf{compatible}(()\mathsf{t})$ we have two cases:

- $\mathsf{t} \leq \mathsf{t}'$. Hence $\mathsf{t} \leq \mathsf{t}''$ and $\mathsf{t}'' \in \mathsf{compatible}(\mathsf{t})$;
- $\mathsf{t}' \leq \mathsf{t}$. Since $\mathsf{t} \leq \mathsf{t}''$, by Lemma 3.1, we have $\mathsf{t} \leq \mathsf{t}''$ or $\mathsf{t}'' \leq \mathsf{t}$ i.e., $\mathsf{t}'' \in \mathsf{compatible}(\mathsf{t})$.

□

We show that the function compatible is monotonic.

Lemma 3.3. *Let* $\mathsf{t}, \mathsf{t}'' \in \mathbb{T}$ *with* $\mathsf{t}' \leq \mathsf{t}''$. *Then* $\mathsf{compatible}(\mathsf{t}') \subseteq \mathsf{compatible}(\mathsf{t}'')$.

Proof. Let $\mathsf{t} \in \mathsf{compatible}(\mathsf{t}')$. We have two cases:

- $\mathsf{t} \leq \mathsf{t}'$. Hence $\mathsf{t} \leq \mathsf{t}''$ and $\mathsf{t} \in \mathsf{compatible}(\mathsf{t}'')$;
- $\mathsf{t}' \leq \mathsf{t}$. Since $\mathsf{t}' \leq \mathsf{t}''$, by Lemma 3.1, we have $\mathsf{t} \leq \mathsf{t}''$ or $\mathsf{t}'' \leq \mathsf{t}$ i.e., $\mathsf{t} \in \mathsf{compatible}(\mathsf{t}'')$.

□

We recall that bytecode instructions operate on *variables*, which encompass both stack elements $\{s_0, \ldots, s_n\}$ and local variables $\{l_0, \ldots, l_m\}$. A standard algorithm, provided by Lindholm and Yellin [10], infers their static types.

Definition 3.3 (Type environment)**.** *Let* $V = L \cup S$ *be the set of* variables *from* $L = \{l_0, \ldots, l_m\}$ *(*local *variables) and* $S = \{s_0, \ldots, s_n\}$ *(*stack *variables). A* type environment *is a function* $\tau \colon V \to \mathbb{T}$. *Its* domain *is written as* $\mathsf{dom}(\tau)$. *The set of all type environments is* $\mathcal{T}$.

Definition 3.4 (State)**.** *A* value *is an element of* $\mathbb{V} = \mathbb{Z} \cup \mathbb{L} \cup \{\mathtt{null}\}$, *where for simplicity we use* $\mathbb{Z}$ *instead of* 32*-bit two's-complement integers as in the actual* JVM *(this choice is irrelevant in this paper) and where* $\mathbb{L}$ *is an infinite set of* memory locations.
A state *over* $\tau \in \mathcal{T}$ *is a pair* $\langle\langle l \parallel s\rangle, \mu\rangle$ *where* l *is an array of values for the local variables in* $\mathsf{dom}(\tau)$, s *is a stack of values for the stack variables in* $\mathsf{dom}(\tau)$, *which grows leftwards, and* μ *is a* memory, *or* heap, *that binds locations to* objects. *The empty stack is denoted by* ε.
We often use another representation for a state: $\langle \rho, \mu \rangle$, *where an* environment ρ *maps each* $l_k \in L$ *to its value* $l[k]$ *and each* $s_k \in S$ *to its value* $s[k]$.
An object o *has class* $o.\kappa$ *(is an* instance *of* $o.\kappa$*) and has an internal environment* $o.\phi \colon \mathbb{F}(o.\kappa) \to \mathbb{V}$ *that maps every field* $\kappa'.f : \mathsf{t}' \in \mathbb{F}(o.\kappa)$ *into its value* $(o.\phi)(\kappa'.f : \mathsf{t}')$.
A value v has type t *in* $\langle \rho, \mu \rangle$ *if:* $v \in \mathbb{Z}$ *and* $\mathsf{t} = \mathsf{int}$, *or* $v = \mathtt{null}$ *and* $t \in \mathbb{K}$, *or* $v \in \mathbb{L}$, $t \in \mathbb{K}$ *and* $\mu(v).\kappa \leq \mathsf{t}$.
In a state $\langle \rho, \mu \rangle$ *over* τ, *we require that* $\rho(v)$ *has type* $\tau(v)$ *for any* $v \in \mathsf{dom}(\tau)$ *and* $(o.\phi)(\kappa'.f : \mathsf{t}')$ *has type* t' *for every* $o \in \mathsf{rng}(\mu)$ *(range* μ*) and every* $\kappa'.f : \mathsf{t}' \in \mathbb{F}(o.\kappa)$.
The set of states is Ξ. *We write* Ξ_τ *when we want to fix the type environment* τ.

As expressed at the end of Definition 3.4, states are well-typed, i.e., each variable and field holds a value consistent with its declared, static type. The Java Virtual Machine supports exceptions. Hence we distinguish normal states Ξ arising during the normal execution of a piece of code, from exceptional states $\underline{\Xi}$ arising just after a bytecode that throws an exception. States in $\underline{\Xi}$ always have a stack of height 1 containing a location (bound to the thrown exception object). When we denote a state by σ, we do not specify whether it is normal or exceptional. If we want to stress that fact, we write $\langle\langle l \parallel s\rangle, \mu\rangle$ for a normal state and $\underline{\langle\langle l \parallel s\rangle, \mu\rangle}$ for an exceptional state.

Definition 3.5 (JVM State)**.** *The set of* JVM *states (from now on just* states*) in type environment* $\tau \in \mathcal{T}$ *is* $\Sigma_\tau = \Xi_\tau \cup \underline{\Xi}_\tau$*, where* τ' *is* τ *with only one stack variable whose* `type` *is a subclass of* `Throwable`*.*

Figure 3.2 represents the JVM state $\sigma = \langle \rho, \mu \rangle$, at the end of the first execution of the second constructor (line 12, Code 2) of the class Element in method main of Code 1. In that configuration, with respect to the formal parameters of that constructor, we have i = $l_2 = 1$, top = $l_1 = s_3$, value = s_2 and rec = s_1.

Example 3.2. *Let* $\tau = [l_1 \mapsto$ Element; $l_2 \mapsto$ int; $s_1 \mapsto$ Element; $s_2 \mapsto$ Object; $s_3 \mapsto$ Element$]$ *be a type environment and consider Figure 3.2 representing the state* $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$*. Environment* ρ *maps variables* ℓ_1, ℓ_2 *and* s_1, s_2, s_3 *to value* $\ell_2, 2$ *and* ℓ_4, ℓ_3, ℓ_2 *respectively. Memory* μ *maps locations* ℓ_1 *and* ℓ_3 *to objects* o_1 *and* o_3 *of class* Integer *(obviously* Object*'s subclasses); it also maps location* ℓ_2 *and* ℓ_4 *to objects* o_2 *and* o_4 *of class* Element*. Objects are represented as boxes with a class tag and a local environment mapping fields to integers, locations or* `null`*. For instance, fields* value *and* next *of objects* o_2 *contain locations* ℓ_1 *and* ℓ_4 *respectively.*

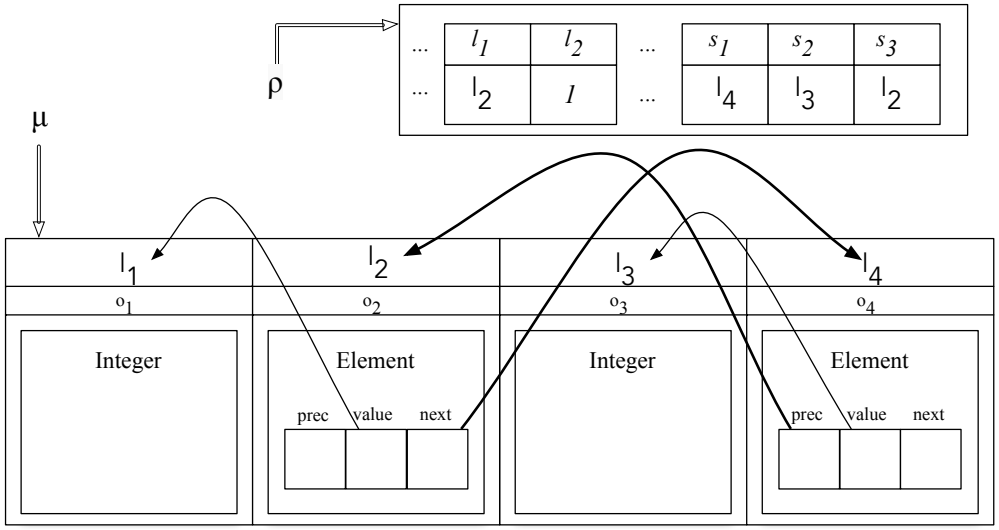


Figure 3.2: A JVM state $\sigma = \langle \rho, \mu \rangle$

3.3 Bytecode Semantics

The semantics of a bytecode instruction ins is a partial map $ins\colon \Sigma_\tau \to \Sigma_\tau$ from initial to final states. The number and type of local variables and stack elements at each program point are statically known and specified by τ. In the following we silently assume that bytecode instructions are run in a program point with type environment $\tau \subseteq \mathcal{T}$ such that $\mathsf{dom}(\tau) = L \cup S$, where L and S are local variables and stack elements, and let i and j be the cardinalities of these sets. Furthermore, we suppose that the semantics is undefined for input states of wrong sizes or types, as is required by Lindholm and Yellin [10]. We are going to discuss the semantics of bytecode instructions presented in Figure 3.1.

$$
\begin{array}{rl}
\mathsf{const}\ v = & \lambda\langle\langle l \parallel s\rangle, \mu\rangle\ .\ \langle\langle l \parallel v :: s\rangle, \mu\rangle \\
\mathsf{dup\ t} = & \lambda\langle\langle l \parallel t :: s\rangle, \mu\rangle\ .\ \langle\langle l \parallel t :: t :: s\rangle, \mu\rangle \\
\mathsf{load}\ k\ \mathsf{t} = & \lambda\langle\langle l \parallel s\rangle, \mu\rangle\ .\ \langle\langle l \parallel l[k] :: s\rangle, \mu\rangle \\
\mathsf{store}\ k\ \mathsf{t} = & \lambda\langle\langle l \parallel t :: s\rangle, \mu\rangle\ .\ \langle\langle l[k \mapsto t] \parallel s\rangle, \mu\rangle \\
\mathsf{ifne\ t} = & \lambda\langle\langle l \parallel t :: s\rangle, \mu\rangle\ .\ \begin{cases} \langle\langle l \parallel s\rangle, \mu\rangle & \text{if } t \in \{0, \mathtt{null}\} \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
\mathsf{ifeq\ t} = & \lambda\langle\langle l \parallel t :: s\rangle, \mu\rangle\ .\ \begin{cases} \langle\langle l \parallel s\rangle, \mu\rangle & \text{if } t \notin \{0, \mathtt{null}\} \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
\mathsf{new}\ \kappa = & \lambda\langle\langle l \parallel s\rangle, \mu\rangle\ .\ \begin{cases} \langle\langle l \parallel \ell :: s\rangle, \mu[\ell \mapsto o]\rangle & \text{if enough memory} \\ \underline{\langle\langle l \parallel \ell\rangle, \mu[\ell \mapsto \mathit{oome}]\rangle} & \text{otherwise} \end{cases} \\
\mathsf{getfield}\ \kappa.f{:}\mathsf{t} = & \lambda\langle\langle l \parallel r :: s\rangle, \mu\rangle\ .\ \begin{cases} \langle\langle l \parallel (\mu(r).\phi)(f) :: s\rangle, \mu\rangle & \text{if } r \neq \mathtt{null} \\ \underline{\langle\langle l \parallel \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases} \\
\mathsf{putfield}\ \kappa.f{:}\mathsf{t} = & \lambda\langle\langle l \parallel t :: r :: s\rangle, \mu\rangle\ .\ \begin{cases} \langle\langle l \parallel s\rangle, \mu[(\mu(r).\phi)(f) \mapsto t]\rangle & \text{if } r \neq \mathtt{null} \\ \underline{\langle\langle l \parallel \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases} \\
\mathsf{throw}\ \kappa = & \lambda\langle\langle l \parallel t :: s\rangle, \mu\rangle\ .\ \begin{cases} \underline{\langle\langle l \parallel t\rangle, \mu\rangle} & \text{if } t \neq \mathtt{null} \\ \underline{\langle\langle l \parallel \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases} \\
\mathsf{catch} = & \underline{\lambda\langle\langle l \parallel t\rangle, \mu\rangle}\ .\ \langle\langle t \parallel l\rangle, \mu\rangle \\
\mathsf{excp\_is}\ K = & \lambda\langle\langle l \parallel t\rangle, \mu\rangle\ .\ \begin{cases} \langle\langle l \parallel t\rangle, \mu\rangle & \text{if } t \in \mathbb{L} \text{ and } (\mu(t).\kappa) \in \mathbb{K} \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
\mathsf{return\ void} = & \lambda\langle\langle l \parallel s\rangle, \mu\rangle\ .\ \langle\langle l \parallel \epsilon\rangle, \mu\rangle \\
\mathsf{return\ t} = & \lambda\langle\langle l \parallel t :: s\rangle, \mu\rangle\ .\ \langle\langle l \parallel t\rangle, \mu\rangle, \quad \text{where t} \neq \mathsf{void}
\end{array}
$$

Table 3.1: Bytecode Semantics. Each instruction is modelled as a function mapping a pre-state to a post-state; $\ell \in \mathbb{L}$ is a fresh location, *oome* is a new instance of `OutOfMemoryException`, while *npe* a new instance of `NullPointerException`.

Basic Instructions. Bytecode const v pushes $v \in \mathbb{Z}$ on the stack. Since $\langle\langle l \parallel s\rangle, \mu\rangle$ (where s might be ϵ) is not underlined, const v is undefined on exceptional states i.e., it is run only when the JVMis in a normal state. This is the case for *all* bytecodes but catch, which starts the exceptional handlers from an exceptional state, and which is undefined on all normal state instead. Bytecode dup t duplicate the top of the stack, of type t. Bytecode load k t pushes on the stack the value of local variable with number k, l_k, which must exist and have type t. Conversely, bytecode store k t pops the top of the stack of type t and writes it in local variable l_k; if L contains less than $k + 1$ variables, the set of local variables grows. In our formalization, conditional bytecodes are used in complementary pair (such as ifne t and ifeq t), at the beginning of the two conditional branches. The semantics of a conditional bytecode is undefined when its condition is false. For instance, ifne t checks of the top of the stack, of type t, is not 0 when t = int or is not `null` otherwise; the *undefined* case means that the JVMdoes not continue the execution of the code if the condition is false.

Object-manipulating instructions. These bytecodes create or access objects in memory. Bytecode new κ pushes on the stack a reference to a new object o of class κ, whose fields are initialized to a default value: `null` for reference fields, and 0 for integer fields [10].

Bytecode getfield$\kappa.f$:t reads the fields $\kappa.f : t$ of a receiver object r popped from the stack, of type κ. Bytecode putfield $\kappa.f$:t writes the top of the stack, of type t, inside field $\kappa.f : t$ of the object pointed to by the underlying value r, ok type κ.

Exception-handling instructions. Bytecode throw κ throws the object pointed by the top of the stack, of type $\kappa <$ `Throwable`. Bytecode catch starts an exceptional handler. It takes an exceptional state and transform it into a normal one, subsequently used by the handler. After catch, bytecode excp_is K can be used to select an appropiate handler depending on the run-time class of the top of the stack: it filters those state whose top of the stack is an instance of $K \in \mathbb{K}$.

Method calls and return. When a caller transfers control to a callee $\kappa.m(\vec{\mathsf{t}})$: t, the JVMruns an operation $makescope\ \kappa.m(\vec{\mathsf{t}})$: t that copies the topmost stack elements, holding the actual arguments of the call, to local variables that correspond to the formal parameters of the callee, and then clears the stack. We only consider instance method, where this is a special argument held in local variable l_0 of the callee.

Definition 3.6. *Let $\kappa.m(\vec{\mathsf{t}})$: t be a method and π the number of stack elements holding its actual parameters, including the implicit parameter* this*. We define a function* ($makescope\ \kappa.m(\vec{\mathsf{t}})$: t): $\Sigma \to \Sigma$ *as*

$$\lambda\langle\langle l \parallel v_{\pi-1} :: \ldots :: v_1 :: \mathsf{rec} :: \mathsf{s}\rangle, \mu\rangle.\langle\langle[\mathsf{rec}, v_1, \ldots, v_{\pi-1}] \parallel \epsilon\rangle, \mu\rangle$$

provided rec $\neq$ `null` *and the look-up of $m(\vec{\mathsf{t}})$*:t *from the class μ*(rec).κ *leads to $\kappa.m(\vec{\mathsf{t}})$*:t*. We let it be undefined otherwise.*

More precisely, the $i - th$ local variable of the callee is a copy of the element located at $(\pi - 1) - i$ positions down the top of the stack of the caller. Bytecode return t terminates a method and clears its operand stack, leaving only the returned value when t $\neq$ void.

Example 3.3. *In Figure 3.1, we give the blocks of Java bytecode of the second constructor of the class* Element *(lines 8-12) in Code 1.*

3.4 Operational Semantics

We define the operational semantics of our language. It uses a stack of activation records to model method and constructor calls.

Definition 3.7 (Configuration)**.** *A* configuration *is a pair $\langle b \parallel \sigma\rangle$ of a block b and a state σ representing the fact that the* JVM *is about to execute b in state σ. An* activation stack *is a stack $c_1 :: c_2 :: \ldots :: c_n$ of configurations, where c_1 is the* active *configuration.*

The *operational semantics* of a Java bytecode program is a relation between activation stacks. It models the transformation of the activation stack induced by the execution of each single bytecode.

Definition 3.8 (Operational Semantics)**.** *The small step operational semantics of a Java Bytecode program P is a relation $a' \Rightarrow_P a''$ (P is usually omitted) providing the immediate successor activation stack a'' of an activation stack a'. It is defined in the rules in Table 3.2.*

- Rule **(1)** runs the first instruction ins of a block, different from call, by using its semantics *ins*.

Table 3.2: The Transition Rules of our Semantics

$$\frac{\textsf{ins is not a call, } \mathit{ins}(\sigma) \text{ is defined}}{\langle \boxed{\begin{array}{l}\mathit{ins}\\ \mathit{rest}\end{array}} \begin{array}{l}\rightarrow\\ \rightarrow\end{array} \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \sigma\rangle :: a \Rightarrow \langle \boxed{\mathit{rest}} \begin{array}{l}\rightarrow\\ \rightarrow\end{array} \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \mathit{ins}(\sigma)\rangle :: a} \tag{1}$$

$$\frac{\begin{array}{c}\pi \text{ is the number of parameters of the target method, including this}\\ \sigma = \langle\langle \ell \parallel v_{\pi-1} :: \ldots :: v_1 \ldots \textsf{rec} \ldots s\rangle, \mu\rangle, \textsf{rec} \neq \texttt{null}\\ 1 \leq i \leq n \text{ is such that } \sigma' = (\textsf{makescope } \kappa_i.m)(\sigma) \text{ is defined}\\ f = \mathit{first}(\kappa_i.m) \text{ is the block where the implementation starts}\end{array}}{\langle \boxed{\begin{array}{l}\textsf{call } \kappa_1.m \ldots \kappa_n.m\\ \mathit{rest}\end{array}} \begin{array}{l}\rightarrow\\ \rightarrow\end{array} \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \sigma\rangle :: a \Rightarrow \langle f \parallel \sigma'\rangle :: \langle \boxed{\mathit{rest}} \begin{array}{l}\rightarrow\\ \rightarrow\end{array} \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \langle\langle l \parallel s\rangle, \mu\rangle\rangle :: a} \tag{2}$$

$$\frac{\begin{array}{c}\pi \text{ is the number of parameters of the target method, including this}\\ \sigma = \langle\langle \ell \parallel v_{\pi-1} :: \ldots :: v_1 \ldots \textsf{rec} \ldots s\rangle, \mu\rangle\\ \ell \in \mathbb{L} \text{ is fresh and } \mathit{npe} \text{ is a new instance of } \texttt{NullPointerException}\end{array}}{\langle \boxed{\begin{array}{l}\textsf{call } \kappa_1.m \ldots \kappa_n.m\\ \mathit{rest}\end{array}} \begin{array}{l}\rightarrow\\ \rightarrow\end{array} \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \sigma\rangle :: a \Rightarrow \langle \boxed{\mathit{rest}} \begin{array}{l}\rightarrow\\ \rightarrow\end{array} \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \langle\langle l \parallel \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle\rangle :: a} \tag{3}$$

$$\frac{}{\langle \boxed{} \parallel \langle\langle l \parallel \mathit{top}\rangle, \mu\rangle\rangle :: \langle b \parallel \langle\langle l' \parallel s'\rangle, \mu'\rangle\rangle :: a \Rightarrow \langle b \parallel \langle\langle l' \parallel \mathit{top} :: s'\rangle, \mu\rangle\rangle :: a} \tag{4}$$

$$\frac{}{\langle \boxed{} \parallel \underline{\langle\langle l \parallel e\rangle, \mu\rangle\rangle} :: \langle b \parallel \langle\langle l' \parallel s'\rangle, \mu'\rangle\rangle :: a \Rightarrow \langle b \parallel \underline{\langle\langle l' \parallel e\rangle, \mu\rangle\rangle} :: a} \tag{5}$$

$$\frac{1 \leq i \leq m}{\langle \boxed{} \begin{array}{l}\rightarrow\\ \rightarrow\end{array} \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \sigma\rangle :: a \Rightarrow \langle b_i \parallel \sigma\rangle :: a} \tag{6}$$

- Rule **(2)** calls a method on a non-`null` receiver: the call instructions are decorated with an over-approximation of the set of their possible run-time target methods. This approximation can be computed by class analysis [16]. The dynamic semantics of call looks up for the exact target implementation $\kappa_i.m(\vec{\mathsf{t}}):\mathsf{t}$ that is executed, by using the look-up rules of the language, builds its initial state σ' by using $makescope$, and creates a new current configuration containing the first block of the target implementation and σ'. It pops the actual arguments from the previous configuration and the call from the instructions to be executed at return time. A method call may lead to many implementations, depending on the run-time class of the receiver, but, since in Java bytecode only one thread of execution continues, we can assume that the look-up is deterministic.

- On the other hand, if a call occurs on a `null` receiver, no actual call happens in this case and Rule **(3)** creates a new stack contains only a reference to a `NullPointerException`.

- After the execution of the method, if the callee ends in a normal state, control returns to the caller by Rule **(4)**: it rehabilitates the caller configuration but keeps the memory at the end of the execution of the callee and pushes the return value on the stack of the caller.

- Contrariwise, if the callee ends in an exceptional state, Rule **(5)** propagates the exception back to the caller.

- Rule **(6)** applies when all instructions inside a block have been executed; it runs one of its immediate successors, if any. In our formalization this rule is always deterministic: if a block has two or more immediate successors then they start with mutually exclusive conditional instructions and only one thread of control is actually followed.

We note that, when we use the notation $\Rightarrow$, we often specify which rule is used: for example we write $\overset{\text{Rule (1)}}{\Rightarrow}$ or simply $\overset{(1)}{\Rightarrow}$ for a derivation step through rule (1).

Chapter 4

Unreachability and Non-Cyclicity

In this chapter we formalize the notion of field-sensitive *unreachability* and *non-cyclicity* of program variables. In order to do that we need to give preliminary definitions and also some lemmas so that the binding between these definitions can be understood more clearly and, furthermore, they will be useful in the next sections.

4.1 Preliminary Definitions and Results

First of all, we introduce some preliminary definitions allowing us to formalize the unreachability and non-cyclicity properties that we want to state.

Definition 4.1 (Locations reachable from a location)**.** *Let $\tau \in \mathcal{T}$. The set of locations reachable from a location $\ell \in \mathbb{L}$ in a memory μ is*

$$\mathsf{L}_\mu(\ell) = \bigcup_{i \geq 0} \mathsf{L}^i_\mu(\ell)$$

where $\mathsf{L}^i_\mu(\ell)$ are the locations reachable from ℓ in at most i steps, defined as:

$$\mathsf{L}^i_\mu(\ell) = \begin{cases} \ell & \text{if } i{=}0 \\ \mathsf{L}^{i-1}_\mu(\ell) \cup \bigcup\limits_{\ell \in \mathsf{L}^{i-1}_\mu(\ell)} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) & \text{if } i{>}0 \end{cases}$$

Definition 4.2 (Locations reachable from a variable)**.** *Let $\tau \in \mathcal{T}$. The set of locations reachable from a variable $a \in \mathsf{dom}(\tau)$ in a state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ is*

$$\mathsf{L}_\sigma(a) = \begin{cases} \mathsf{L}_\mu(\rho(a)) & \text{if } \rho(a) \in \mathbb{L} \\ \emptyset & \text{otherwise} \end{cases}$$

Definition 4.3 (Fields in memory)**.** *The set of fields in a memory μ is*

$$\mathcal{F}_\mu = \bigcup_{\ell \in \mathsf{dom}(\mu)} \mathbb{F}(\mu(\ell).\kappa)$$

We note that $\mathcal{F}_\mu \subseteq \mathcal{F}$.

Definition 4.4 (Path among locations)**.** *Let $\tau \in \mathcal{T}$ and $\ell_1, \ell_2 \in dom(\mu) \subseteq \mathbb{L}$ two locations in a memory μ. We define a path $\mathcal{P}$ from ℓ_1 to ℓ_2 in μ as a n-tuple $\langle \kappa_1.f_1 : \mathsf{t}_1, \ldots, \kappa_n.f_n : \mathsf{t}_n \rangle \subseteq \mathcal{F}_\mu$ such that:*

$$\exists \ell^1, \ldots, \ell^{n+1} \in \mathsf{dom}(\mu).\ \ell^1 = \ell_1, \ell^{n+1} = \ell_2 \wedge \forall i = 1, \ldots, n\ \ (\mu(\ell^i).\phi)(\kappa_i.f_i : \mathsf{t}_i) = \ell^{i+1}$$

We denote it as $\ell_1 \rightsquigarrow_\mu^{\mathcal{P}} \ell_2$.

Hence, a path is a tuple of program fields that starting from an object in a location ℓ_1 allows, by going through a set of objects linked by the ordered fields in the tuple, to reach another object in a location ℓ_2.

Definition 4.5 (Path among variables)**.** *Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ and two variables $a, b \in \mathsf{dom}(\tau)$. A path $\mathcal{P}$ in σ between a and b is a path $\mathcal{P}$ between $\rho(a)$ and $\rho(b)$.*
We denote it as $a \rightsquigarrow_\sigma^{\mathcal{P}} b$.

According to the tuple definition given in Section 2.1, we note that we can also consider $\mathcal{P}$ as a set of fields. In this way, we use the common set relations and operations also between tuples and sets: the former will not change the nature of the tuples as ordered list of elements, contrariwise the latter will create a new set. Set operations between tuples are not allowed.

The following lemmas seek to shed light about the relation between the Definition 4.5 and the notions of aliasing and reachability among program variables.

Lemma 4.1. *Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ and two variables $a, b \in \mathsf{dom}(\tau)$.*

$$\left(\exists \mathcal{P} \subseteq \mathcal{F}_\mu.\ a \rightsquigarrow_\sigma^{\mathcal{P}} b \wedge \mathcal{P} = \langle\rangle\right) \Longleftrightarrow \rho(a) = \rho(b)$$

Proof. $\Rightarrow$) If $\mathcal{P} = \langle\rangle$ we have $n = 0$ and hence, $\rho(b) \doteq \ell^{0+1} \doteq \ell^1 \doteq \rho(a)$.
$\Leftarrow$) If $\rho(a) = \rho(b)$ we can satisfy the definition 4.4 by setting $\ell^1 = \rho(a)$, $\ell^{n+1} = \rho(b)$ and $n = 0$. In this way $\forall i = 1, \ldots, n$ is empty and then the last subformula of the Definition 4.4 is trivially true. □

We note that the definition of 4.5 is related with the locations reachable from a variable, definition 4.2, by the following lemma.

Lemma 4.2. *Let $\tau \in \mathcal{T}$ and $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$. Then*

$$\left(\exists \mathcal{P} \in \mathcal{F}_\mu.\ a \rightsquigarrow_\sigma^{\mathcal{P}} b\right) \Longleftrightarrow \rho(b) \in \mathsf{L}_\sigma(a)$$

Proof. $\Rightarrow$) If $\mathcal{P} = \langle\rangle$, then $\rho(a) = \rho(b)$ by lemma 4.1 and, therefore, $\rho(b) \in \mathsf{L}_\sigma(a)$.
If $\mathcal{P} \neq \langle\rangle$, we have $\ell^1, \ldots, \ell^{n+1} \in \mathsf{dom}(\mu)$. $\ell^1 = \rho(a), \ell^{n+1} = \rho(b) \wedge \forall i = 1, \ldots, n (\mu(\ell^i).\phi)$ $(\kappa_i.f_i : \mathsf{t}_i) = \ell^{i+1}$. We prove, by induction on i, $\forall i \in [1, n+1]\ \ell^i \in \mathsf{L}_\mu^{i-1}(\rho(a)) \wedge (\mu(\ell^i).\phi)$ $(\kappa_i.f_i : \mathsf{t}_i) \in \mathsf{L}_\mu^i(\rho(a))$. In this way, $\rho(b) \doteq (\mu(\ell^n).\phi)(\kappa_n.f_n : \mathsf{t}_n) \in \mathsf{L}_\mu^{n+1}(\rho(a)) \in \mathsf{L}_\sigma(a)$.

i) `Base case:` $i = 1$; we prove $\rho(a) \doteq \ell^1 \in \mathsf{L}_\mu^0(\rho(a)) \wedge (\mu(\ell^1).\phi)(\kappa_1.f_1 : \mathsf{t}_1) \in \mathsf{L}_\mu^1(\rho(a))$.
Since $\mathsf{L}_\mu^0(\rho(a)) \doteq \{\rho(a)\}$, we have $\ell^1 \in \mathsf{L}_\mu^0(\rho(a))$ and, since $\mathsf{rng}(\mu(\ell^1).\phi) \cap \mathbb{L} \subseteq \mathsf{L}_\mu^1(\rho(a))$, we have $(\mu(\ell^1).\phi)(\kappa_1.f_1 : \mathsf{t}_1) \in \mathsf{L}_\mu^1(\rho(a))$.

ii) `Inductive step:` we assume that $\ell^i \in \mathsf{L}_\mu^{i-1}(\rho(a)) \wedge (\mu(\ell^i).\phi)(\kappa_i.f_i : \mathsf{t}_i) \in \mathsf{L}_\mu^i(\rho(a))$. We prove $\ell^{i+1} \in \mathsf{L}_\mu^i(\rho(a)) \wedge (\mu(\ell^{i+1}).\phi)(\kappa_{i+1}.f_{i+1} : \mathsf{t}_{i+1}) \in \mathsf{L}_\mu^{i+1}(\rho(a))$.
Let $\ell^{i+1} = (\mu(\ell^i).\phi)(\kappa_i.f_i : \mathsf{t}_i) \in \mathsf{L}_\mu^i(\rho(a))$ by inductive hypotesis, and then we have $(\mu(\ell^{i+1}).\phi)(\kappa_{i+1}.f_{i+1} : \mathsf{t}_{i+1}) \in \mathsf{rng}(\mu(\ell^{i+1}).\phi) \cap \mathbb{L} \subseteq \bigcup_{\ell \in \mathsf{L}_\mu^i(\rho(a))} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) \subseteq \mathsf{L}_\mu^{i+1}(\rho(a))$.

$\Leftarrow$) If $\rho(b) \in \mathsf{L}_\sigma(a)$, then $\exists n \in \mathbb{N}$ such that $\rho(b) \in \mathsf{L}^n_\mu(\rho(a)) \setminus \mathsf{L}^{n-1}_\mu(\rho(a))$.
Hence we can build a path $\mathcal{P} = \langle \kappa_1.f_1 : \mathsf{t}_1, \ldots, \kappa_n.f_n : \mathsf{t}_n \rangle$ from a to b such that $\forall i \in [1, n] \exists \ell^i \in \mathsf{L}^{i-1}_\mu(\rho(a))$. $\kappa_i.f_i : \mathsf{t}_i \in \mathsf{dom}(\mu(\ell^i).\phi) \wedge \ell^{i+1} = (\mu(\ell^i).\phi)(\kappa_i.f_i : \mathsf{t}_i) \in \mathsf{L}^i_\mu(\rho(a))$ and $\ell^{n+1} \in \mathsf{L}^n_\mu(\rho(a))$ is such that $\ell^{n+1} = \rho(b)$. □

We also need to introduce a notion of static reachability between types.

Definition 4.6 (Reachability between types)**.** *Let* $\mathsf{t} \in \mathbb{T}$. *The* set of types reachable from t *is*

$$\mathsf{T}(\mathsf{t}) = \bigcup_{i \geq 0} \mathsf{T}^i(\mathsf{t})$$

where $\mathsf{T}^i(\mathsf{t})$ *are the types reachable from* t *in at most* i *steps:*

$$\mathsf{T}^i(\mathsf{t}) = \begin{cases} \mathsf{compatible}(\mathsf{t}) & \text{if } i = 0 \\ \mathsf{T}^{i-1}(\mathsf{t}) \cup \bigcup\limits_{\substack{\kappa \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{K}, \\ \kappa.f\,:\,\mathsf{t} \in \mathbb{F}(\kappa)}} \mathsf{compatible}(\mathsf{t}') & \text{if } i \geq 0 \end{cases}$$

We say that $\mathsf{t}' \in \mathbb{T}$ *is reachable from* t *if* $\mathsf{t}' \in \mathbb{T}(\mathsf{t})$ *and we write it as* $\mathsf{t} \rightsquigarrow \mathsf{t}'$.

Let us show some important results regarding type reachability. All these results are introduced by Nikolić and Spoto [13], but we include here to better understand the reachability between types, since it could be crucial to improve the precision of the analysis that we are going to present in Chapter 6.

Namely, we show that if a class reaches another class, then the former also reaches all possible superclasses of the latter (Lemma 4.3) and that the set of reachable types of a class t is included in the set of reachable types of all t's superclasses (Lemmas 4.5 and 4.6).

Lemma 4.3. *If* $\mathsf{t} \rightsquigarrow \mathsf{t}'$ *then for every* t'' *such that* $\mathsf{t} \leq \mathsf{t}''$ *(i.e., for every supertype of* t'*),* $\mathsf{t}' \rightsquigarrow \mathsf{t}''$ *holds as well.*

Proof. We prove that, for every $i \leq 0$, if $\mathsf{t}' \in \mathsf{T}(\mathsf{t})$ then $\mathsf{t} \in \mathsf{T}^i(\mathsf{t})$ for every t'' such that $\mathsf{t}' \leq \mathsf{t}''$. This entails the result for $\mathsf{T}(\mathsf{t})$ and hence the thesis. Assume hence $i = 0$. By Definition 4.6 we have $\mathsf{t} \in \mathsf{compatible}(\mathsf{t})$ and by Lemma 3.2 we have $\mathsf{t}'' \in \mathsf{compatible}(\mathsf{t})$ i.e., $\mathsf{t}'' \in \mathsf{T}^0(\mathsf{t})$. Let now $i > 0$ and assume, by inductive hypothesis, that $\mathsf{t}'' \in \mathsf{T}^i(\mathsf{t})$. Since $\mathsf{t}' \in \mathsf{T}^\mathsf{t}(i)$, by Definition 4.6, we distinguish two cases:

- $\mathsf{t}' \in \mathsf{T}^{i-1}(\mathsf{t})$ and, by inductive hypothesis, also $\mathsf{t}'' \in \mathsf{T}^{i-1}(\mathsf{t})$ which entails $\mathsf{t}'' \in \mathsf{T}^i(\mathsf{t})$;
- $\mathsf{t}' \in \mathsf{compatible}(\mathsf{t}_1)$ with $\kappa'.f : \mathsf{t}' \in \mathbb{F}(\kappa)$. In this case, by Definition 4.6, $\mathsf{compatible}(\mathsf{t}_1) \in \mathsf{T}^i(\mathsf{t})$. Since $\mathsf{t}' \leq \mathsf{t}''$, by Lemma 3.2 we have $\mathsf{t}'' \in \mathsf{compatible}(\mathsf{t}_1)$ and hence $\mathsf{t}'' \in \mathsf{T}^i(\mathsf{t})$.

□

Lemma 4.4. *Let* $\mathsf{t} \in \mathbb{T}$ *and* $i \geq 0$. *The set* $\mathsf{T}^i(\mathsf{t})$ *is closed w.r.t.* $\leq$.

Proof. The set $\mathsf{compatible}(\mathsf{t}')$ is closed w.r.t. $\leq$ for every $\mathsf{t}' \in \mathbb{T}$. The thesis follow by induction on i and Definition 4.6. □

Lemma 4.5. *Let* $\mathsf{t}, \mathsf{t}' \in \mathbb{T}$ *be such that* $\mathsf{t} \leq \mathsf{t}'$. *Then,* $\mathsf{T}(\mathsf{t}) \subseteq \mathsf{T}(\mathsf{t}')$.

Proof. We prove that, for every $i \geq 0$, $\mathsf{T}^i(\mathsf{t}) \subseteq \mathsf{T}^{i-1}(\mathsf{t}')$, by induction over i. If $i = 0$ the thesis follows by Lemma 3.3. Assume hence that $\mathsf{T}^{i-1}(\mathsf{t}) \subseteq \mathsf{T}^{i-1}(\mathsf{t}')$, for $i \leq 0$. Then,

$$\mathsf{T}^i(\mathsf{t}) = \mathsf{T}^{i-1}(\mathsf{t}) \cup \bigcup_{\substack{\kappa \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{K}, \\ \kappa.f\,:\,\mathsf{t} \in \mathbb{F}(\kappa)}} \mathsf{compatible}(\mathsf{t}'') \subseteq \mathsf{T}^{i-1}(\mathsf{t}') \cup \bigcup_{\substack{\kappa \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{K}, \\ \kappa.f\,:\,\mathsf{t} \in \mathbb{F}(\kappa)}} \mathsf{compatible}(\mathsf{t}'') = \mathsf{T}^i(\mathsf{t}')$$

□

The following lemma shows a relationship between variable and type reachability, namely, it shows that if one variable is reachable from another variable, then the static type of the former is reachable from the static type of the latter.

Lemma 4.6. *Let $\tau \in \mathcal{T}$, $\sigma \in \Sigma_\tau$ and $a, b \in \mathsf{dom}(\tau)$. If $\rho(b) \subseteq \mathsf{L}_\sigma(a)$, then $\tau(a) \rightsquigarrow \tau(b)$.*

Proof. By letting $\sigma = \langle \rho, \mu \rangle$, from $\rho(b) = \mathsf{L}_\sigma(a)$ and Definition 4.2, we have $\rho(a), \rho(b) \in \mathbb{L}$. We prove that for every $i \leq 0$, the following property $P(i)$ holds: for every $\ell \in \mathsf{L}^i_\mu(\rho(a))$, there exists $0 \leq j \leq i$ such that $\mu(\ell).\kappa \in \mathsf{T}^j(\tau(a))$. This entails our thesis. Namely, since $\rho(b) \in \mathsf{L}_\sigma(\rho(a))$, there exists $0 \leq j \leq i$ such that $\rho(b) \in \mathsf{L}^i_\mu(\rho(a)) \subseteq \mathsf{L}_\sigma(\rho(a))$, and $P(i)$ ensures that there also exists $0 \leq j \leq i$ such that $\mu(\rho(b)).\kappa \in \mathsf{T}^i(\tau(a)) \subseteq \mathsf{T}(\tau(a))$, i.e., $\tau(a) \rightsquigarrow \tau(b)$. Since (Definition 3.4) $\mu(\rho b).\kappa \leq \tau(b)$, by Lemma 4.3 we conclude that $\tau(a) \rightsquigarrow \tau(b)$. Let us now prove that, for every $i \geq 0$, $P(i)$ holds.

i) `Base case:` $i = 0$. Since $\rho(b) \in \mathsf{L}_\sigma(a)$, we have $\rho(a) \in \mathbb{L}$ and therefore $\mathsf{L}^0_\sigma(a) = \{\rho(a)\}$. By Definition 3.4, $\mu(\rho(a)).\kappa \leq \tau(a)$ i.e., $\mu(\rho(a)).\kappa \in \mathsf{compatible}(\tau(a)) = \mathsf{T}^0(\tau(a))$. Since $j = 0 \leq 0 = i$, $P(0)$ holds.

ii) `Inductive step:` Suppose that for every $k < i$, $P(k)$ holds and consider a location $\ell \in \mathsf{L}^i_\mu(\rho a)$. Then, by Definition 4.1, we have two cases:

- $\ell \in \mathsf{L}^{i-1}_\mu(\rho(a))$. By inductive hypothesis ($P(i-1)$ holds) we know that there exists $0 \leq j \leq i - 1 < i$ such that $\mu(\ell).\kappa \in \mathsf{T}^j(\tau(a))$. Therefore, $P(i)$ holds.
- $\ell \in \mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}$ for some for some $\ell' \in \mathsf{L}^{i-1}_\mu(\rho(a))$. By inductive hypothesis we know that there exists $0 \leq j \leq i - 1$ such that $\mu(\ell').\kappa \in \mathsf{T}^j(\tau(a))$. Moreover, $\ell \in (\mu(\ell').\phi)(\kappa'.f : \mathsf{t}')$ for a field $\kappa'.f : \mathsf{t}' \in \mathbb{F}(\mu(\ell').\kappa)$ with $\mu(\ell).\kappa \leq \mathsf{t}'$ (Definition 3.4). By Definition 4.6, we have $\mathsf{t}' \in \mathsf{T}^{j+1}(\tau(a))$. By Lemma 4.4 we conclude that $\mu(\ell).\kappa \in \mathsf{T}^{j+1}(\tau(a))$ and since $0 \leq j + 1 \leq i$, $P(i)$ holds as well.

□

By Lemma 4.6, we obtain a necessary condition, $\tau(a) \rightsquigarrow \tau(b)$, for reachability and, therefore, a sufficient condition, $\tau(a) \not\rightsquigarrow \tau(b)$, for unreachability, both between program variables.

4.2 The Field-Sensitive Properties

Let us formalize the two properties that we want to state in each point of the program under analysis.

Definition 4.7 (Field-Sensitive Unreachability among variables)**.** *Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$, $F \subseteq \mathcal{F}$ a set of program fields and variables $a, b \in \mathsf{dom}(\tau)$. We say that "for each path $\mathcal{P}$ from a to b in σ, the fields in F are not part of that path" iff*

$$\forall \mathcal{P} \subseteq \mathcal{F}_\mu \left(a \rightsquigarrow^{\mathcal{P}}_\sigma b \implies \mathcal{P} \cap F = \emptyset \right)$$

We denote it as $a \not\rightsquigarrow^F_\sigma b$.

Definition 4.8 (Field-Sensitive Non-Cyclicity)**.** *Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$, $F \subseteq \mathcal{F}$ a set of program fields and a variable $a \in \mathsf{dom}(\tau)$. We say that "for each cycle reachable from a in σ, the fields in F are not part of the cycle" iff*

$$\forall \ell \in \mathsf{L}_\sigma(a), \forall \mathcal{P} \subseteq \mathcal{F}_\mu \left(\ell \rightsquigarrow_\mu^{\mathcal{P}} \ell \Rightarrow \mathcal{P} \cap F = \emptyset \right)$$

We denote it as $a \rightsquigarrow_\sigma^{\not\circlearrowleft F}$.

In particular, the expression $\ell \rightsquigarrow_\mu^{\mathcal{P}} \ell$ with $\mathcal{P} \neq \langle\rangle$ denotes a cyclic-path and hence the cyclicity of ℓ i.e., a path with length greater than zero (a path with at least a field of the object stored in this location) through which we can reach the location itself.

It is worth noting that, when we assert $v \not\rightsquigarrow_\sigma^F w$ or $v \rightsquigarrow_\sigma^{\not\circlearrowleft F}$ for some $v, w \in \mathsf{dom}(\tau)$, these properties intuitively holds also for every $F' \subseteq F$. It will be crucial in our approximation since, every time we will able to assert one of these properties, we will also have to take into account all the subsets. We prove it in the following Lemma.

Lemma 4.7. *Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$, $F \subseteq \mathcal{F}$ and variables $a, b \in \mathsf{dom}(\tau)$. Then, we have:*

- $a \not\rightsquigarrow_\sigma^F b \iff \forall F' \subseteq F, a \not\rightsquigarrow_\sigma^{F'} b$;
- $a \rightsquigarrow_\sigma^{\not\circlearrowleft F} \iff \forall F' \subseteq F, a \rightsquigarrow_\sigma^{\not\circlearrowleft F'}$.

Proof. $\Rightarrow$) Let us prove the first item. If $a \not\rightsquigarrow_\sigma^F b$ then we have either $b \notin \mathsf{L}_\sigma(a)$ or for each path $\mathcal{P}_i$ between a and b, $\mathcal{P}_i \cap F = \emptyset$. If $b \notin \mathsf{L}_\sigma(a)$, the antecedent is always false and hence a and b are such that they satisfy the Definition 4.7 with any set F'. Otherwise we have $\mathcal{P}_i \cap F = \emptyset$ and, therefore, for each $F' \subseteq F$ we also have $\mathcal{P}_i \cap F' = \emptyset$ which entails our thesis. The proof about non-cyclicity item is similar, with the only difference that a could not reach any cycle instead of another variable.
$\Leftarrow$) Since in both items we have $F' \subseteq F$ the token is true also when $F' = F$. This entails the two theses. □

Therefore, from the opposite point of view, for every $a \not\rightsquigarrow_\sigma^{F_1} b$ and for every $c \rightsquigarrow_\sigma^{\not\circlearrowleft F_2}$ we also have that exist the maximal sets $\overline{F_1}$ and $\overline{F_2}$ such that $a \not\rightsquigarrow_\sigma^{\overline{F_1}} b$ with $F_1 \subseteq \overline{F_1}$ and $c \rightsquigarrow_\sigma^{\not\circlearrowleft \overline{F_2}}$ with $F_2 \subseteq \overline{F_2}$.

4.3 Related Variables Properties

Regarding Definition 4.7, given a variables pair, it is relevant to understand the relationship between "field-sentitive unreachability" Definition and both the reachability and alias properties. The following Lemma seeks to shed light about it.

Lemma 4.8. *Let $\tau \in \mathcal{T}$ and $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$. Then*

$$a \not\rightsquigarrow_\sigma^{\mathcal{F}} b \wedge \rho(a) \neq \rho(b) \Longrightarrow \rho(b) \notin \mathsf{L}_\sigma(a)$$

Proof. From $a \not\rightsquigarrow_\sigma^{\mathcal{F}} b$, by Definition 4.7 either $\nexists \mathcal{P} \subseteq \mathcal{F}_\mu.\ a \rightsquigarrow_\sigma^{\mathcal{P}} b$ or for each path $\mathcal{P} \subseteq \mathcal{F}_\mu$ between a and b we have $\mathcal{P} \cap \mathcal{F} = \emptyset$.

If $\mathcal{P}$ is a path between a and b, to enforce $\mathcal{P} \cap \mathcal{F} = \emptyset$, we must have $\mathcal{P} = \langle\rangle$. But, by Lemma 4.1, $\left(\exists \mathcal{P} \subseteq \mathcal{F}.\ a \rightsquigarrow_\sigma^{\mathcal{P}} b \wedge \mathcal{P} = \langle\rangle\right)$ iff $\rho(a) = \rho(b)$ and this is in contradiction with the other hypothesis i.e., $\rho(a) \neq \rho(b)$. Hence we cannot have a path $\mathcal{P}$ such that $\mathcal{P} = \langle\rangle$.

In both case we have that $\mathcal{P}$ cannot be a path between a and b. Therefore $\nexists \mathcal{P} \subseteq \mathcal{F}_\mu.\ a \rightsquigarrow_{\mathcal{P}}^{\sigma} b$ and then, by Lemma 4.2, $\rho(b) \notin \mathsf{L}_\sigma(a)$. □

By building the negative compound of that formula, we obtain the following corollary.

Corollary 4.8.1. *Let $\tau \in \mathcal{T}$ and $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$. Then*

$$\rho(b) \in \mathsf{L}_\sigma(a) \Longrightarrow \rho(a) = \rho(b) \vee \forall F \subseteq \mathcal{F} \left(a \not\rightsquigarrow_\sigma^F b \Rightarrow F \subset \mathcal{F} \right)$$

Proof. We are going to show that this is the negative compound of formula in Lemma 4.8 i.e.,

$$\neg \left[\forall F \subseteq \mathcal{F} \left(a \not\rightsquigarrow_\sigma^F b \Rightarrow F \subset \mathcal{F} \right) \right] \wedge \rho(a) \neq \rho(b) \Longrightarrow \rho(b) \notin \mathsf{L}_\sigma(a)$$

Thus we have only to show that $\neg \left[\forall F \subseteq \mathcal{F} \left(a \not\rightsquigarrow_\sigma^F b \Rightarrow F \subset \mathcal{F} \right) \right] \equiv a \not\rightsquigarrow_\sigma^{\mathcal{F}} b$:

$$\begin{aligned} \neg \left[\forall F \subseteq \mathcal{F} \left(a \not\rightsquigarrow_\sigma^F b \Rightarrow F \subset \mathcal{F} \right) \right] &\equiv \exists F \subseteq \mathcal{F} \neg \left(a \not\rightsquigarrow_\sigma^F b \Rightarrow F \subset \mathcal{F} \right) \\ &\equiv \exists F \subseteq \mathcal{F} \left(a \not\rightsquigarrow_\sigma^F b \wedge F = \mathcal{F} \right) \equiv a \not\rightsquigarrow_\sigma^{\mathcal{F}} b \end{aligned}$$

□

Corollary 4.8.2. *Let $\tau \in \mathcal{T}$ and $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$. Then*

$$a \not\rightsquigarrow_\sigma^{\mathcal{F}} b \wedge \rho(b) \in \mathsf{L}_\sigma(a) \Longrightarrow \rho(a) = \rho(b)$$

Proof. We are going to show that it is the same formula of Lemma 4.8:

$$\begin{aligned} a \not\rightsquigarrow_\sigma^{\mathcal{F}} b \wedge \rho(b) \in \mathsf{L}_\sigma(a) \Rightarrow \rho(a) = \rho(b) &\equiv a \not\rightsquigarrow_\sigma^{\mathcal{F}} b \Rightarrow \rho(a) = \rho(b) \vee \rho(b) \notin \mathsf{L}_\sigma(a) \\ &\equiv a \not\rightsquigarrow_\sigma^{\mathcal{F}} b \wedge \rho(a) \neq \rho(b) \Rightarrow \rho(b) \notin \mathsf{L}_\sigma(a) \end{aligned}$$

□

Thanks to the above Lemma and its two Corollaries, given a pair of variables, we can state:

- a *necessary* condition for *reachability*: $\rho(a) = \rho(b) \vee \forall F \subseteq \mathcal{F}_\mu \left(a \not\rightsquigarrow_\sigma^F b \Rightarrow F \subset \mathcal{F} \right)$;
- a *sufficient* condition for *aliasing*: $a \not\rightsquigarrow_\sigma^{\mathcal{F}} b \wedge \rho(b) \in \mathsf{L}_\sigma(a)$.

We also note that the reachability property allows to define a condition for alias and vice-versa. Therefore, given an ordered pair of variables $\langle v, w \rangle$, by having the information about the maximal set $\overline{F}$ such that $v \not\rightsquigarrow_\sigma^{\overline{F}} w$ and one property among reachability and alias, we can define, for every $\tau \in \mathcal{T}$ and for every σ over τ, two sets, `mayReach` or `mustAlias` that contain respectively possible information about reachability and defnite information about aliasing:

$$\texttt{mayReach}(\tau, \sigma) = \left\{ \langle a, b \rangle \in \mathsf{dom}(\tau) \,\middle|\, \neg \left(a \not\rightsquigarrow_\sigma^{\mathcal{F}} b \right) \vee \rho(a) = \rho(b) \right\}$$

$$\texttt{mustAlias}(\tau, \sigma) = \left\{ (a, b) \in \mathsf{dom}(\tau) \,\middle|\, a \not\rightsquigarrow_\sigma^{\mathcal{F}} b \wedge \rho(b) = \mathsf{L}_\sigma(a) \right\}$$

We can even establish a relation between Definitions 4.7 and 4.8:

Lemma 4.9. *Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$, $F \subseteq \mathcal{F}$ and a variable $a \in \mathsf{dom}(\tau)$. Then*

$$\forall F \subseteq \mathcal{F} \left(a \not\rightsquigarrow_\sigma^F a \Longrightarrow \exists F' \subseteq \mathcal{F}.\ a \not\rightsquigarrow_\sigma^{\not\ni F'} \wedge F \subseteq F' \right)$$

Proof. If F is such that $a \not\rightsquigarrow_\sigma^F a$, then $\forall \mathcal{P} \subseteq \mathcal{F}_\mu.\ a \rightsquigarrow_\sigma^{\mathcal{P}} a$ we have $\mathcal{P} \cap F = \emptyset$ (by Definition 4.7) and $\rho(a) \rightsquigarrow_\mu^{\mathcal{P}} \rho(a)$. Note that there is at least a path $\mathcal{P}$ between i.e., the empty path. Furthermore, the paths which allow $\rho(a)$ to reach $\rho(a)$ itself are such that:

$$\overline{\mathcal{P}_1} = \left\{ \mathcal{P} \subseteq \mathcal{F}_\mu \,\middle|\, \rho(a) \rightsquigarrow_\mu^{\mathcal{P}} \rho(a) \right\} \subseteq \left\{ \mathcal{P} \subseteq \mathcal{F}_\mu \,\middle|\, \ell \in \mathsf{L}_\sigma(a) \wedge \ell \rightsquigarrow_\mu^{\mathcal{P}} \ell \right\} = \overline{\mathcal{P}_2}$$

Since $\exists F' \subseteq \mathcal{F}.\ \forall \mathcal{P} \in \overline{\mathcal{P}_2}, \mathcal{P} \cap F' = \emptyset$ by Definition 4.8, we also have $\forall \mathcal{P} \in \overline{\mathcal{P}_1}, \mathcal{P} \cap F' = \emptyset$. Therefore, since $\forall \mathcal{P} \in \overline{\mathcal{P}_1}, \mathcal{P} \cap F = \emptyset$ by Definition 4.7 and $F, F' \subseteq \mathcal{F}$, it must also be $F \subseteq F'$. □

Chapter 5

Abstract Interpretation

Among the approaches to accomplish program analysis [12], we use Abstract Interpretation [5] in order to approximate our concrete semantics w.r.t. the field-sensitive unreachability and non-cyclicity properties.

It is worth noting that, since any property which these analyses try to infer is undecidable[1], we can group them in two major categories:

- `over-approximation analyses`: an analysis over-approximates a property ψ when we can soundly calculate the abstract semantic in a such way that every abstract element represents those concrete states where that ψ may be valid;

- `under-approximation analyses`: an analysis under-approximates a property ψ when we can soundly calculate the abstract semantic in a such way that every abstract element represents those concrete states where that ψ is certainly valid.

We can also interpret ψ as a set Ψ i.e., the set of all the concrete items which satisfy the logic predicate that defines ψ. In this way, we note that the former provides a sufficient condition to assert $\neg\psi$ i.e., the concrete items which the correspondent abstract one is not inside Ψ, whereas the latter provides a sufficient condition to assert ψ i.e., the concrete items which the correspondent abstract one is in Ψ.

Although the former is the most common one, we are going to introduce an definite (i.e., under-approximation) analysis, since our goal is find a subset of program fields such that any path or any cycle does not surely go though them.

5.1 Concrete and Abstract Domains

Definition 5.1 (Concrete and Abstract Domain)**.** *Given a type environment $\tau \in \mathcal{T}$, we define the* concrete lattice *over τ as $\mathsf{C}_\tau = \langle \wp(\Sigma_\tau), \sqsubseteq_\mathsf{C}, \sqcup_\mathsf{C}, \sqcap_\mathsf{C} \rangle$ and the* abstract lattice *over τ as $\mathsf{A}_\tau = \langle \mathsf{UR}_\tau \cup \mathsf{NC}_\tau, \sqsubseteq_\mathsf{A}, \sqcup_\mathsf{A}, \sqcap_\mathsf{A} \rangle$ i.e., the union of two sets:*

- $\mathsf{UR}_\tau = \wp(\mathsf{dom}(\tau) \times \mathsf{dom}(\tau) \times \wp(\mathcal{F}))$ *i.e., the powerset of the product between the set of ordered pairs of variables and the powerset of the program fields. For every $v, w \in \mathsf{dom}(\tau)$ and $F \subseteq \mathcal{F}$, we write $v \not\rightsquigarrow^F w$ to denote the triple $\langle v, w, F \rangle$.*

[1] This is because all these properties are non-trivial i.e., neither empty nor all recursively enumerable languages, and therefore, by Rice's Theorem, all undecidable. [9].

- $\mathsf{NC}_\tau = \wp(\mathsf{dom}(\tau) \times \wp(\mathcal{F}))$ *i.e., the powerset of the product between the set of variables and the powerset of the program fields. For every* $v \in \mathsf{dom}(\tau)$ *and* $F \subseteq \mathcal{F}$*, we write* $v \rightsquigarrow\!\!\!\!\!\!\diagup^{F}$ *to denote the pair* $\langle v, F \rangle$.

Given two concrete elements $S_1, S_2 \in \mathsf{C}_\tau$ *the partial order* $\sqsubseteq_\mathsf{C}$ *is defined as* $S_1 \sqsubseteq_\mathsf{C} S_2 \equiv S_1 \subseteq S_2$*, the* join operator $\sqcup_\mathsf{C}$ *as* $S_1 \sqcup_\mathsf{C} S_2 \equiv S_1 \cup S_2$ *and, respectively, the* meet operator $\sqcap_\mathsf{C}$ *as* $S_1 \sqcap_\mathsf{C} S_2 \equiv S_1 \cap S_2$*. Instead of, given two abstract elements* $I_1, I_2 \in \mathsf{A}_\tau$*, the partial order* $\sqsubseteq_\mathsf{A}$ *is defined as* $I_1 \sqsubseteq_\mathsf{A} I_2 \equiv I_1 \supseteq I_2$*, the* join operator $\sqcup_\mathsf{A}$ *as* $I_1 \sqcup_\mathsf{A} I_2 \equiv I_1 \cap I_2$ *and the* meet operator *respectively* $\sqcap_\mathsf{A}$ *as* $I_1 \sqcap_\mathsf{A} I_2 \equiv I_1 \cup I_2$*. Hence* A_τ *and* C_τ *satisfy the* complete lattices *definition given in 2.4.*

An abstract element $I \in \mathsf{A}_\tau$ represents those concrete states in Σ_τ whose unreachability and non-cyclicity information w.r.t. a set of fields is *under-approximated* by the tokens in I. Thus, we induce a *definite* unreachability and non-cyclicity w.r.t. an *under-approximation* of a set of program fields.

Through the join and meet operator, we can also spot respectively the *least upper bound* (lub, also called supremum) and the *greatest lower bound* (*glb*, also called infimum) of these two posets:

	Concrete Domain C_τ	Abstract Domain A_τ
glb	$\emptyset$	$\mathsf{dom}(\tau)^2 \times \wp(\mathcal{F}) \cup \mathsf{dom}(\tau) \times \wp(\mathcal{F})$
lub	Σ_τ	$\emptyset$

It is worth noting that the *glb* of the concrete domain is the same of the lub of the abstract one and also the concrete lub is related with the abstract *glb*. That is because, since our analysis under-approximate the real behaviour of the program, the meet abstract operator is equals to the join concrete one and vice-versa.

5.2 Concretization Map

We introduce the function that maps every abstract element I into the correspondent concrete one S.

Definition 5.2 (Concretization map)**.** *Let* $\tau \in \mathcal{T}$ *and* $I \in \mathsf{A}_\tau$*, we define the* concretization map $\gamma_\tau : \mathsf{A}_\tau \to \mathsf{C}_\tau$ *as*

$$\gamma_\tau(I) = \left\{ \sigma \in \Sigma_\tau \;\middle|\; \begin{array}{l} \left(\forall a \not\rightsquigarrow^{F} b \in I, a \not\rightsquigarrow_{\sigma}^{F'} b \wedge F \subseteq F'\right) \wedge \\ \left(\forall c \rightsquigarrow\!\!\!\!\!\!\diagup^{F} \in I, c \rightsquigarrow_{\sigma}\!\!\!\!\!\!\diagup^{F'} \wedge F \subseteq F'\right) \end{array} \right\}$$

Hence, every time we statically assert an unreachability or non-cyclicity property above a general program, we can conclude that the program has that property, but some unreachability or non-cyclicity relations could not be captured by our analysis.

Furthermore it is worth noting that, by Lemma 4.7, also in I, every time we have a unreachability or non-cyclicity token w.r.t. a particular set of fields F, the same relations are also valid with any subset of F. For this reason, we define the following operator.

Definition 5.3 (Normalize Operator)**.** *Let* $\tau \in \mathcal{T}$ *and* $I \in \mathsf{A}_\tau$*. We define the operator* normalize : $\mathsf{A}_\tau \to \mathsf{A}_\tau$ *as*

$$\lambda I.I \cup \left\{ a \not\rightsquigarrow^{F'} b \,\middle|\, a \not\rightsquigarrow^{F} b \in I \wedge F' \subseteq F \right\} \cup \left\{ c \rightsquigarrow\!\!\!\!\!\!\diagup^{F'} \,\middle|\, c \rightsquigarrow\!\!\!\!\!\!\diagup^{F} \in I \wedge F' \subseteq F \right\}$$

In other words, given an element $I \in \mathsf{A}_\tau$, for each unreachability and non-cyclicity token w.r.t. a set F, the normalize operator adds the same relations with every $F' \subseteq F$.

5.3 Galois Connection

We have now to show that A_τ and C_τ are respectively an abstract and concrete lattice, in the sense of *Abstract Interpretation*. In order to assert it, we have to find two functions α_τ and γ_τ which define a *Galois Connection* $(\mathsf{C}_\tau, \alpha_\tau, \gamma_\tau, \mathsf{A}_\tau)$ between these posets [12], so that α_τ is completely additive, whereas γ_τ completely co-additive (also called multiplicative).

Let us prove that γ_τ previously defined is co-additive.

Lemma 5.1 (Co-additivity of γ_τ)**.** *Let $\tau \in \mathcal{T}$. The function γ_τ is co-additive i.e., it is such that $\gamma_\tau(\sqcap_{\mathsf{A}} I) = \sqcap_{\mathsf{C}} \gamma_\tau(I)$.*

Proof. Let $I_j \in \mathsf{A}_\tau$ for $j \geq 0$. Then we have:

$$
\begin{aligned}
\gamma_\tau\Big(\bigsqcap_{\mathsf{A}} I_j\Big) \doteq \gamma_\tau\left(\bigcup_{j\geq 0} I_j\right) &= \left\{\sigma \in \Sigma_\tau \;\middle|\; \begin{array}{l} \left(\forall a \not\rightsquigarrow^F b \in \bigcup\limits_{j\geq 0} I_j,\, a \not\rightsquigarrow_\sigma^{F'} b \wedge F \subseteq F'\right) \wedge \\ \left(\forall c \rightsquigarrow\!\!\!\!\!\!\diagup^F \in \bigcup\limits_{j\geq 0} I_j,\, c \rightsquigarrow\!\!\!\!\!\!\diagup_\sigma^{F'} \wedge F \subseteq F'\right) \end{array}\right\} \\
&= \left\{\sigma \in \Sigma_\tau \;\middle|\; \begin{array}{l} \left(\forall j \geq 0, \forall a \not\rightsquigarrow^F b \in I_j,\, a \not\rightsquigarrow_\sigma^{F'} b \wedge F \subseteq F'\right) \wedge \\ \left(\forall j \geq 0, \forall c \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I_j,\, c \rightsquigarrow\!\!\!\!\!\!\diagup_\sigma^{F'} \wedge F \subseteq F'\right) \end{array}\right\} \\
&= \left\{\sigma \in \Sigma_\tau \;\middle|\; \forall j \geq 0 \left[\begin{array}{l} \left(\forall a \not\rightsquigarrow^F b \in I_j,\, a \not\rightsquigarrow_\sigma^{F'} b \wedge F \subseteq F'\right) \wedge \\ \left(\forall c \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I_j,\, c \rightsquigarrow\!\!\!\!\!\!\diagup_\sigma^{F'} \wedge F \subseteq F'\right) \end{array}\right]\right\} \\
&= \bigcap_{j\geq 0} \left\{\sigma \in \Sigma_\tau \;\middle|\; \begin{array}{l} \left(\forall a \not\rightsquigarrow^F b \in I_j,\, a \not\rightsquigarrow_\sigma^{F'} b \wedge F \subseteq F'\right) \wedge \\ \left(\forall c \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I_j,\, c \rightsquigarrow\!\!\!\!\!\!\diagup_\sigma^{F'} \wedge F \subseteq F'\right) \end{array}\right\} \\
&= \bigcap_{j\geq 0} \gamma_\tau\left(I_j\right) \doteq \bigsqcap_{\mathsf{C}} \gamma_\tau\left(I_j\right)
\end{aligned}
$$

□

Hence, since γ_τ is co-additive, it preserves the greatest lower bound and furthermore we have a *Galois Connection*.

We recall from the Section 2.6 that one of the properties of the Galois Connections is that α_τ can be completely determined by γ_τ as $\alpha_\tau(S) = \sqcap_{\mathsf{A}} \{I \in \mathsf{A}_\tau \,|\, S \sqsubseteq_{\mathsf{C}} \gamma_\tau(I)\}$. Thus, for completeness, we also add the definition of the abstraction map and the proof of its additivity.

Definition 5.4 (Abstraction map)**.** *Let $\tau \in \mathcal{T}$ and $S \subseteq \Sigma_\tau$, we define the* abstraction map *$\alpha_\tau : \mathsf{C}_\tau \to \mathsf{A}_\tau$ as*

$$
\begin{aligned}
\alpha_\tau(S) &\doteq \bigsqcap_{\mathsf{A}} \{I \in \mathsf{A}_\tau \,|\, S \sqsubseteq_{\mathsf{C}} \gamma_\tau(I)\} \\
&\doteq \bigcup_{j\geq 0} \left\{I_j \in \mathsf{A}_\tau \,\middle|\, S \subseteq \gamma_\tau(I_j)\right\} \\
&= \left\{ \begin{array}{l} a \not\rightsquigarrow^{F_1} b \in \mathsf{dom}(\tau) \times \mathsf{dom}(\tau) \times \wp(\mathcal{F}), \\ c \rightsquigarrow\!\!\!\!\!\!\diagup^{F_2} \in \mathsf{dom}(\tau) \times \wp(\mathcal{F}) \end{array} \;\middle|\; \forall \sigma \in S \left(\begin{array}{l} a \not\rightsquigarrow_\sigma^{F_1'} b \wedge F_1 \subseteq F_1', \\ c \rightsquigarrow\!\!\!\!\!\!\diagup_\sigma^{F_2'} \wedge F_2 \subseteq F_2' \end{array}\right)\right\}
\end{aligned}
$$

Lemma 5.2 (Additivity of α_τ)**.** *Let $\tau \in \mathcal{T}$. The function α_τ is additive i.e., it is such that $\alpha_\tau(\bigsqcup_{\mathsf{C}} S) = \bigsqcup_{\mathsf{A}} \alpha_\tau(S)$.*

Proof. Let $S_j \in \mathsf{C}_\tau$ for $j \geq 0$. Then we have:

$$
\begin{aligned}
\alpha_\tau\left(\bigsqcup\nolimits_{\mathsf{C}} S_j\right) &\doteq \alpha_\tau\left(\bigcup_{j\geq 0} S_j\right) \\
&= \left\{ \begin{array}{l} a \not\rightsquigarrow^{F_1} b \in \mathsf{dom}(\tau) \times \mathsf{dom}(\tau) \times \wp(\mathcal{F}), \\ c \rightsquigarrow \not\ni^{F_2} \in \mathsf{dom}(\tau) \times \wp(\mathcal{F}) \end{array} \middle| \forall \sigma \in \bigcup_{j\geq 0} S_j \left(\begin{array}{l} a \not\rightsquigarrow_\sigma^{F_1'} b \wedge F_1 \subseteq F_1', \\ c \rightsquigarrow_\sigma \not\ni^{F_2'} \wedge F_2 \subseteq F_2' \end{array} \right) \right\} \\
&= \left\{ \begin{array}{l} a \not\rightsquigarrow^{F_1} b \in \mathsf{dom}(\tau) \times \mathsf{dom}(\tau) \times \wp(\mathcal{F}), \\ c \rightsquigarrow \not\ni^{F_2} \in \mathsf{dom}(\tau) \times \wp(\mathcal{F}) \end{array} \middle| \forall j \geq 0, \forall \sigma \in S_j \left(\begin{array}{l} a \not\rightsquigarrow_\sigma^{F_1'} b \wedge F_1 \subseteq F_1', \\ c \rightsquigarrow_\sigma \not\ni^{F_2'} \wedge F_2 \subseteq F_2' \end{array} \right) \right\} \\
&= \bigcap_{j\geq 0} \left\{ \begin{array}{l} a \not\rightsquigarrow^{F_1} b \in \mathsf{dom}(\tau) \times \mathsf{dom}(\tau) \times \wp(\mathcal{F}), \\ c \rightsquigarrow \not\ni^{F_2} \in \mathsf{dom}(\tau) \times \wp(\mathcal{F}) \end{array} \middle| \forall \sigma \in S_j \left(\begin{array}{l} a \not\rightsquigarrow_\sigma^{F_1'} b \wedge F_1 \subseteq F_1', \\ c \rightsquigarrow_\sigma \not\ni^{F_2'} \wedge F_2 \subseteq F_2' \end{array} \right) \right\} \\
&= \bigcap_{j\geq 0} \alpha_\tau\left(S_j\right) \doteq \bigsqcup\nolimits_{\mathsf{A}} \alpha_\tau(S)
\end{aligned}
$$

□

Hence, since α_τ is additive, it preserves the least upper bound.

Since our analysis under-approximates the real behavior of the program, it is worth noting that we have also obtained an under-approximating Galois Connection: let S be a set of states we have $\gamma(\alpha(S)) \subseteq S$, with α the corresponding abstraction map of that Connection.

Chapter 6

The Field-Sentitive Analysis

Before introducing the field-sensitive analysis, we discuss how to exploit other static analyses in order to achieve a better overall precision of the assertions that we are able to state.

6.1 Exploiting other Static Analyses

In the following we assume that these analyses are processed asynchronously i.e., we also have the related approximated information at each program point.

Since the unreachability and non cyclicity analysis is parametrized by other analyses both over and under approximated, we explain how different analyses of the same property can be combined to achieve the best approximation among all.

Let us suppose we want to state a property ψ and we have n not related analyses $\Psi_1, \Psi_2, \ldots, \Psi_n$[1] to infer it. If they are all over-approximation analyses, in order to combine them to achieve a better approximation, we have to create the intersection set $\underline{\Psi}$ i.e.,

$$\underline{\Psi} = \Psi_1 \cap \Psi_2 \cap \cdots \cap \Psi_n$$

We note that, since $\cap$ corresponds to the meet operator $\sqcap_{\mathsf{C}}$, $\underline{\Psi}$ approaches to the *glb* of the concrete lattice i.e., , the empty set (see Section 5.1).

Contrarywise if we have n under-approximation analyses, to combine them we have to create the union set $\overline{\Psi}$ i.e.,

$$\overline{\Psi} = \Psi_1 \cup \Psi_2 \cup \cdots \cup \Psi_n$$

We note that, since $\cup$ corresponds to the meet operator $\sqcup_{\mathsf{C}}$, $\overline{\Psi}$ approaches to the lub of the concrete lattice i.e., the set Σ_τ of all the states in that type environment (see Section 5.1).

The three main analyses that we need to exploit are: *Possible Reachability*, an over--approximation of the concrete reachability information, *Possible Sharing*, an over-approximation of the concrete sharing information, and *Definite Aliasing*, an under-approximation of the concrete aliasing information, all respect to pairs of variables and/or fields of the program. Let us explain them more in depth.

6.1.1 Definite Aliasing

The definite alias analysis tries to infer, in each program point, which variables or variable's fields *must* be aliased to another variable. In order to state this property between variables pairs,

[1] These analyses must be all under or over approximation of the concrete behaviours of the program.

we could try to exploit our analysis: the Corollary 4.8.2 provides us a sufficient condition for aliasing between variables and hence we can try to define, for each program point, the set

$$\mathsf{uAV}_\tau = \left\{(a, b) \in \mathsf{dom}(\tau) \,\middle|\, a \not\rightsquigarrow^{\mathcal{F}} b \in I \wedge a \text{ definitely reaches } b\right\}$$

The main issue of this approach is that we need a definite reachability analysis between variables. Fortunately we can get around it by using a definite aliasing analysis already developed by Nikolic and Spoto [15]. This analysis is between variables and expressions where expressions include both variables and variables' fields. Based on this analysis we can suppose to have, for each program point, a set $\mathcal{DA}_\tau$ which contains the aliasing pairs both of variables and of variables and fields. Furthermore our analysis works correctly even when this approximation is not available: by setting $\mathcal{DA}_\tau = \emptyset$, the unreachability information (and hence non-cyclicity) we determine would be less precise, but still sound.

6.1.2 Possible Reachability

The possible reachability analysis tries to infer if *may* exist, in memory μ, a path between two locations bound to: (1) two variables, (2) a variable and a field or (3) two fields of the program. Unfortunately to state this property we cannot exploit its information, although the Corollary 4.8.1 gives us a necessary condition for reachability: indeed, by having I and the definite alias set $\mathcal{DA}_\tau$ explained in the subsection 6.1.1, we can define, for each program point, the set

$$\mathsf{uRV}_\tau = \left\{\langle a, b\rangle \in \mathsf{dom}(\tau) \,\middle|\, a \not\rightsquigarrow^{\mathcal{F}} b \notin I \vee (a, b) \in \mathcal{DA}_\tau\right\}$$

but, since I contains under-appoximated tokens, this set is an *under-approximation* of concrete reachability among variables, whereas we need an over-approximated one.

Therefore we have to parametrize our analysis with another that is able to state the possible (may) reachability property. The most related work about it is the *Constraint-Based Reachability Analysis* for Java bytecode [13, 14], because it is typeset with the same framework and even implemented in the same tool, Julia [23, 24]. In this paper it has been built an *abstract constraint graph* that for each program point (i.e., bytecode instruction) provides a set R_τ with the ordered pairs of variables such that the first one may reach the second one.

Unfortunately, this analysis does not consider the possible reachability between two variables' fields or between a field and a variable (or vice-versa). Nevertheless we note that if a variable v reaches a variables w, then v also reaches all the w fields, whereas we cannot infer which v field(s) reaches w. Therefore we suppose to be able to benefit of a state of the art analysis from a variable field to a variable (and hence all its fields) such that, for each program point, there is a set $\mathcal{FR}_\tau$ which contains all the possible reachability pairs ⟨field, variable⟩.

Definition 6.1. *Let $\tau \in \mathcal{T}$ Then we define the state of the art set $\mathcal{MR}_\tau$ (May Reach) w.r.t. the possible reachability among variables or from a field and a variable as*

$$\mathcal{MR}_\tau = \mathcal{VR}_\tau \cup \mathcal{FR}_\tau$$

where $\mathcal{VR}_\tau$ (Variables' Reachability) is the state of the art set w.r.t. the reachability among variables i.e.,

$$\mathcal{VR}_\tau = VR_1 \cap VR_2 \cap \cdots \cap VR_n$$

whereas $\mathcal{FR}_\tau$ (Fields' Reachability) is the state of the art set w.r.t. the reachability from a variable's field to a variable i.e.,

$$\mathcal{FR}_\tau = FR_1 \cap FR_2 \cap \cdots \cap FR_n$$

We note that our analysis works correctly even when these two approximations are not available: thanks to the static information about the type of variables and fields, by Lemma 4.6, we can always assert a static possible reachability property i.e., we can set VR_1 and FR_1 respectively to:

$$VR_1 = \{\langle a, b\rangle \in \mathsf{dom}(\tau) \,|\, \tau(a) \rightsquigarrow \tau(b)\}$$
$$FR_1 = \{\langle a.(\kappa.f : \mathsf{t}), b\rangle \in \mathbb{F}(\kappa) \times \mathsf{dom}(\tau) \,|\, \mathsf{t} \rightsquigarrow \tau(b)\}$$

Obviously these sets are the least precise ones, since they only take in account of the static information between types, nevertheless they are sound.

6.1.3 Possible Sharing

The possible sharing analysis determines those pairs of program variables bound at run-time to overlapping data structures. In other words two variables share if they might reach the same location at run-time.

It is worth noting that this analysis is an over-approximation of the possible reachability analysis i.e., every time the reachability analysis state that a variable may reach another one, then these two variables can share. Unfortunately also in this case we are not able to exploit our analysis to state this property. Instead, we will exploit the analysis presented in [21] and implemented in Julia tool [24], which, at every program point, is able to provide a set $\mathcal{MS}_\tau$ with an over-approximation off all the pairs of variables that *share* in that point. Nevertheless if this analysis is not available, you could exploit the reachability one in this way:

$$\mathcal{MS}_\tau = \{(a, b) \in \mathsf{dom}(\tau) \,|\langle a, b\rangle \in \mathcal{MR}_\tau \vee \langle b, a\rangle \in \mathcal{MR}_\tau\}$$

6.2 The Abstract Constraint Graph

Our analysis is constraint-based [12], in the sense that it builds an abstract constraint graph from the program under analysis, by creating a node of the graph for each bytecode instruction *ins* of the program. This node contains an element from A_τ, where τ is the static type information at the beginning of *ins*. Arcs of this graph propagate abstract domain elements, reflecting, in abstract terms, the effects of the concrete semantics over the reachability information. In other words, an arc from the node for bytecode instruction ins_1 to the node for bytecode instruction ins_2 propagates the information at ins_1 into that at ins_2. The exact meaning of propagates depends here on ins_1, since each bytecode instruction has different effects on our unreachability and non-cyclicity properties.

Definition 6.2 (ACG)**.** *Let P be the program under analysis (i.e., a control flow graph of basic blocks for each method or constructor). The* abstract constraint graph *(*ACG*) of P is a directed graph $\langle V, E\rangle$ (nodes, arcs) where:*

- *V contains a node $\boxed{\mathsf{ins}}$, for every bytecode instruction* ins *of P;*
- *V contains nodes $\boxed{\mathsf{exit@}m}$ and $\boxed{\mathsf{exception@}m}$ for each method or constructor m in P, and these nodes correspond to the normal and exceptional end of m;*
- *E contains directed arcs from a source to a sink;*
- *for every arc in E, there is a* propagation rule $\Pi^{\#i}$*, i.e., a function over A_τ, from the information at its source to the information at its sink.*

The arcs in E of the ACG are built from P as follows. We assume that τ and τ' are the static type information at and immediately after the execution of a bytecode ins*, respectively. Furthermore, we assume that τ contains j stack elements and i local variables.*
We define the propagation rules as an order set of functions $\{\Pi^{\#1}, \ldots, \Pi^{\#n}\}$, where n is the number of the java bytecode instructions which we consider and, therefore, $\Pi^{\#i}$ represents the function relative to a specific instruction. In the following we discuss different types of arcs which define each propagation rule.

Sequential arcs. *If* ins *is a bytecode in P, distinct from* call*, immediately followed by a bytecode* ins′*, distinct from* catch*, then a simple arc is built from* $\boxed{\text{ins}}$ *to* $\boxed{\text{ins}'}$*, with one of the propagation rules in Table 6.1.*

Table 6.1: Propagation Rules of Sequential Arcs

Π	*Instruction*	*Propagation Rule*
#1	dup t	$\lambda I.I \cup I[s_{j-1} \mapsto s_j] \cup \left\{ s_{j-1} \not\rightsquigarrow^F s_j, s_j \not\rightsquigarrow^F s_{j-1} \,\middle|\, s_{j-1} \not\rightsquigarrow^F s_{j-1} \in I \right\}$
#2	new κ, const v	$\lambda I.I \cup \left\{ s_j \not\rightsquigarrow^{\mathcal{F}} s_j,\ s_j \rightsquigarrow \not\!\!\!\!\!\!\bigcirc^{\mathcal{F}} \right\} \cup \left\{ a \not\rightsquigarrow^{\mathcal{F}} s_j, s_j \not\rightsquigarrow^{\mathcal{F}} a \,|\, a \in dom(\tau) \right\}$
#3	load k t	$\lambda I.I \cup I[l_k \mapsto s_j] \cup \left\{ l_k \not\rightsquigarrow^F s_j, s_j \not\rightsquigarrow^F l_k \,\middle|\, l_k \not\rightsquigarrow^F l_k \in I \right\}$
#4	store k t	$\lambda I. \left\{ \left(a \not\rightsquigarrow^F b \right) \left[s_{j-1} \mapsto l_k \right] \middle| a \not\rightsquigarrow^F b \in I \wedge a, b \neq l_k \right\}$ $\cup \left\{ \left(c \rightsquigarrow \not\!\!\!\!\!\!\bigcirc^F \right) \left[s_{j-1} \mapsto l_k \right] \middle| c \rightsquigarrow \not\!\!\!\!\!\!\bigcirc^F \in I \wedge a \neq l_k \right\}$
#5	getfield $\kappa.f$:t	$\lambda I. \left\{ a \not\rightsquigarrow^{F_{ab}} b, c \rightsquigarrow \not\!\!\!\!\!\!\bigcirc^{F_c} \in I \,\middle|\, a, b, c \neq s_{j-1} \right\}$ $\cup \left\{ a \not\rightsquigarrow^{\mathcal{F}} s_{j-1} \,\middle|\, a \in \mathsf{dom}(\tau) \setminus \{s_{j-1}\} \wedge \langle a, s_{j-1}.(\kappa.f : \mathsf{t}) \rangle \notin \mathcal{MR}_\tau \right\}$ $\cup \left\{ a \not\rightsquigarrow^F s_{j-1} \,\middle|\, \begin{array}{l} a \neq s_{j-1} \wedge \langle a, s_{j-1}.(\kappa.f : \mathsf{t}) \rangle \in \mathcal{MR}_\tau \wedge \\ a \not\rightsquigarrow^F b \in I \wedge \left(b, s_{j-1}.(\kappa.f : \mathsf{t}) \right) \in \mathcal{DA}_\tau \end{array} \right\}$ $\cup \left\{ s_{j-1} \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, b \in \mathsf{dom}(\tau) \setminus \{s_{j-1}\} \wedge \langle s_{j-1}.(\kappa.f : \mathsf{t}), b \rangle \notin \mathcal{MR}_\tau \right\}$ $\cup \left\{ s_{j-1} \not\rightsquigarrow^{F'} b \,\middle|\, \begin{array}{l} b \neq s_{j-1} \wedge \langle s_{j-1}.(\kappa.f : \mathsf{t}), b \rangle \in \mathcal{MR}_\tau \wedge \\ s_{j-1} \not\rightsquigarrow^F b \in I \wedge F' = F \cup \{\kappa_k.f_k : \mathsf{t}_k \in \mathcal{F} \,|\mathsf{t}_k \notin \mathsf{T}(\mathsf{t})\} \end{array} \right\}$ $\cup \left\{ s_{j-1} \not\rightsquigarrow^F s_{j-1} \,\middle|\, a \not\rightsquigarrow^F a \in I \wedge \left(a, s_{j-1}.(\kappa.f : \mathsf{t}) \right) \in \mathcal{DA}_\tau \right\}$ $\cup \left\{ s_{j-1} \rightsquigarrow \not\!\!\!\!\!\!\bigcirc^{F'} \,\middle|\, \begin{array}{l} s_{j-1} \rightsquigarrow \not\!\!\!\!\!\!\bigcirc^F \in I \wedge \\ F' = F \cup \{\kappa_k.f_k : \mathsf{t}_k \in \mathcal{F} \,|\mathsf{t}_k \notin \mathsf{T}(\mathsf{t})\} \end{array} \right\}$

Continue in the following page

Continue from previous page

Π	Instruction	Propagation Rule
#6	putfield $\kappa.f$: t	$\lambda I.\left\{ a \not\rightsquigarrow^{F} b \in I \,\middle\vert\, \begin{array}{l} a, b \notin \{s_{j-1}, s_{j-2}\} \wedge \\ \left(\langle a, s_{j-2}\rangle \notin \mathcal{MR}_\tau \vee \langle s_{j-1}, b\rangle \notin \mathcal{MR}_\tau\right) \end{array} \right\}$ $\cup \left\{ a \not\rightsquigarrow^{F'} b \,\middle\vert\, \begin{array}{l} 1.\ a, b \notin \{s_{j-1}, s_{j-2}\} \wedge \langle a, s_{j-2}\rangle, \langle s_{j-1}, b\rangle \in \mathcal{MR}_\tau \\ 2.\ a \not\rightsquigarrow^{F_{ab}} b,\ a \not\rightsquigarrow^{F_{a2}} s_{j-2},\ s_{j-1} \not\rightsquigarrow^{F_{1b}} b \in I \\ 3.\ F' = \{F_{ab} \cap F_{a2} \cap F_{1b}\} \setminus \{\kappa.f : \mathsf{t}\} \end{array} \right\}$ $\cup \left\{ c \rightsquigarrow \circlearrowleft^{F} \in I \,\middle\vert\, \begin{array}{l} c \notin \{s_{j-1}, s_{j-2}\} \wedge \\ \langle c, s_{j-1}\rangle, \langle c, s_{j-2}\rangle \notin \mathcal{MR}_\tau \end{array} \right\}$ $\cup \left\{ c \rightsquigarrow \circlearrowleft^{F'} \,\middle\vert\, \begin{array}{l} 1.\ c \notin \{s_{j-1}, s_{j-2}\} \wedge \langle s_{j-1}, s_{j-2}\rangle \notin \mathcal{MR}_\tau \\ 2.\ \langle c, s_{j-2}\rangle \in \mathcal{MR}_\tau \vee \langle c, s_{j-1}\rangle \in \mathcal{MR}_\tau \\ 3.\ c \rightsquigarrow \circlearrowleft^{F_c},\ s_{j-1} \rightsquigarrow \circlearrowleft^{F_{j1}} \in I \wedge F' = F_c \cap F_{j1} \end{array} \right\}$ $\cup \left\{ c \rightsquigarrow \circlearrowleft^{F'} \,\middle\vert\, \begin{array}{l} 1.\ c \notin \{s_{j-1}, s_{j-2}\} \wedge \langle s_{j-1}, s_{j-2}\rangle \in \mathcal{MR}_\tau \\ 2.\ \langle c, s_{j-1}\rangle \in \mathcal{MR}_\tau \vee \langle c, s_{j-2}\rangle \in \mathcal{MR}_\tau \\ 3.\ s_{j-1} \not\rightsquigarrow^{F_{12}} s_{j-2},\ c \rightsquigarrow \circlearrowleft^{F_c},\ s_{j-1} \rightsquigarrow \circlearrowleft^{F_{j1}} \in I \\ 4.\ F' = \left\{F_c \cap F_{j1} \cap F_{12}\right\} \setminus \{\kappa.f : \mathsf{t}\} \end{array} \right\}$
#7	catch, excp_is K ifne t, ifeq t	$\lambda I.\left\{ a \not\rightsquigarrow^{F_{ab}} b,\ c \rightsquigarrow \circlearrowleft^{F_c} \in I \,\middle\vert\, a, b, c \in \mathsf{dom}(\tau') \right\}$

Final arcs. *For each* return t *and* throw κ *occurring in a method or in a constructor m of P, there are simple arcs from* [return t] *to* [exit@m] *and from* [throw κ] *to* [exception@m] *respectively, with one of the propagation rules in Table 6.2.*

Table 6.2: Propagation Rules of Final Arcs

Π	Instruction	Propagation Rule
#8	return *void*	$\lambda I.\left\{ a \not\rightsquigarrow^{F_{ab}} b,\ c \rightsquigarrow \circlearrowleft^{F_c} \in I \,\middle\vert\, a, b, c \notin \{s_0, \ldots, s_{j-1}\} \right\}$
#9	return t *with* t ≠ void throw κ	$\lambda I.\left\{ \left(a \not\rightsquigarrow^{F} b\right)\left[s_{j-1} \mapsto s_0\right] \,\middle\vert\, a \not\rightsquigarrow^{F} b \in I \wedge a, b \notin \{s_0, \ldots, s_{j-2}\} \right\}$ $\cup \left\{ \left(c \rightsquigarrow \circlearrowleft^{F}\right)\left[s_{j-1} \mapsto s_0\right] \,\middle\vert\, c \rightsquigarrow \circlearrowleft^{F} \in I \wedge c \notin \{s_0, \ldots, s_{j-2}\} \right\}$

Parameter passing arcs. *For each* ins_c = call $m_1 \ldots m_k$ *to a method with π parameters (including* this*), we build a simple arc from* [ins_c] *to the node corresponding to the first bytecode of m_w with the propagation rule in Table 6.3, for each $1 \leq w \leq k$.*

Table 6.3: Propagation Rules of Parameter Passing Arcs

Π	Instruction	Propagation Rule
#10	call $m_1 \dots m_k$	$\lambda I. \left\{ \left(a \not\rightsquigarrow^F b \right) \left[\begin{smallmatrix} s_{j-\pi} \mapsto l_0 \\ \dots \\ s_{j-1} \mapsto l_{\pi-1} \end{smallmatrix} \right] \middle\| \, a \not\rightsquigarrow^F b \in I \wedge a, b \in \{s_{j-\pi}, \dots, s_{j-1}\} \right\}$ $\cup \left\{ \left(c \rightsquigarrow \not\circlearrowleft^F \right) \left[\begin{smallmatrix} s_{j-\pi} \mapsto l_0 \\ \dots \\ s_{j-1} \mapsto l_{\pi-1} \end{smallmatrix} \right] \middle\| \, c \rightsquigarrow \not\circlearrowleft^F \in I \wedge c \in \{s_{j-\pi}, \dots, s_{j-1}\} \right\}$

Side-effects arc. *For each* ins_c = call $m_1 \dots m_k$ *to a method with* π *parameters (including* this*) returning no value (*void*) and each subsequent bytecode* ins′*, we build a multi-arc from a node* C = $\boxed{\text{call } m_1 \dots m_k}$ *and* E = $\boxed{\text{exit@}m_w}$ *(2 sources, in that order) to* Q = $\boxed{\text{ins}'}$*, where* ins′ *is not a* catch *for each* $1 \leq w \leq k$*. The propagation rule is given in Table 6.4, where* max = $j - \pi$*. Furthermore, we denote* τ *and* τ' *as the static type information at* C *and* Q *respectively.*

Table 6.4: Propagation Rules of Side Effects Arcs

Π	Instruction	Propagation Rule
#11	call $m_1 \dots m_k$	$\lambda I_1.\lambda I_2.\lambda max. \left\{ a \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, a, b \in \mathsf{dom}(\tau') \wedge \langle a, b \rangle \notin \mathcal{MR}_{\tau'} \right\}$ $\cup \left\{ \begin{array}{l} a \not\rightsquigarrow^{F_1} b, \\ a \rightsquigarrow \not\circlearrowleft^{F_2} \in I_1 \end{array} \middle| \begin{array}{l} a, b \in L \cup \{s_0, \dots, s_{max-1}\} \wedge \\ \forall j - \pi \leq p < j, (a, s_p) \notin \mathcal{MS}_\tau \end{array} \right\}$ $\cup \left\{ \begin{array}{l} a \not\rightsquigarrow^{F_1} b, \\ a \rightsquigarrow \not\circlearrowleft^{F_2} \in I_1 \end{array} \middle| \begin{array}{l} a, b \in L \cup \{s_0, \dots, s_{max-1}\} \wedge \\ \forall j - \pi \leq p < j \,\big|\, (a, s_p) \in \mathcal{MS}_\tau, \\ \quad \forall \boxed{\text{ins}} \text{ in } m_w \,\big|\, \boxed{\text{ins = putfield } \kappa.f\text{:t}} \vee \boxed{\text{ins = call } m_1 \dots m_k}, \\ \quad \left[\text{no } \boxed{\text{store } \mathsf{l}_{p-j+\pi}} \text{ until } \boxed{\text{ins}} \text{ with type information } \tau^* \right] \wedge \\ \quad \text{if } \boxed{\text{ins = putfield } \kappa.f\text{:t}} \text{ linking } s_{j-2} \text{ to } s_{j-1}, \\ \qquad \langle l_{p-j+\pi}, s_{j-2} \rangle \notin \mathcal{MR}_{\tau*} \\ \quad \text{if } \boxed{\text{ins = call } m_1 \dots m_k} \text{ with actual params } s_k, \dots, s_{k-\phi}, \\ \qquad \forall q \in [k, k-\phi], (l_{p-j+\pi}, s_q) \notin \mathcal{MS}_{\tau*} \end{array} \right\}$ $\cup \left\{ a \not\rightsquigarrow^F b \,\middle|\, \begin{array}{l} 1.\ a, b \in L \cup \{s_0, \dots, s_{max-1}\} \wedge \langle a, b \rangle \in \mathcal{MR}_{\tau'} \\ 2.\ \exists j - \pi \leq p_a, p_b < j \,\big|\, (a, s_{p_a}), (b, s_{p_b}) \in \mathcal{DA}_\tau \wedge \\ \quad \left[\text{no } \boxed{\text{store } \mathsf{l}_{p_a-j+\pi}} \text{ nor } \boxed{\text{store } \mathsf{l}_{p_b-j+\pi}} \text{ occurs in } m_w \right] \\ 3.\ l_{p_a-j+\pi} \not\rightsquigarrow^F l_{p_b-j+\pi} \in I_2 \end{array} \right\}$ $\cup \left\{ c \rightsquigarrow \not\circlearrowleft^F \,\middle|\, \begin{array}{l} 1.\ c \in L \cup \{s_0, \dots, s_{max-1}\} \\ 2.\ \exists j - \pi \leq p < j \,\big|\, (c, s_p) \in \mathcal{DA}_\tau \wedge \\ \quad \left[\text{no } \boxed{\text{store } \mathsf{l}_{p-j+\pi}} \text{ occurs in } m_w \right] \\ 3.\ l_{p-j+\pi} \rightsquigarrow \not\circlearrowleft^F \in I_2 \end{array} \right\}$

Return value arc. *For each* ins_c = call $m_1 \dots m_k$ *to a method with* π *parameters (including* this*) returning a value of type* t $\in \mathbb{K}$ *and each subsequent bytecode* ins′ *distinct from* catch*, we build a multi-arc from* C = $\boxed{\text{call } m_1 \dots m_k}$ *and* E = $\boxed{\text{exit@}m_w}$ *(2 sources, in that*

order) to Q = $\boxed{\text{ins}'}$ *with the propagation rule defined in Table 6.5, for each* $1 \leq w \leq k$. *Furthermore, we denote* τ *and* τ' *as the static type information at* C *and* Q *respectively.*

Table 6.5: Propagation Rules of Return Value Arcs

Π	Instruction	Propagation Rule
#12	call $m_1 \dots m_k$	$\lambda I_1.\lambda I_2.\left\{ s_{j-\pi} \not\rightsquigarrow^{F_1} s_{j-\pi},\ s_{j-\pi} \rightsquigarrow \not\Delta^{F_2} \,\middle|\, s_0 \not\rightsquigarrow^{F_1} s_0,\ s_0 \rightsquigarrow \not\Delta^{F_2} \in I_2 \right\}$ $\cup \left\{ a \not\rightsquigarrow^{\mathcal{F}} s_{j-\pi} \,\middle|\, a \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\} \wedge \langle a, s_{j-\pi}\rangle \notin \mathcal{MR}_{\tau'} \right\}$ $\cup \left\{ a \not\rightsquigarrow^{F} s_{j-\pi} \,\middle|\, \begin{array}{l} 1.\ a \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\} \wedge \langle a, s_{j-\pi}\rangle \in \mathcal{MR}_{\tau'} \\ 2.\ \exists j - \pi \leq p < j \,\middle|\, (s_p, a) \in \mathcal{DA}_\tau \wedge \\ \quad \left[\textit{no } \boxed{\text{store } l_{p-j+\pi}} \textit{ occurs in } m_w \right] \\ 3.\ l_{p-j+\pi} \not\rightsquigarrow^{F} s_0 \in I_2 \end{array} \right\}$ $\cup \left\{ s_{j-\pi} \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, b \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\} \wedge \langle s_{j-\pi}, b\rangle \notin \mathcal{MR}_{\tau'} \right\}$ $\cup \left\{ s_{j-\pi} \not\rightsquigarrow^{F} b \,\middle|\, \begin{array}{l} 1.\ b \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\} \wedge \langle s_{j-\pi}, b\rangle \in \mathcal{MR}_{\tau'} \\ 2.\ \exists j - \pi \leq p < j \,\middle|\, (s_p, b) \in \mathcal{DA}_\tau \wedge \\ \quad \left[\textit{no } \boxed{\text{store } l_{p-j+\pi}} \textit{ occurs in } m_w \right] \\ 3.\ s_0 \not\rightsquigarrow^{F} l_{p-j+\pi} \in I_2 \end{array} \right\}$ $\cup\ \Pi^{\#11}(I_1, I_2, j - \pi)$

Exceptional arcs. *For each* ins *different from* call $m_1 \dots m_k$ *throwing an exception, immediately followed by a* catch, *a arc is built from* $\boxed{\text{ins}}$ *to* $\boxed{\text{catch}}$, *with one of the propagation rules in Table 6.6.*

Table 6.6: Propagation Rules of Exceptional Arcs

Π	Instruction	Propagation Rule
#13	throw κ	$\lambda I. \left\{ \left(a \not\rightsquigarrow^{F} b \right) \left[s_{j-1} \mapsto s_0 \right] \,\middle|\, a \not\rightsquigarrow^{F} b \in I \wedge a, b \notin \{s_0, \dots, s_{j-2}\} \right\}$ $\cup \left\{ \left(c \rightsquigarrow \not\Delta^{F} \right) \left[s_{j-1} \mapsto s_0 \right] \,\middle|\, c \rightsquigarrow \not\Delta^{F} \in I \wedge c \notin \{s_0, \dots, s_{j-2}\} \right\}$
#14	new κ getfield $\kappa.f$:t putfield $\kappa.f$:t	$\lambda I. \left\{ a \not\rightsquigarrow^{F_{ab}} b,\ c \rightsquigarrow \not\Delta^{F_c} \in I \,\middle|\, a, b, c \notin \{s_0, \dots, s_{j-1}\} \right\}$ $\cup \left\{ a \not\rightsquigarrow^{\mathcal{F}} s_0,\ s_0 \not\rightsquigarrow^{\mathcal{F}} a \,|\, a \in L \right\} \cup \left\{ s_0 \not\rightsquigarrow^{\mathcal{F}} s_0,\ s_0 \rightsquigarrow \not\Delta^{\mathcal{F}} \right\}$

For each ins = call $m_1 \dots m_k$, *method with* π *parameters (including* this*) throwing an exception, immediately followed by a* catch, *we build a multi-arc from a node* C = $\boxed{\text{call } m_1 \dots m_k}$ *and* E = $\boxed{\text{exception@}m_w}$ *(2 sources, in that order) to* Q = $\boxed{\text{catch}}$, *for each* $1 \leq w \leq k$. *The propagation rule is given in Table 6.7. Furthermore, we denote* τ *and* τ' *as the static type information at* C *and* Q *respectively.*

Table 6.7: Propagation Rules of Multi-Exceptional Arcs

Π	*Instruction*	*Propagation Rule*
#15	call $m_1 \dots m_k$	$\lambda I_1.\lambda I_2.\Pi^{\#11}(I_1, I_2, 0) \cup \left\{s_0 \not\rightsquigarrow^{\mathcal{F}} s_0\right\}$ $\cup \left\{a \not\rightsquigarrow^{\mathcal{F}} s_0 \,\middle|\, a \in \mathsf{dom}(\tau') \setminus \{s_0\} \wedge \langle a, s_0\rangle \notin \mathcal{MR}_{\tau'}\right\}$ $\cup \left\{s_0 \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, b \in \mathsf{dom}(\tau') \setminus \{s_0\} \wedge \langle s_0, b\rangle \notin \mathcal{MR}_{\tau'}\right\}$ $\cup \left\{s_0 \rightsquigarrow^{\not\mathcal{F}} \,\middle|\, \forall a \in \mathsf{dom}(\tau') \setminus \{s_0\}, \langle s_0, a\rangle \notin \mathcal{MR}_{\tau'}\right\}$

Definition 6.2 specifies how the ACG is built from the program under analysis. For each bytecode instruction $\boxed{\mathsf{ins}}$ in the graph, *decorated* by an element of our abstract domain defined in 5.1 and representing an under-approximation of our two properties that we want to state at that point. These rules state how this approximation is propagated along the arcs of the ACG; before starting to fully explain them through and *abstract execution* example we introduce the concept of Normalized Propagation Rules that help us to formalize how to compute the abstract set I for every node whose in-degree i.e., the number of head endpoints adjacent to it, is greater than 1.

Definition 6.3 (Normalize Propagation Rules)**.** *Let $\tau \in \mathcal{T}$, $I \in \mathsf{A}_\tau$. We define the* Abstract Function Φ *as*

$$\Phi^{\#i} = \mathsf{normalize} \circ \Pi^{\#i}$$

i.e., the composition of two functions: the normalize *operator, Definition 5.3, and one of the propagation rules $\Pi^{\#i}$ discussed in Definition 6.2.*

With abuse of notation, we will use Π and Φ instead of $\Pi^{\#i}$ and $\Phi^{\#i}$ respectively, either to express statements about any rule or when the particular propagation rule is clear by the context. In this way, to build the abstract set of a node with in-degree greater that 1 we simply use the join operator $\sqcup_{\mathsf{A}}$.

Example 6.1. *Let I_X the abstract set of the node X with in-degree greater than 1 ($k \geq 1$), hence $I_1, \dots, I_k$ the abstract sets of its predecessors $Y_1, \dots, Y_k$ and $\Pi_1, \dots, \Pi_k$ the propagation rules associated to the arcs linking $Y_1, \dots, Y_k$, respectively, to node X. The we can simply calculate I_X as*

$$I_X = \Phi_1(I_1) \sqcup_{\mathsf{A}} \cdots \sqcup_{\mathsf{A}} \Phi_k(I_k) \overset{\text{by Def. 5.1}}{=} \Phi_1(I_1) \cap \cdots \cap \Phi_k(I_k)$$

In fact, since the normalize *operator takes every token depending to a set of fields F and add the same relations for every $F' \subseteq F$, we can simply use the join operator $\sqcup_{\mathsf{A}} = \cap$ to estimate the least upper bound of these sets and, hence, be less precise but sound respect all of them (since the least upper bound approximates all the values). In the next section, the Example 6.6 will show how to handle nodes with in-degree greater than* 1.

We are going to introduce a lemma and a subsequent corollary in order to understand the difference of applying the concretization map γ_τ to the result of a propagation rule Π respect to the result of its normalized version Φ.

Lemma 6.1. *Let $\tau \in \mathcal{T}$, $I \subseteq \mathsf{A}_\tau$, $\Pi^{\#i}$ one of the propagation rules and $\Phi^{\#i}$ the corresponding normalize one. Then $\Pi^{\#i}(I) \subseteq \Phi^{\#i}(I)$.*

Proof. We have:

$$\Pi^{\#i}(I) \overset{\text{by Def. 5.3}}{\subseteq} \mathsf{normalize}(\Pi^{\#i}(I)) \overset{\text{by Def. 6.3}}{=} \Phi^{\#i}(I)$$

□

Then, since in Subsection 5.3 we proved that $\langle \mathsf{C}_\tau, \alpha_\tau, \gamma_\tau, \mathsf{A}_\tau \rangle$ is a Galois Connection, we have that α and γ are both monotonic and hence the following corollary of Lemma 6.1 holds.

Corollary 6.1.1. *Let $\tau \in \mathcal{T}$, $I \subseteq \mathsf{A}_\tau$, $\Pi^{\#i}$ one of the propagation rules and $\Phi^{\#i}$ the corresponding normalize one. Then $\gamma_\tau(\Pi^{\#i}(I)) = \gamma_\tau(\Phi^{\#i}(I))$*

Proof. $\subseteq$) It follows by Lemma 6.1 and by Definition of Galois Connection $\langle \mathsf{C}_\tau, \alpha_\tau, \gamma_\tau, \mathsf{A}_\tau \rangle$.
$\supseteq$) We are going to prove it for the unreachability tokens; about non-cyclical ones we can follow the same reasoning. The two propagation rules are both applied on a state σ and they lead to a state σ'.

Let $a \not\rightsquigarrow^{F} b \in \Phi^{\#i}(I)$. Then, by Definition 6.3, it must exist a maximal set $\overline{F}$ such that $F \subseteq \overline{F} \subseteq \mathcal{F}$, $a \not\rightsquigarrow^{\overline{F}} b \in \Phi^{\#i}(I)$ and for each $a \not\rightsquigarrow^{F''} b \in \Phi^{\#i}(I)$ we have $F'' \subseteq \overline{F}$. Then let be $\sigma' \in \gamma_\tau(\Phi^{\#i}(I))$, we have $a \not\rightsquigarrow^{\overline{F}'}_{\sigma'} b$ with $\overline{F} \subseteq \overline{F}'$ and, therefore, by Lemma 4.7, also $a \not\rightsquigarrow^{F'}_{\sigma'} b$ for each $F' \subseteq \overline{F}'$. Furthermore, by Definition 5.3, the token $a \not\rightsquigarrow^{\overline{F}} b$ is also inside $\Pi^{\#i}(I)$ and, since the propagation rule is applied on the same σ as the normalize one, we can conclude $\sigma' \in \gamma_\tau(\Pi^{\#i}(I))$. □

We are now able to define how to build the solution of our ACG with respect to the unreachability and non-cyclicity properties in order to obtain the most precise abstract set of tokens related with our properties for every node of it.

Definition 6.4 (Field-Sensitive Unreachability and Non-Cyclicity Analysis)**.** *A solution of an* ACG *is an assignment of an element $J_n \in \mathsf{A}_\tau$ to each node n of the* ACG*, where τ is the type environment associated to n, such that the normalize propagation rules Φ of the arcs is satisfied i.e., for every arc from nodes $n_1, \ldots, n_k$[2] to n' with normalize propagation rule $\lambda I_1, \ldots, \lambda I_k.\Phi(I_1, \ldots, I_k)$, the condition $\Phi(J_{n_1}, \ldots, J_{n_k}) \sqsubseteq_{\mathsf{A}} J_{n'}$ holds. The unreachability and non-cyclicity analysis of the program is the minimal solution of its ACG w.r.t. the partial order $\sqsubseteq_{\mathsf{A}}$ and hence, since $\sqsubseteq_{\mathsf{A}}$ is $\supseteq$, the* maximal solution *of its ACG w.r.t. $\supseteq$. The analysis at a bytecode instruction of the programm* ins *is $J_{\boxed{\mathsf{ins}}}$.*

We observe that a maximal solution exists since all the propagation rules are monotonic w.r.t. set inclusion $\supseteq$: it can hence be computed by starting from the complete approximation for every node i.e., $a \not\rightsquigarrow^{\mathcal{F}} b$ and $a \rightsquigarrow^{\mathcal{F}} \not\ni$ for every $a, b \in \mathsf{dom}(\tau)$ of that node, and propagating this approximation along the arcs, until reaching a *Fix Point*.

Namely, the main steps of the fix-point algorithm can be summarized as:

1. Propagate the rules along the ACG arcs, computing the simple intersection $I_1 \cap \cdots \cap I_n$ whenever there is a node with in-degree greater than 1.

2. Since ACG is mostly cyclical and hence information about nodes can be refined at the same step, continuously propagate them until the $(i{+}1)$-th step does not modify any information computed at i-th step i.e., until reaching a fix-point.

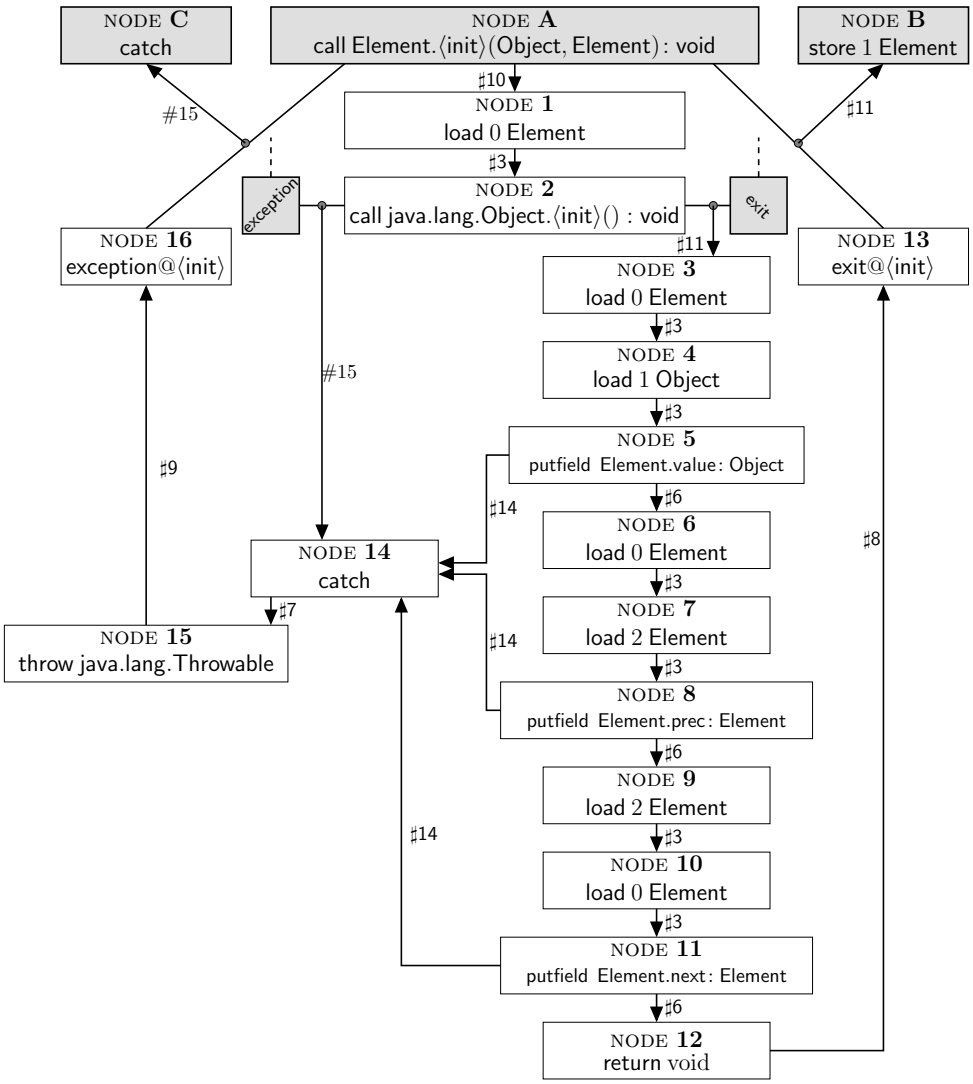


Figure 6.1: The ACG for the second constructor of Code 1

6.3 An Abstract Execution Example

Figure 6.1 shows the ACG built for the constructor in Figure 3.1. We use the gray to distinguish the nodes that are not part of the constructor: three nodes (A, B, C) of a caller of this constructor and two nodes of the callee of call `java.lang.Object.⟨init⟩: void`, which invokes the superclass $\langle init \rangle$ method, in this case `Object`. Each arc is decorated with the number of their associated normalized propagation rule. Finally we note that the graph for the whole program includes other nodes and arcs; in Figure 6.1 is shown only the relevant subgraph, providing an interesting execution of our analysis.

[2] According to the Definition 6.2, either $k = 1$ or $k = 2$

We are going to present how the propagation rules allow to assert some statements for each node of our graph. Let i_n and j_n be the number of local L and stack S variables at n respectively, and τ_n be the static type information available at n i.e., for each node n.

We suppose that at node A we have $i_A = 3$ and $j_A = 4$, with $l_1 = \texttt{top}$, $l_2 = \texttt{i} = 1$, while $s_1 = \mathsf{rec}$, $s_2 = \texttt{value}$, $s_3 = \texttt{top}$. Moreover, from the other static analyses that we exploit, we suppose to know that

$$\mathcal{DA}_{\tau_A} = \{(s_0, s_1), (l_1, s_3), (l_0, l_0), (l_1, l_1), (s_0, s_0), (s_1, s_1), (s_2, s_2), (s_3, s_3)\}$$

$$\mathcal{MR}_{\tau_A} = \{\langle s_0, s_1\rangle, \langle s_1, s_0\rangle, \langle l_1, s_3\rangle, \langle s_3, l_1\rangle, \langle l_0, l_0\rangle, \langle l_1, l_1\rangle, \langle s_0, s_0\rangle, \langle s_1, s_1\rangle, \langle s_2, s_2\rangle, \langle s_3, s_3\rangle\}$$

$$\mathcal{MS}_{\tau_A} = \{(s_0, s_1), (l_1, s_3), (l_0, l_0), (l_1, l_1), (s_0, s_0), (s_1, s_1), (s_2, s_2), (s_3, s_3)\}$$

At that point constructor $\mathsf{call}\ \texttt{Element.}\langle\texttt{init}\rangle(\texttt{Object}, \texttt{Element}) \colon \texttt{void}$ is invoked and we suppose that the analysis performed until node A provides the following approximation:

$$I_A = \Big\{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\},$$
$$s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}, s_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}, s_3 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}\Big\} \cup$$
$$\Big\{\{l_0, l_1, s_0, s_1, s_2, s_3\} \rightsquigarrow\!\!\!\!\!\!\diagup^{\mathcal{F}}\Big\}$$

where $\mathcal{F} = \{\mathsf{El}.value, \mathsf{E}.prec, \mathsf{El}.next\}$, with El an abbreviation for the class $\mathsf{Element}$. Obviously, we do not mention l_2 since it has a primitive type (int) end hence for sure $\rho(l_2)$ can not share with other locations in any state σ.

It is worth noting that in all sets of this example, instead of writing every token separately, we have used a more compact notation by unifying them whether they share the same set of fields: for instance, $a \not\rightsquigarrow^F x$, $a \not\rightsquigarrow^F y$ become $a \not\rightsquigarrow^F \{x, y\}$ and $a \rightsquigarrow\!\!\!\!\!\!\diagup^F$, $b \rightsquigarrow\!\!\!\!\!\!\diagup^F$ become $\{a, b\} \rightsquigarrow\!\!\!\!\!\!\diagup^F$. Furthermore, in order to improve the readability of the example, we include in the abstract sets only the maximal elements resulting from the Π.

Let us explain the propagation rules given in Definition 6.2.

Sequential arcs link an instruction to its immediate successor.

Arc **#1**, starting from a node corresponding to a dupt and going to the node corresponding to its immediate successor in the Control Flow Graph, states that the information of the node remains unchanged at its successor's node as well: $\lambda I.I$. Moreover, since the new top of the stack, s_j, is an alias of the old one, s_{j-1}, its unreachability and non-cyclicity information is the same as s_j: $I[s_{j-1} \mapsto s_j]$. Furthermore, we have to add the new unreachability information between these two variables: since they are alias, we add two new unreachability relations among s_{j-1} and s_j w.r.t. the same set of fields of the unreachability relations between s_{j-1} and itself.

Arc **#2** starts from a node corresponding either to $\mathsf{new}\ \kappa$ or to $\mathsf{const}\ v$. In both cases, a new *fresh* variable is pushed on top of the stack. It means that this new variable, s_j, is only reachable from itself and hence, in addition to the information provided by the previous node ($\lambda I.I$), we have to add new unreachability relations w.r.t. all the program fields between s_j and all the other variables in $\mathsf{dom}(\tau)$: $\Big\{a \not\rightsquigarrow^{\mathcal{F}} s_j, s_j \not\rightsquigarrow^{\mathcal{F}} a \,|\, a \in dom(\tau)\Big\}$.

Moreover, we add the non-cyclicity information of the new top of the stack, $s_j \rightsquigarrow\!\!\!\!\!\!\diagup^{\mathcal{F}}$, and also the unreachability information between this new variable and itself, $s_j \not\rightsquigarrow^{\mathcal{F}} s_j$; in both cases the set of fields is $\mathcal{F}$ because s_j is fresh and, therefore, there can not be paths starting or ending at the location bound to it.

Arc **#3**, starting from a node corresponding to load k t, has been built through the same reasoning of the propagation rule #1 with l_k instead of s_{j-1}.

Arc **#4**, starting from a node corresponding to store k t, states that all the information that does not relate to l_k and s_{j-1} (the old top of the stack) remains unchanged. Contrariwise the information regarding these two variables is substituted with new one regarding l_k, having the same unreachability and non-ciclicity properties of the old top of the stack: $\left(a \not\rightsquigarrow^F b\right)\left[s_{j-1} \mapsto l_k\right]$ and $\left(c \rightsquigarrow \not\circlearrowleft^F\right)\left[s_{j-1} \mapsto l_k\right]$.

Arc **#5**, starting from a node corresponding to getfield $\kappa.f$: t, is more complicated. First of all we exploit two other static analyses: possible reachability and definite aliasing. Since getfield $\kappa.f$: t replaces the old top of the stack, s_{j-1}, with the value of its field $\kappa.f$: t, all reachability properties that do not consider s_{j-1} are still valid after its execution, as expressed in the first set of the corresponding rule. The reachability information between this new variable and itself is inferred by checking if it is aliased by another variable: $\left(a, s_{j-1}.(\kappa.f : \mathsf{t})\right) \in \mathcal{DA}_\tau$. In that case, as shown in its sixth set, the unreachability information of s_{j-1} is the same one of the variable it is alias by. Additionally, for all the variables in $\mathsf{dom}(\tau) \setminus s_{j-1}$ that may not reach this field or that may not be reached by this field, we can add the unreachability information with the new variables w.r.t. all the program fields. Contrariwise the variables that may reach the field of s_{j-1} are treated in the second set: if we found a relation $a \not\rightsquigarrow^F b \in I$ where b is alias with the field of s_{j-1} i.e., $\left(b, s_{j-1}.(\kappa.f : \mathsf{t})\right) \in \mathcal{DA}_\tau$, we obtain *safety* information about the new top of the stack and then we can use it to produce the new token $a \not\rightsquigarrow^F s_{j-1}$. Regarding the variables that may be reached by this field (fifth set) we maintain all the unreachability information regarding s_{j-1} and we also add all the other fields whose type cannot be reached by the type of our field $\kappa.f$: t: $F' = F \cup \{\kappa_k.f_k : \mathsf{t}_k \in \mathcal{F} \mid \mathsf{t}_k \notin \mathsf{T}(\mathsf{t})\}$. That is because there could be other paths in the heap that involve both the old top of the stack and b without having $\kappa.f$: t inside them. Therefore, since F does not contain the fields of these paths, in order to be more precise, we can add them in F', the set stating the unreachability information between the new top of the stack and b. Finally, the last set, with the non-cyclicity information of the new top of the stack, is built with the same reasoning just explained for the fifth set.

Arc **#6**, starting from a node corresponding to putfield $\kappa.f$: t, states, in the first and third sets, that the unreachability and non-cyclicity information of the variables' pairs (respectively of the variables) that does not reach or are not reached by the topmost two values of the stack[3] remains unchanged. Contrariwise if a variables' pair $\langle a, b\rangle$ is such that the former may reach s_{j-2} and the latter may be reached by s_{j-1}, the putfield $\kappa.f$: t could create new paths between the two locations bound to these two variables. Therefore, in order to ensure the Definition 4.7, the set of fields that defines the unreachability information between a and b must be consistent also both with the unreachability information of $a \not\rightsquigarrow^{F_{a2}} s_{j-2}$ and $s_{j-1} \not\rightsquigarrow^{F_{1b}} b$: $F' = \{F_{ab} \cap F_{a2} \cap F_{1b}\}$. Furthermore, since putfield $\kappa.f$: t links s_{j-2} with s_{j-1} through $\kappa.f$: t, also this field must be delete from any set of unreachability information of variables' pair like $\langle a, b\rangle$: $F' = \{F_{ab} \cap F_{a2} \cap F_{1b}\} \setminus \{\kappa.f : \mathsf{t}\}$.
Regarding the non-cyclicity information we distinguish if s_{j-1} may or not reach s_{j-2} i.e., if the putfield $\kappa.f$: t will create or not a new cycle between the locations in memory. If s_{j-1} may not reach s_{j-2}, for sure new cycles will not be created by execution of that instruction. However, since s_{j-2} is linked to s_{j-1}, also all the variables that may reach s_{j-2}

[3] We remember that, by putfield $\kappa.f$: t semantics defined in Figure 3.1, the topmost two values of the stack disappear from the it after the instruction is executed.

will be able to reach s_{j-1} and, hence, in order to maintain the consistency with Definition 4.8, their non-cyclicity information F_c must take into account also the non-cyclicity information of s_{j-1} i.e., F_{j-1}: $F' = F_c \cap F_{j1}$. On the other hand, if s_{j-1} may reach s_{j-1} a new cycle could be created after the execution of putfield $\kappa.f:\mathsf{t}$ and, therefore we have also to delete all the fields that, potentially, are part of that cycle: this information is provided by unreachability information since we know which fields are not part of any path from s_{j-1} to s_{j-2} i.e., F_{12} such that $s_{j-1} \not\rightsquigarrow^{F_{12}} s_{j-2} \in I$. This set minus the field $\kappa.f:\mathsf{t}$ is the set of fields that is not surely part of the new cycle that could be formed: $F' = \left\{F_c \cap F_{j1} \cap F_{12}\right\} \setminus \{\kappa.f:\mathsf{t}\}$. It is worth noting that this is the only case where we use unreachability information to state a non-cyclicity property.

Arc **#7**, starting from a node corresponding to catch, excp_is K, ifne t or ifeq t, states that these instructions does not changed the unreachability and non-cyclicity information. That is because the state after their execution is the same as before.

Example 6.2. *Consider, for instance, nodes* 10*,*11*,* 12 *in Figure 6.1, and suppose that the unreachability and non-cyclicity information at node* 10 *is*

$$I_{10} = \Big\{ l_0 \not\rightsquigarrow^{\mathcal{F}} l_0,\ l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec,\mathsf{El}.next\}} l_1,\ l_0 \not\rightsquigarrow^{\{\mathsf{El}.value,\mathsf{El}.next\}} \{l_2, s_0\},\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},$$
$$l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},\ s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},\ \{l_0, l_1, l_2, s_0\} \rightsquigarrow^{\not\ni\mathcal{F}} \Big\}$$

Node 10 *and* 11 *are linked by a sequential arc with propagation rule #3, while node* 11 *and* 12 *are linked by a sequential arc with propagation rule #6. By Definition 6.2 we can compute* $I_{11} \subseteq I_{10} \cup I_{10}[l_0 \mapsto s_1] \cup \{l_0 \not\rightsquigarrow^{\mathcal{F}} s_1,\ s_1 \not\rightsquigarrow^{\mathcal{F}} l_0\}$ *by obtaining*

$$I_{11} = \Big\{ l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_1\} l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec,\mathsf{El}.next\}} l_1,\ l_0 \not\rightsquigarrow^{\{\mathsf{El}.value,\mathsf{El}.next\}} \{l_2, s_0\},\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},$$
$$l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},\ s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},\ s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_1\},\ s_1 \not\rightsquigarrow^{\{\mathsf{El}.prec,\mathsf{El}.next\}} l_1,$$
$$s_1 \not\rightsquigarrow^{\{\mathsf{El}.value,\mathsf{El}.next\}} \{l_2, s_0\},\ \{l_0, l_1, l_2, s_0, s_1\} \rightsquigarrow^{\not\ni\mathcal{F}} \Big\}.$$

In order to build I_{12} *we have to execute the propagation rule of the* putfield $\kappa.f:\mathsf{t}$*, since* s_0 *is linked to* s_1 *by the field* $\mathsf{El}.next$*. The application of that rule leads to this set:*

$$I_{12} = \Big\{ l_0 \not\rightsquigarrow^{\mathcal{F}} l_0,\ l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec,\mathsf{El}.next\}} l_1,\ l_0 \not\rightsquigarrow^{\{\mathsf{El}.value,\mathsf{El}.next\}} l_2,\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\},$$
$$l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_1, l_2\},\ l_2 \not\rightsquigarrow^{\{\mathsf{El}.value,\mathsf{El}.prec\}} l_0,\ l_1 \rightsquigarrow^{\not\ni\mathcal{F}}, \{l_0, l_2\} \rightsquigarrow^{\not\ni\{\mathsf{El}.value\}} \Big\}$$

We note that now there is no more the fields $\mathsf{El}.next$ *inside the set of fields associated to the pair* $\langle l_2, l_0\rangle$*. That because* $\langle l_2, s_0\rangle, \langle s_1, l_0\rangle \in \mathcal{MR}_{\tau_{11}}$ *and hence their set of filed changes according to the second set of the Rule #6:*

$$F_{l_2,l_0} = \{F_{l_2,l_0} \cap F_{l_2,s_0} \cap F_{s_1,l_0}\} \setminus \{\mathsf{El}.next\} = \{\mathcal{F} \cap \mathcal{F} \cap \mathcal{F}\} \setminus \{\mathsf{El}.next\} = \{\mathsf{El}.value, \mathsf{El}.prec\}$$

Moreover, by executing this rule also the non-cyclicity tokens related to l_0 *and* l_2 *change; that is because a new cycle could be formed*[4] *by executing the* putfield $\kappa.f:\mathsf{t}$*. Indeed, since* $\langle l_2, s_0\rangle, \langle l_0, s_1\rangle, \langle s_1, s_0\rangle \in \mathcal{MR}_{\tau_{11}}$*, according to the last set of the Rule #6, we have to modify the non-cyclicity information of* l_2 *and* l_0 *and, in this case, the operation and then the result will be the same. We show it only for* l_2*, the latter is the same:*

$$F_{l_2} = \{F_{l_2} \cap F_{l_0} \cap F_{s_1,s_0}\} \setminus \{\mathsf{El}.next\} = \{\mathcal{F} \cap \mathcal{F} \cap \{\mathsf{El}.value, \mathsf{El}.next\}\} \setminus \{\mathsf{El}.next\} = \{\mathsf{El}.value\}$$

[4] In the concrete execution a new cycle is really created inside the heap.

We note that, as expressed in Code 1, during the execution of the second constructor a new cycle between the two location bound to this *and* $prec$ *is established since* this *is linked to* prec *through the field* El.$prec$*, whereas* prec *is linked to* this *through the field* El.$next$*. Therefore the fields* El.$prec$ *and* El.$next$ *set up a path* $\mathcal{P} = \{\mathsf{El}.prec, \mathsf{El}.next\}$ *such that* $\rho(\mathsf{this}) \rightsquigarrow^{\mathcal{P}} \rho(\mathsf{this})$ *and, respectively,* $\rho(\mathsf{prec}) \rightsquigarrow^{\mathcal{P}} \rho(\mathsf{prec})$ *and hence, according to the Definition 4.8, they must not be inside* $F_{\mathsf{this}} = F_{l_0}$ *and* $F_{\mathsf{prec}} = F_{l_2}$.

Final arcs feed nodes $\boxed{\mathsf{exit@}m}$ or $\boxed{\mathsf{exception@}m}$ for each method or constructor of m. The former (respectively the latter) contains the information present in all states at a non-exceptional (respectively exceptional) end of m.

Hence, $\boxed{\mathsf{exit@}m}$ is the sink of the arcs starting from the bytecode instruction return t present inside m. The propagation rule states that either the stack is emptied at the end of execution of m i.e., when t = void (rule **#8** is applied) or only one element survives i.e., the returned value (rule **#9**).

Similarly, $\boxed{\mathsf{exception@}m}$ is the sink of the bytecode instruction throw κ with no exception handler in m (i.e., not followed by a catch inside m). The rule is the same as return t with $t \neq$ void (rule #9) since only a stack element, the topmost s_{j-1}, survives and it is renamed into the exception object s_0. We observe that only instructions throw κ are allowed to throw an exception to the caller since, in our representation of the code as basic blocks, all other instructions that might throw an exception are always linked to an exception handler, possibly minimal (as the three putfield $\kappa.f$:t in Figure 6.1).

Example 6.3. *Consider, for instance, nodes* 12 *and* 13 *in Figure 6.1, and suppose that the unreachability and non-cyclicity approximation at node* 12 *is*

$$I_{12} = \Big\{ l_0 \not\rightsquigarrow^{\mathcal{F}} l_0,\ l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1,\ l_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} l_2,\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\},$$
$$l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_1, l_2\},\ l_2 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.prec\}} l_0,\ l_1 \rightsquigarrow \not\emptyset^{\mathcal{F}},\ \{l_0, l_2\} \rightsquigarrow \not\emptyset^{\{\mathsf{El}.value\}} \Big\}.$$

Node 12 *and* 13 *are linked by a final arc with propagation rule #8. By Definition 6.2,* I_{13} *contains all the tokens of* I_{12} *containing no stack variable, and since* $j_{12} = 0$ *we conclude that* $I_{13} = I_{12}$.

Parameter passing arcs link every node corresponding to a method call to the node corresponding to the first bytecode instruction of the method(s) m_w that might be called here. Propagation rule **#10** simply states that the actual parameters of m_w, held in the stack variables $s_{j-\pi}, \ldots, s_{j-1}$, are renamed into its formal parameters i.e., the local variables $l_0, \ldots, l_{\pi-1}$. No other variables exist at the beginning of m_w.

Example 6.4. *Consider, for instance, node* A *and* 1 *in Figure 6.1. We remember that we assumed that the unreachability and non-cyclicity information at node* A *is:*

$$I_A = \Big\{ l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\},\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\},\ s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\},$$
$$s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\},\ s_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\},\ s_3 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\} \Big\} \cup$$
$$\Big\{ \{l_0, l_1, s_0, s_1, s_2, s_3\} \rightsquigarrow \not\emptyset^{\mathcal{F}} \Big\}$$

Node A *and* 1 *are linked by a parameter passing arc with propagation rule #10. We have* $j_A = 4$ *and* $\pi = 3$ *(stack element* s_1*,* s_2 *and* s_3 *hold the actual parameters of a call to this*

constructor). Hence, by Defintion 6.2, it must be:

$$\left\{\left(a \not\rightsquigarrow^F b\right)\begin{bmatrix} s_1 \mapsto l_0 \\ s_2 \mapsto l_1 \\ s_3 \mapsto l_2 \end{bmatrix} \middle| \; a \not\rightsquigarrow^F b \in I_A \wedge a, b \in \{s_1, s_2, s_3\}\right\} \cup \left\{\left(c \rightsquigarrow \not\circlearrowleft^F\right)\begin{bmatrix} s_1 \mapsto l_0 \\ s_2 \mapsto l_1 \\ s_3 \mapsto l_2 \end{bmatrix} \middle| \; c \rightsquigarrow \not\circlearrowleft^F \in I_A \wedge c \in \{s_1, s_2, s_3\}\right\} \supseteq I_1$$

In particular, we obtain:

$$I_1 = \left\{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, \{l_0, l_1, l_2\} \rightsquigarrow \not\circlearrowleft^{\mathcal{F}}\right\}$$

Side-effect multi-arc links the nodes $\boxed{\text{call } m_1 \dots m_k}$ and $\boxed{\text{exit@}\, m_w}$ to the next instruction after the non-exceptional ending of the method m_w only if it does not return any value (void). Furthermore, since the Propagation Rule **#11** of this arc determines unreachability and non-cyclicity information of the caller's variables after the execution of the method m_w, it is also used as part of the propagation rules #12 and #15. For this reason, we add the (meta-)parameter max as input: $max = j - \pi$ when I_2 is the information set of a node $\boxed{\text{exit@}\, m_w}$, while $max = 0$ when I_2 is the information set of a node $\boxed{\text{exception@}\, m_w}$. Finally, before explaining the Rule, we remind that, when we use τ' as subscript of another static analysis result (e.g. $\mathcal{MR}_{\tau'}$), we mean that it is the set after the abstract execution of that node and we can exploit this *forward information* because we have assumed that these analyses are processed asynchronously (see Section 6.1).

Rule #11 states that between all the variables pairs $\langle a, b\rangle$ such that a cannot reach b after the execution of method m_w the unreachability information is $a \not\rightsquigarrow^{\mathcal{F}} b$.

On the other hand, all the unreachability tokens before the method execution are still valid (first set), provided that the first variable a cannot *share* with any actual parameters at that program point: $\forall j - \pi \leq p < j, (a, s_p) \notin \mathcal{MS}_\tau$. In this way, inside m_w we are sure that no paths through locations which share with a are modified and hence its unreachability information remains unchanged. The same reasoning about non-cyclicity tokens leads to the second set of the rule.

We can still improve the precision of this rule by keeping other tokens of I_1 also if a may share with one or more actual parameters (and hence share with it). The third set states that we can add other tokens of I_1 also if the variable may reach one or more actual parameters of the method m_w if the following conditions are satisfied:

1. there are not store $\mathsf{l}_{q-j+\pi}$ before any putfield $\kappa.f$:t and call $m_1 \dots m_k$ in m_w;
2. at the program point of every putfield $\kappa.f$:t in m_w, the pair $\langle l_{q-j+\pi}, s_{j-2}\rangle$ is not inside the possible reachability set $\mathcal{MR}_{\tau_*}$ related to that point;
3. at the program point of every call $m_1 \dots m_k$ in m_w, $l_{q-j+\pi}$ does not share with any actual parameter $s_k, \dots, s_{k-\phi}$ of that method.

Regarding the tokens coming from I_2 we exploit them to add new unreachability and non-cyclicity information that are available at the end of the method m_w. We add new unreachability tokens $a \not\rightsquigarrow^F b$ if the following conditions are satisfied:

1. a may reach b after the method execution (contrariwise we relapse in the first set): $\langle a, b\rangle \in \mathcal{MR}_{\tau'}$;
2. a and b must be alias of at least a couple of actual parameters s_{p_a}, s_{p_b} respectively: $(a, s_{p_a}), (b, s_{p_b}) \in \mathcal{DA}_\tau$;

3. these two actual parameters, whose corresponding formal parameters $l_{p_a-j+\pi}$ and $l_{p_b-j+\pi}$, are not re-assigned inside m_w through a store $\mathsf{l}_{\mathsf{pa-j+}\pi}$ or store $\mathsf{l}_{\mathsf{pb-j+}\pi}$.

If these conditions are satisfied the unreachability information between $l_{p_a-j+\pi}$ and $l_{p_b-j+\pi}$ in I_2 (i.e., at the end of the callee method m_w) corresponds to the unreachability information of a and b in the caller: $I_2 \ni l_{p_a-j+\pi} \not\rightsquigarrow^F l_{p_b-j+\pi} = a \not\rightsquigarrow^F b$.

About the non-cyclicity tokens in I_2, we add the new information in a way similar to the unreachability one but considering only one variables both in caller and in callee method.

Example 6.5. *Consider, for instance, nodes A and* 13 *in Figure 6.1. The approximation information of these nodes, as already presented above, is:*

$$I_A = \Big\{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\},$$
$$s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}, s_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}, s_3 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}\Big\} \cup$$
$$\Big\{\{l_0, l_1, s_0, s_1, s_2, s_3\} \rightsquigarrow \not\Diamond^{\mathcal{F}}\Big\}$$

and

$$I_{13} = \Big\{l_0 \not\rightsquigarrow^{\mathcal{F}} l_0, l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, l_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} l_2, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\},$$
$$l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_1, l_2\}, l_2 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.prec\}} l_0, l_1 \rightsquigarrow \not\Diamond^{\mathcal{F}}, \{l_0, l_2\} \rightsquigarrow \not\Diamond^{\{\mathsf{El}.value\}}\Big\}.$$

Consider the side-effect multi-arc linking node A and 13 *with node B; we illustrate the application of rule #11 on the presence of the sharing, reachability and aliasing information* $\mathcal{MS}_{\tau_S}, \mathcal{MR}_{\tau_A}, \mathcal{MR}_{\tau_B}, \mathcal{DA}_{\tau_A}$. *We remember that there are* $\pi = 3$ *parameters: the implicit* this *and two parameters of type* Object *and* Element. *Since the return type of the constructor is* void *we obtain* $i_B = 3$ *while* $j_B = 1$, *and for each variable* $v \in \mathsf{dom}(\tau_B) = \{l_0, l_1, l_2, s_0\}$, $\tau_B(v) = \tau_A(v)$. *By application of propagation rule #11 we obtain:*

$$I_B = \Big\{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1\}, l_1 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.prec\}} s_0, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_0\},$$
$$s_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} l_1, l_0 \rightsquigarrow \not\Diamond^{\mathcal{F}}, \{l_1, s_0\} \rightsquigarrow \not\Diamond^{\{\mathsf{El}.value\}}\Big\}$$

First of all we have $max = j_A - \pi = 4 - 3 = 1$ *i.e., regarding the stack elements, we have only to take into account the information about* s_0. *About the tokens that remains unchanged after the execution of the method i.e., they are from one (or more*[5]*) of the first three set, we have:*

$$\left\{\begin{array}{r} l_1 \not\rightsquigarrow^{\mathcal{F}} l_0, s_0 \not\rightsquigarrow^{\mathcal{F}} l_0, \\ l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_1, s_0\} \end{array}\right\} \subseteq \Big\{a \not\rightsquigarrow^{\mathcal{F}} b \,\Big|\, a, b \in \mathsf{dom}(\tau') \wedge \langle a, b\rangle \notin \mathcal{MR}_{\tau_B}\Big\}$$

$$\text{with } \mathcal{MR}_{\tau_B} = \{\langle s_0, l_1\rangle, \langle l_1, s_0\rangle, \langle l_0, l_0\rangle, \langle l_1, l_1\rangle, \langle s_0, s_0\rangle\}$$

$$\{l_0 \not\rightsquigarrow^{\mathcal{F}} l_0, l_0 \rightsquigarrow \not\Diamond^{\mathcal{F}}\} \subseteq \left\{\begin{array}{l|l} a \not\rightsquigarrow^{F_1} b, & a, b \in \{l_0, l_1, s_0\} \wedge \\ a \rightsquigarrow \not\Diamond^{F_2} \in I_A & \forall 1 \le p < 4, (a, s_p) \notin \mathcal{MS}_{\tau_A} \end{array}\right\}$$

$$\text{with } \mathcal{MS}_{\tau_A} = \{(s_0, s_1), (l_1, s_3), (l_0, l_0), (l_1, l_1), (s_0, s_0), (s_1, s_1), (s_2, s_2), (s_3, s_3)\}$$

[5]Regarding the first 3 sets of Propagation Rule #11 we note that they might assert unreachability and non-cyclicity properties to the same variables' pairs (or, respectively, variables) but with respect to a different set of fields. Fortunately it does not matter since these sets are all monotonous and then, when we apply the normalize operator, the token with the maximal set becomes the reference one.

Instead of, about the unreachability and non-cyclicity tokens that derive from the information available at the end of method m_w, we have:

$$\left\{\begin{array}{l} l_1 \not\rightsquigarrow^{\mathcal{F}} l_1, s_0 \not\rightsquigarrow^{\mathcal{F}} s_0, \\ l_1 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.prec\}} s_0, \\ s_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.prec\}} l_1 \end{array}\right\} \subseteq \left\{ a \not\rightsquigarrow^{F} b \;\middle|\; \begin{array}{l} 1.\ a, b \in \{l_0, l_1, s_0\} \wedge \langle a, b\rangle \in \mathcal{MR}_{\tau_B} \\ 2.\ \exists 1 \le p_a, p_b < 4 \,\middle|\, (a, s_{p_a}), (b, s_{p_b}) \in \mathcal{DA}_{\tau_A} \wedge \\ \quad \left[no\ \boxed{\mathsf{store}\ \mathsf{l}_{\mathsf{pa}-\mathsf{j}+\pi}}\ nor\ \boxed{\mathsf{store}\ \mathsf{l}_{\mathsf{p_b}-\mathsf{j}+\pi}}\ occurs\ in\ m_w \right] \\ 3.\ l_{p_a-j+\pi} \not\rightsquigarrow^{F} l_{p_b-j+\pi} \in I_{13} \end{array} \right\}$$

$$with\ \mathcal{DA}_{\tau_A} = \{(s_0, s_1), (l_1, s_3), (l_0, l_0), (l_1, l_1), (s_0, s_0), (s_1, s_1), (s_2, s_2), (s_3, s_3)\}$$

and

$$\left\{\{l_1, s_0\} \rightsquigarrow \circlearrowleft\!\!\!\!/^{\{\mathsf{El}.value\}}\right\} \subseteq \left\{ c \rightsquigarrow \circlearrowleft\!\!\!\!/^{F} \;\middle|\; \begin{array}{l} 1.\ c \in \{l_0, l_1, s_0\} \\ 2.\ \exists 1 \le p < 4 \,\middle|\, (c, s_p) \in \mathcal{DA}_{\tau_A} \wedge \\ \quad \left[no\ \boxed{\mathsf{store}\ \mathsf{l}_{\mathsf{p}-\mathsf{j}+\pi}}\ occurs\ in\ m_w \right] \\ 3.\ l_{p-j+\pi} \rightsquigarrow \circlearrowleft\!\!\!\!/^{F} \in I_{13} \end{array} \right\}$$

Return value multi-arc links the nodes $\boxed{\mathsf{call}\ m_1 \dots m_k}$ and $\boxed{\mathsf{exit}@m_w}$ to the next instruction after the non-exceptional ending of the method m_w only if it returns a value. The rule **#12** extends $\Pi^{\#11}$, since also in this case in the caller we have side-effects due to the method execution, but now we have also to handle the unreachability and non-cyclicity tokens related to the returned value $s_{j-\pi}$. The returned value in the callee corresponds to the stack variable s_0 and hence its information at node $\boxed{\mathsf{exit}@m_w}$.

First of all the non-cyclicity information of $s_{j-\pi}$ in the caller are the same as s_0 at the end of the callee. Furthermore the rule state that also the unreachability information between $s_{j-\pi}$ and itself are the same of s_0 and itself in the callee.

Regarding the other variables presented after the call and different from the returned value i.e., $a \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\}$, if they does not reach $s_{j-\pi}$ after the call we can add the triples such that $a \not\rightsquigarrow^{\mathcal{F}} s_{j-\pi}$, as shown in the second set. Contrariwise if they does not be reached by $s_{j-\pi}$ we can add the triples such that $s_{j-\pi} \not\rightsquigarrow^{\mathcal{F}} a$, as shown in the fourth set.

Instead of, regarding the variables that may reach $s_{j-\pi}$ after the method execution i.e., $\langle a, s_{j-\pi}\rangle \in \mathcal{MR}_{\tau'}$, we add unreachability tokens if the following conditions are satisfied:

1. a must be alias of at least an actual parameter s_{p_a}: $(a, s_{p_a}) \in \mathcal{DA}_{\tau}$;
2. this actual parameter, whose corresponding formal parameter $l_{p_a-j+\pi}$, is not reassigned inside m_w through a $\mathsf{store}\ \mathsf{l}_{\mathsf{pa}-\mathsf{j}+\pi}$.

If these conditions are satisfied the unreachability information between $l_{p_a-j+\pi}$ and s_0 in I_2 i.e., at the end of the callee method m_w, corresponds to the unreachability information of a and $s_{j-\pi}$ in the caller: $I_2 \ni l_{p_a-j+\pi} \not\rightsquigarrow^{F} s_0 = a \not\rightsquigarrow^{F} s_{j-\pi}$.

On the other hand, regarding the variables that may be reached by $s_{j-\pi}$ after the method execution i.e., $\langle s_{j-\pi}, b\rangle \in \mathcal{MR}_{\tau'}$, we add unreachability tokens if the following conditions are satisfied:

1. b must be alias of at least an actual parameter s_{p_b}: $(s_{p_b}, b) \in \mathcal{DA}_{\tau}$;

2. this actual parameter, whose corresponding formal parameter $l_{p_b-j+\pi}$, is not re-assigned inside m_w through a store $\mathsf{l}_{\mathsf{p_b-j+\pi}}$.

If these conditions are satisfied the unreachability information between s_0 and $l_{p_b-j+\pi}$ in I_2 i.e., at the end of the callee method m_w, corresponds to the unreachability information of $s_{j-\pi}$ and b in the caller: $I_2 \ni s_0 \not\rightsquigarrow^F l_{p_b-j+\pi} = s_{j-\pi} \not\rightsquigarrow^F b$.

Exceptional arcs link every instruction that might throw an exception to the catch at the beginning of their exception handler(s).

Rule **#13** is built to be identical to #9 since it deals with the same source node, $\boxed{\mathsf{throw}\,\kappa}$, but it is applied in the case of an exceptional execution.

Rule **#14** deals with all other bytecode instructions that might throw an exception (new κ, getfield $\kappa.f : \mathsf{t}$, putfield $\kappa.f : \mathsf{t}$): it states that the stack disappears but the unreachability among local variables remains unaffected. Moreover a new stack element containing a new fresh variable s_0 of type $\kappa <$ `Throwable` is created. It means that this new variable, s_0, is only reachable from itself and hence, in addition to the information provided by the previous node ($\lambda I.I$), we have to add new unreachability information w.r.t. all the program fields between s_0 and all the other variables in L: $\left\{a \not\rightsquigarrow^{\mathcal{F}} s_0, s_0 \not\rightsquigarrow^{\mathcal{F}} a \,|\, a \in L\right\}$.

Finally, we add the non-cyclicity information of the new top of the stack, $s_0 \rightsquigarrow^{\not\circlearrowleft_{\mathcal{F}}}$, and also the unreachability one between this new variable and itself, $s_0 \not\rightsquigarrow^{\mathcal{F}} s_0$; in both cases the set of fields is $\mathcal{F}$ because s_0 is fresh and, therefore, it is isolated from the rest of the heap.

Rule **#15** states a pessimistic assumption about the exceptional states after a method call. The peculiarity of this rule is the handling of unreachability and non-cyclicity tokens about the new variable s_0 of type $\kappa <$ `Throwable`: we deal with the variables s_0 in rule #14 with the difference that we check that the other variables may not reach and be reached from s_0. That is because this variable, bound to an exception object, may reach or be reached by another exception.

Example 6.6. *Consider nodes* 2*,* 5*,* 8*,* 11 *and* 14 *in Figure 6.1. We assume that the approximation information of the first three node are:*

$$I_2 = \Big\{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},$$
$$\{l_0, l_1, l_2, s_0\} \rightsquigarrow^{\not\circlearrowleft_{\mathcal{F}}}\Big\}$$

$$I_5 = \Big\{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},$$
$$s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, \{l_0, l_1, l_2, s_0, s_1\} \rightsquigarrow^{\not\circlearrowleft_{\mathcal{F}}}\Big\}$$

$$I_8 = \Big\{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_2, s_0, s_1\}, l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},$$
$$s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_2, s_0\}, s_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, \{l_0, l_1, l_2, s_0, s_1\} \rightsquigarrow^{\not\circlearrowleft_{\mathcal{F}}}\Big\}$$

$$I_{11} = \Big\{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_1\} l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, l_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} \{l_2, s_0\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},$$
$$l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_1\}, s_1 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1,$$
$$s_1 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} \{l_2, s_0\}, \{l_0, l_1, l_2, s_0, s_1\} \rightsquigarrow^{\not\circlearrowleft_{\mathcal{F}}}\Big\}$$

These three nodes are all linked with node 14 *with an exceptional arc with the same propagation rule #14. Thus the node* 14 *has in-degree greater than 1 and hence to establish the set* $I_1 4$ *we have to compute the least upper bound of the sets obtained by applying the normalized propagation rules of these arcs i.e.,*

$$I_{14} \subseteq \Phi^{\#14}(I_2) \sqcup_{\mathsf{A}} \Phi^{\#14}(I_5) \sqcup_{\mathsf{A}} \Phi^{\#14}(I_8) \sqcup_{\mathsf{A}} \Phi^{\#14}(I_{11}) = \\ \Phi^{\#14}(I_2) \cap \Phi^{\#14}(I_5) \cap \Phi^{\#14}(I_8) \cap \Phi^{\#14}(I_{11})$$

The resulting (not normalized) set is:

$$I_{14} = \Big\{ l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_0\} l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1,\ l_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} l_2,\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, \\ l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},\ s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\}, \{l_0, l_1, l_2, s_0\} \rightsquigarrow \not\circlearrowright^{\mathcal{F}} \Big\}$$

We note that this set is exactly $I_{14} \subseteq \Pi^{\#14}(I_{11})$*, since* I_{11} *is, among the sets of the three* putfield $\kappa.f$: t *nodes, the one with the least number of tokens concerning the local variables.*

We are now able to give the maximal solution of the ACG w.r.t. the partial order $\supseteq$. The rows highlighted represent the abstraction sets of nodes following another one representing the putfield $\kappa.f$: t instruction.

Table 6.8: Solution of the Abstract Contraint Graph in Figure 6.1

Node n	Solution of Approximation I_n
A	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\},\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\},\ s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\},$ $s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\},\ s_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\},\ s_3 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\},$ $\{l_0, l_1, l_2, s_0, s_1, s_2, s_3\} \rightsquigarrow \not\circlearrowright^{\mathcal{F}}$
1	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\},\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\},\ l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, \{l_0, l_1, l_2\} \rightsquigarrow \not\circlearrowright^{\mathcal{F}}$
2	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},\ l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},\ s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},$ $\{l_0, l_1, l_2, s_0\} \rightsquigarrow \not\circlearrowright^{\mathcal{F}}$
3	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\},\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\},\ l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, \{l_0, l_1, l_2\} \rightsquigarrow \not\circlearrowright^{\mathcal{F}}$
4	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},\ l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},\ s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},$ $\{l_0, l_1, l_2, s_0\} \rightsquigarrow \not\circlearrowright^{\mathcal{F}}$
5	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},\ l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},$ $s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},\ s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, \{l_0, l_1, l_2, s_0, s_1\} \rightsquigarrow \not\circlearrowright^{\mathcal{F}}$
6	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_2\},\ l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1,\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\},\ l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, \{l_0, l_1, l_2\} \rightsquigarrow \not\circlearrowright^{\mathcal{F}}$
7	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_2, s_0\},\ l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1,\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},\ l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\},$ $s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_2, s_0\},\ s_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, \{l_0, l_1, l_2, s_0\} \rightsquigarrow \not\circlearrowright^{\mathcal{F}}$
8	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_2, s_0, s_1\},\ l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1,\ l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},\ l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},$ $s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_2, s_0\},\ s_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1,\ s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, \{l_0, l_1, l_2, s_0, s_1\} \rightsquigarrow \not\circlearrowright^{\mathcal{F}}$

Continue in the following page

Continue from previous page

Node n	Solution of Approximation I_n
9	$l_0 \not\rightsquigarrow^{\mathcal{F}} l_0, l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, l_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} l_2, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, \{l_0, l_1, l_2\} \rightsquigarrow^{\not\Delta\mathcal{F}}$
10	$l_0 \not\rightsquigarrow^{\mathcal{F}} l_0, l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, l_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} \{l_2, s_0\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\}, \{l_0, l_1, l_2, s_0\} \rightsquigarrow^{\not\Delta\mathcal{F}}$
11	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_1\} l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, l_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} \{l_2, s_0\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_1\}, s_1 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, s_1 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} \{l_2, s_0\}, \{l_0, l_1, l_2, s_0, s_1\} \rightsquigarrow^{\not\Delta\mathcal{F}}$
12	$l_0 \not\rightsquigarrow^{\mathcal{F}} l_0, l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, l_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} l_2, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\},$
13	$l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_1, l_2\}, l_2 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.prec\}} l_0, l_1 \rightsquigarrow^{\not\Delta\mathcal{F}}, \{l_0, l_2\} \rightsquigarrow^{\not\Delta\{\mathsf{El}.value\}}$
14	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_0\} l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, l_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} l_2, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},$
15	$l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0\}, \{l_0, l_1, l_2, s_0\} \rightsquigarrow^{\not\Delta\mathcal{F}}$
16	
B	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1\}, l_1 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.prec\}} s_0, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_0\}, s_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} l_1, l_0 \rightsquigarrow^{\not\Delta\mathcal{F}}, \{l_1, s_0\} \rightsquigarrow^{\not\Delta\{\mathsf{El}.value\}}$
C	$l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0\}, \{l_0, l_1, s_0\} \rightsquigarrow^{\not\Delta\mathcal{F}}$

Chapter 7

Soundness

We are going to prove that the analysis presented in Chapter 6 correctly approximates the behavior of the program with respect to the two properties defined in Chapter 4. Nevertheless, the proof of our analysis requires some preliminary results.

7.1 Preliminary Results

The following lemma states some sufficient conditions that, whenever hold, ensure, for two different states σ and σ', the unchanging of the current information that we want to assert. It is strictly related to a lemma provided by Nikolić and Spoto [13], but obviously we have adapt it to our properties. In fact, it will be useful to prove most of the cases related to our rules, since it will be enough to attest that its four conditions hold.

Lemma 7.1. *Let* $\tau, \tau' \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ *and* $\sigma' = \langle \rho', \mu' \rangle \in \Sigma_{\tau'}$. *Let* $a, b \in \mathsf{dom}(\tau)$ *and* $a', b' \in \mathsf{dom}(\tau')$ *be such that*

1. $\rho(a) = \rho'(a')$
2. $\rho(b) = \rho'(b')$
3. $\mathsf{dom}(\mu) \subseteq \mathsf{dom}(\mu')$
4. *for all* $\ell \in \mathsf{L}_\sigma(a)$ *we have* $\mu(\ell) = \mu'(\ell)$

Then, $a \not\rightsquigarrow^F_\sigma b \iff a' \not\rightsquigarrow^F_{\sigma'} b'$ *and* $a \rightsquigarrow^{\not\Delta F}_\sigma \iff a' \rightsquigarrow^{\not\Delta F}_{\sigma'}$.

Proof. Both the Definition 4.7 about Unreachability and 4.8 about Non-Cyclicity, are based on the concept of Path between Locations defined in 4.4. By path definition and by Lemma 4.2, there exists a path $\mathcal{P}$ between variables a and b iff $\rho(b) = \mathsf{L}_\sigma(a)$ and therefore that path goes through only locations in $\mathsf{L}_\sigma(a)$. Hence, since $\rho(b) = \rho'(b')$ by hypothesis, it is enough to prove $\mathsf{L}_\sigma(a) = \mathsf{L}_{\sigma'}(a')$: if the locations reachable from $\rho(a) = \rho'(a')$ do not change, also all paths starting from $\rho(a) = \rho'(a')$ remain unchanged and thus also the unreachability and non-cyclicity information.

If $\rho(a) = \rho'(a') \notin \mathbb{L}$ then $\mathsf{L}_\sigma(a) = \mathsf{L}_{\sigma'}(a') = \emptyset$ (Definition 4.2) and the thesis is trivially true. Assuming $\rho(a) = \rho'(a') \in \mathbb{L}$, we prove that $\mathsf{L}^i_\mu(\rho(a)) = \mathsf{L}^i_{\mu'}(\rho'(a'))$, by induction on i.

i) `Base case:` i=0; we have $\mathsf{L}^0_\mu(\rho(a)) \doteq \rho(a) = \rho'(a') \doteq \mathsf{L}^0_{\mu'}(\rho'(a'))$.

ii) `Inductive step:` we assume that $\forall i > 0$, $\mathsf{L}_{\mu}^{i-1}(\rho(a)) = \mathsf{L}_{\mu'}^{i-1}(\rho'(a'))$. Thus we have:

$$\begin{aligned}
\mathsf{L}_{\mu}^{i}(\rho(a)) &\overset{\text{Def. 4.1}}{=} \mathsf{L}_{\mu}^{i-1}(\rho(a)) \cup \bigcup_{\ell \in \mathsf{L}_{\mu}^{i-1}(\rho(a))} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) \\
&\overset{\text{Ind. Hyp.}}{=} \mathsf{L}_{\mu'}^{i-1}(\rho'(a')) \cup \bigcup_{\ell \in \mathsf{L}_{\mu'}^{i-1}(\rho'(a'))} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) \\
&\overset{\text{points (3) and (4)}}{=} \mathsf{L}_{\mu}^{i-1}(\rho'(a')) \cup \bigcup_{\ell \in \mathsf{L}_{\mu}^{i-1}(\rho'(a'))} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \\
&\overset{\text{Def. 4.1}}{=} \mathsf{L}_{\mu'}^{i}(\rho'(a')).
\end{aligned}$$

□

Note that point 3 and 4 of lemma 7.1 hold, in particular, either when $\mu = \mu'$ or when $\mathsf{dom}(\mu) \subseteq \mathsf{dom}(\mu')$ with $\mathsf{L}_{\sigma}(v) = \mathsf{L}_{\sigma'}(v)$ for some $v \in \mathsf{dom}(\tau)$ and $v \in \mathsf{dom}(\tau')$.

Lemma 7.2. *Let $\tau \in \mathcal{T}$, $\sigma \in \Sigma_{\tau}$, $a \in \mathsf{dom}(\tau)$ and $F = \{\kappa_k.f_k : \mathsf{t}_k \in \mathcal{F} \mid \mathsf{t}_k \notin \mathsf{T}(\tau(a))\}$. Then for each $\ell \in \mathsf{L}_{\sigma}(a)$ and for each path $\mathcal{P} \subseteq \mathcal{F}$ such that $\rho(a) \rightsquigarrow_{\sigma}^{\mathcal{P}} \ell$, it does not exist $\kappa_k.f_k : \mathsf{t}_k \in F$ which is also inside $\mathcal{P}$.*

Proof. We are going to prove that every field in every path $\mathcal{P}$ that satisfies our hypotheses is inside $\mathsf{T}(\tau(a))$ and, therefore, this entails our thesis. If $\mathcal{P} \subseteq \mathcal{F}$ is such that $\rho(a) \rightsquigarrow_{\sigma}^{\mathcal{P}} \ell$, for each $\kappa_i.f_i : \mathsf{t}_i \in \mathcal{P}$, by Definition 4.4, it has to be $\mathsf{t}_i = \mu(\ell^{i+1}).\kappa \in \mathbb{T}$. Then, by Definition 4.6, we have $\mathsf{t}_i \in \mathsf{T}(\tau(a))$. □

7.2 Soundness of Propagation Rules

We are going to prove that the *normalized propagation rules* of Definition 6.3 (that refers to the Propagation Rules in Definition 6.2) are correct.

7.2.1 Soundness of Sequential Arcs

Lemma 7.3 (Sequential Arcs)**.** *The normalized propagation rules for* sequential arcs *of Definition 5.3 are correct. That it, consider a sequential arc from a bytecode* ins *and its normalized propagation rule* Φ*. Assume that* ins *has* static type *information τ at its beginning and τ' immediately after its non-exceptional execution. Then, for every $I \in \mathsf{A}_{\tau}$ we have*

$$\mathit{ins}\,(\gamma_{\tau}(I)) \cap \Xi_{\tau'} \subseteq \gamma_{\tau'}(\Phi(I))$$

(we recall that ins *is the semantic of* ins *bytecode).*

Proof. Since $\gamma_{\tau'}(\Pi(I)) = \gamma_{\tau'}(\Phi(I))$ by Corollary 6.1.1, it is enough to prove

$$\mathit{ins}\,(\gamma_{\tau}(I)) \cap \Xi_{\tau'} \subseteq \gamma_{\tau'}(\Pi(I)).$$

Let $\mathsf{dom}(\tau) = L \cup S$ contains i local variables $L = \{l_0, \ldots, l_{i-1}\}$ and j stack elements $S = \{s_0, \ldots, s_{j-1}\}$. Furthermore let $\mathsf{dom}(\tau') = L' \cup S'$, where L' and S' are the local and stack variables after the execution of ins. Consider an arbitrary abstract element $I \in \mathsf{A}_{\tau}$ and an

arbitrary state $\omega' = \langle \rho', \mu' \rangle \in \mathit{ins}(\gamma_\tau(I)) \cap \Xi_{\tau'}$. We prove that $\omega' \in \gamma_{\tau'}(\Pi(I))$ i.e., by definition 5.2 that

$$\text{for each } x \not\rightsquigarrow^F y \in \Pi(I) \text{ we have } x \not\rightsquigarrow_{\omega'}^{F'} y \wedge F \subseteq F' \tag{7.1}$$

$$\text{for each } z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in \Pi(I) \text{ we have } z \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega'}^{F'} \wedge F \subseteq F' \tag{7.2}$$

Note that, by the choice of ω', there exists $\omega = \langle \rho, \mu \rangle \in \gamma_\tau(I)$ such that $\omega' = \mathit{ins}(\omega)$ and, by Definition 5.2, for each $x \not\rightsquigarrow^F y \in I$ we obtain $x \not\rightsquigarrow_{\omega}^{F'} y$ with $F \subseteq F'$ and, respectively, for each $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I$ we obtain $z \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega}^{F'}$ with $F \subseteq F'$. We analyze different propagation rules corresponding to different types of sequential arcs.

ins = dup t. We have $L' = L$ and $S' = S \cup \{s_j\}$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_j) = \rho'(s_{j-1})$. By Definition 6.2, $\Pi(I) = I \cup I[s_{j-1} \mapsto s_j] \cup I_1$ where $I_1 = \left\{ s_{j-1} \not\rightsquigarrow^F s_j, s_j \not\rightsquigarrow^F s_{j-1} \,\middle|\, s_{j-1} \not\rightsquigarrow^F s_{j-1} \in I \right\}$. About expression (7.1), we distinguish the following cases:

- $\mathtt{x}, \mathtt{y} \neq \mathtt{s_j}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$ we have $x \not\rightsquigarrow^F y \in I$, since $\rho'(x) = \rho(x)$, $\rho'(y) = \rho(y)$ and $I \subseteq \Pi(I)$. By $x \not\rightsquigarrow^F y \in I$ we obtain $x \not\rightsquigarrow_{\omega}^{F'} y \wedge F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_{\omega}^{F'} y$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{s_j}, \mathtt{y} \neq \mathtt{s_j}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$, since $\rho'(x) = \rho(s_{j-1})$, it has to be $x \not\rightsquigarrow^F y \in I[s_{j-1} \mapsto s_j] \cup I_1 \subseteq \Pi(I)$. Then we have $\rho'(y) = \rho(y)$ and hence $s_{j-1} \not\rightsquigarrow^F y \in I$, both if $y = s_{j-1}$ or $y \neq s_{j-1}$. Therefore we obtain $s_{j-1} \not\rightsquigarrow_{\omega}^{F'} y \wedge F \subseteq F'$ and, by Lemma 7.1, $s_{j-1} \not\rightsquigarrow_{\omega}^{F'} y$ entails $a \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} \neq \mathtt{s_j}, \mathtt{y} = \mathtt{s_j}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$, since $\rho'(y) = \rho(s_{j-1})$, it has to be $x \not\rightsquigarrow^F y \in I[s_{j-1} \mapsto s_j] \cup I_1 \subseteq \Pi(I)$. Then we have $\rho'(x) = \rho(x)$ and hence $x \not\rightsquigarrow^F s_{j-1} \in I$, both if $x = s_{j-1}$ or $x \neq s_{j-1}$. Therefore we obtain $a \not\rightsquigarrow_{\omega}^{F'} s_{j-1} \wedge F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_{\omega}^{F'} s_{j-1}$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{y} = \mathtt{s_j}$: by $x \not\rightsquigarrow^F y \in \Pi(I)$, since $\rho'(x) = \rho'(y) = \rho(s_{j-1})$, it has to be $x \not\rightsquigarrow^F y \in I[s_{j-1} \mapsto s_j] \subseteq \Pi(I)$. Hence we have $s_{j-1} \not\rightsquigarrow^F s_{j-1} \in I$ and therefore we obtain $s_{j-1} \not\rightsquigarrow_{\omega}^{F'} s_{j-1} \wedge F \subseteq F'$ and, by Lemma 7.1, $s_{j-1} \not\rightsquigarrow_{\omega}^{F'} s_{j-1}$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.

Regarding expression (7.2), we distinguish the following cases:

- $\mathtt{z} \neq \mathtt{s_j}$: for each $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in \Pi(I)$ we have $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I$, since $\rho'(z) = \rho(z)$ and $I \subseteq \Pi(I)$. By $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I$ we obtain $z \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega}^{F'} \wedge F \subseteq F'$ and, by Lemma 7.1, $z \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega}^{F'}$ entails $z \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega'}^{F'}$.
- $\mathtt{z} = \mathtt{s_j}$: by $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in \Pi(I)$ then, since $\rho'(z) = \rho(s_{j-1})$, it has to be $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I[s_{j-1} \mapsto s_j] \subseteq \Pi(I)$. Hence we have $s_{j-1} \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I$ and therefore we obtain $s_{j-1} \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega}^{F'} \wedge F \subseteq F'$ and, by Lemma 7.1, $s_{j-1} \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega}^{F'}$ entails $z \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega'}^{F'}$.

ins = new κ, const v. We can prove these two instructions together.

About new κ instruction, we have $L' = L$ and $S' = S \cup \{s_j\}$. Furthermore, for every $v \in \mathsf{dom}(\tau') \setminus \{s_j\} = \mathsf{dom}(\tau)$, $\rho'(v) = \rho(v)$ and $\mu' = \mu[\ell \mapsto o]$, where o is a new object of class κ. Whereas $\rho'(s_j) = \ell \in \mathbb{L}$, where ℓ is a fresh location, is only reachable from itself i.e., for every $a \in \mathsf{dom}(\tau)$ we have $\rho(a) \notin \mathsf{L}_\omega(s_j)$ and, therefore, it cannot either reach any cycle.

About const v instruction, we have $L' = L$, $S' = S \cup \{s_j\}$, $\mu = \mu'$, $\rho'(s_j) = v \in \mathbb{Z}$ and for every $v \in \mathsf{dom}(\tau') \setminus \{s_j\} = \mathsf{dom}(\tau)$, $\rho'(v) = \rho(v)$. Hence $\mathsf{L}_{\omega'}(s_j) = \emptyset$ i.e., for every $a \in \mathsf{dom}(\tau)$ we have $\rho(a) \notin \mathsf{L}_\omega(s_j)$ and, therefore, it cannot either reach any cycle.

In both case By Definition 6.2,

$$\Pi(I) = I \cup \overbrace{\left\{ s_j \not\rightsquigarrow^{\mathcal{F}} s_j, s_j \rightsquigarrow^{\not\Delta_{\mathcal{F}}} \right\}}^{I_1} \cup \overbrace{\left\{ a \not\rightsquigarrow^{\mathcal{F}} s_j, s_j \not\rightsquigarrow^{\mathcal{F}} a \,|\, a \in dom(\tau) \right\}}^{I_2}.$$

About expression (7.1) we distinguish the following cases:

- $\mathtt{x}, \mathtt{y} \neq \mathtt{s_j}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$ we have $x \not\rightsquigarrow^F y \in I$, since $\rho'(x) = \rho(x)$ and $\rho'(x) = \rho(y)$ and $I \subseteq \Pi(I)$. By $x \not\rightsquigarrow^F y \in I$ we obtain $x \not\rightsquigarrow_\omega^{F'} y \wedge F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_\omega^{F'} y$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{s_j}, \mathtt{y} \neq \mathtt{s_j}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$ we have $x \not\rightsquigarrow^F y \in I_2$, since $\rho'(y) = \rho(y)$ and $y \in \mathsf{dom}(\tau')$. Furthermore, since $\rho'(x)$ is fresh ($\mathsf{new}\ \kappa$) or an integer ($\mathsf{const}\ v$), $\rho(y) \notin \mathsf{L}_\omega(x)$ and therefore we can assert $x \not\rightsquigarrow_{\omega'}^{F'} y$ for every $F' \subseteq \mathcal{F}$.
- $\mathtt{x} \neq \mathtt{s_j}, \mathtt{y} = \mathtt{s_j}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$ we have $x \not\rightsquigarrow^F y \in I_2$, since $\rho'(x) = \rho(x)$ and $x \in \mathsf{dom}(\tau')$. Furthermore, since $\rho'(y)$ is fresh ($\mathsf{new}\ \kappa$) or an integer ($\mathsf{const}\ v$), $\rho(y) \notin \mathsf{L}_\omega(x)$ and therefore we can assert $x \not\rightsquigarrow_{\omega'}^{F'} y$ for every $F' \subseteq \mathcal{F}$.
- $\mathtt{x} = \mathtt{y} = \mathtt{s_j}$: if $x \not\rightsquigarrow^F y \in \Pi(I)$, it has to be $x \not\rightsquigarrow^F y = s_j \not\rightsquigarrow^{\mathcal{F}} s_j \in I_1$. Hence, since $\rho(x) = \rho(y)$ is fresh ($\mathsf{new}\ \kappa$) or an integer ($\mathsf{const}\ v$), we have either $\rho(y) = \mathsf{L}_{\omega'}(x)$ or $\mathsf{L}_{\omega'}(x) = \emptyset$ respectively and hence, by Definition 4.7, $x \not\rightsquigarrow_{\omega'}^{F} y$ with $F \subseteq \mathcal{F}$.

Regarding expression (7.2), we distinguish the following cases:

- $\mathtt{z} \neq \mathtt{s_j}$: for each $z \rightsquigarrow^{\not\Delta_F} \in \Pi(I)$ we have $z \rightsquigarrow^{\not\Delta_F} \in I$, since $\rho'(z) = \rho(z)$ and $I \subseteq \Pi(I)$. By $z \rightsquigarrow^{\not\Delta_F} \in I$ we obtain $z \rightsquigarrow_\omega^{\not\Delta_{F'}} \wedge F \subseteq F'$ and, by Lemma 7.1, $z \rightsquigarrow_\omega^{\not\Delta_{F'}}$ entails $z \rightsquigarrow_{\omega'}^{\not\Delta_{F'}}$.
- $\mathtt{z} = \mathtt{s_j}$: by $z \rightsquigarrow^{\not\Delta_F} \in \Pi(I)$, it has to be $z \rightsquigarrow^{\not\Delta_F} = s_j \rightsquigarrow^{\not\Delta_{\mathcal{F}}} \in I_1$. Hence, since $\rho'(c)$ is fresh ($\mathsf{new}\ \kappa$) or an integer ($\mathsf{const}\ v$), it cannot reach any cycle. Therefore $z \rightsquigarrow_\omega^{\not\Delta_F}$ holds for every $F \subseteq \mathcal{F}$.

ins = load k t. We have $L' = L$ and $S' = S \cup \{s_j\}$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_j) = \rho(l_k)$. By definition 6.2, $\Pi(I) = I \cup I[l_k \mapsto s_j] \cup I_1$ where $I_1 = \left\{ l_k \not\rightsquigarrow^F s_j, s_j \not\rightsquigarrow^F l_k \,\middle|\, l_k \not\rightsquigarrow^F l_k \in I \right\}$. About expression (7.1), we distinguish the following cases:

- $\mathtt{x}, \mathtt{y} \neq \mathtt{s_j}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$ we have $x \not\rightsquigarrow^F y \in I$, since $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$ and $I \subseteq \Pi(I)$. By $x \not\rightsquigarrow^F y \in I$ we obtain $x \not\rightsquigarrow_\omega^{F'} y \wedge F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_\omega^{F'} y$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{s_j}, \mathtt{y} \neq \mathtt{s_j}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$, since $\rho'(x) = \rho(l_k)$ and $\rho'(y) = \rho(y)$, it has to be $x \not\rightsquigarrow^F y \in I[l_k \mapsto s_j] \cup I_1 \subseteq \Pi(I)$. Then we have $l_k \not\rightsquigarrow^F y \in I$ and therefore we obtain $l_k \not\rightsquigarrow_\omega^{F'} y \wedge F \subseteq F'$ and, by Lemma 7.1, $l_k \not\rightsquigarrow_\omega^{F'} y$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} \neq \mathtt{s_j}, \mathtt{y} = \mathtt{s_j}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$, since $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(l_k)$, it has to be $x \not\rightsquigarrow^F y \in I[l_k \mapsto s_j] \cup I_1 \subseteq \Pi(I)$. Then we have $x \not\rightsquigarrow^F l_k \in I$ and therefore we obtain $x \not\rightsquigarrow_\omega^{F'} l_k \wedge F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_\omega^{F'} l_k$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{y} = \mathtt{s_j}$: by $x \not\rightsquigarrow^F y \in \Pi(I)$, since $\rho'(x) = \rho'(y) = \rho(l_k)$, it has to be $x \not\rightsquigarrow^F y \in I[l_k \mapsto s_j] \subseteq \Pi(I)$. Hence we have $l_k \not\rightsquigarrow^F l_k \in I$ and therefore we obtain $l_k \not\rightsquigarrow_\omega^{F'} l_k \wedge F \subseteq F'$ and, by Lemma 7.1, $l_k \not\rightsquigarrow_\omega^{F'} l_k$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.

Regarding expression (7.2), we distinguish the following cases:

- $\mathtt{z} \neq \mathtt{s_j}$: for each $z \rightsquigarrow^{\not\Delta F} \in \Pi(I)$ we have $z \rightsquigarrow^{\not\Delta F} \in I$, since $\rho'(z) = \rho(z)$ and $I \subseteq \Pi(I)$. By $z \rightsquigarrow^{\not\Delta F} \in I$ we obtain $z \rightsquigarrow_{\omega}^{\not\Delta F'} \wedge F \subseteq F'$ and, by Lemma 7.1, $z \rightsquigarrow_{\omega}^{\not\Delta F'}$ entails $z \rightsquigarrow_{\omega'}^{\not\Delta F'}$.
- $\mathtt{z} = \mathtt{s_j}$: by $z \rightsquigarrow^{\not\Delta F} \in \Pi(I)$ then, since $\rho'(z) = \rho(l_k)$, it has to be $z \rightsquigarrow^{\not\Delta F} \in I[l_k \mapsto s_j] \subseteq \Pi(I)$. Hence we have $l_k \rightsquigarrow^{\not\Delta F} \in I$ and therefore we obtain $l_k \rightsquigarrow_{\omega}^{\not\Delta F'} \wedge F \subseteq F'$ and, by Lemma 7.1, $l_k \rightsquigarrow_{\omega}^{\not\Delta F'}$ entails $z \rightsquigarrow_{\omega'}^{\not\Delta F'}$.

ins = store k t. We have $L' = L$ and $S' = S \setminus \{s_{j-1}\}$, $\mu = \mu'$ and for all $v \in \mathsf{dom}(\tau') \setminus \{l_k\}$ we have $\rho'(v) = \rho(v)$ while $\rho'(l_k) = \rho(s_{j-1})$. By Definition 6.2,

$$\Pi(I) = \overbrace{\left\{\left(a \not\rightsquigarrow^{F} b\right)\left[s_{j-1} \mapsto l_k\right] \middle| a \not\rightsquigarrow^{F} b \in I \wedge a, b \neq l_k\right\}}^{I_1} \cup \underbrace{\left\{\left(a \rightsquigarrow^{\not\Delta F}\right)\left[s_{j-1} \mapsto l_k\right] \middle| a \rightsquigarrow^{\not\Delta F} \in I \wedge a \neq l_k\right\}}_{I_2}.$$

About expression (7.1), we distinguish the following cases:

- $\mathtt{x}, \mathtt{y} \neq \mathtt{l_k}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$ we have $x \not\rightsquigarrow^{F} y \in I$, since $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$ and hence the substitution $[s_{j-1} \mapsto l_k]$ did not affect these two variables. By $x \not\rightsquigarrow^{F} y \in I$ we obtain $x \not\rightsquigarrow_{\omega}^{F'} y \wedge F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_{\omega}^{F'} y$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{l_k}, \mathtt{y} \neq \mathtt{l_k}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$, since $\rho'(x) = \rho(s_{j-1})$ and $\rho'(y) = \rho(y)$, it has to be $x \not\rightsquigarrow^{F} y = \left(s_{j-1} \not\rightsquigarrow^{F} y\right)[s_{j-1} \mapsto l_k] \in I_1$. Then we have $s_{j-1} \not\rightsquigarrow^{F} y \in I$ and therefore we obtain $s_{j-1} \not\rightsquigarrow_{\omega}^{F'} y \wedge F \subseteq F'$ and, by Lemma 7.1, $s_{j-1} \not\rightsquigarrow_{\omega}^{F'} y$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} \neq \mathtt{l_k}, \mathtt{y} = \mathtt{l_k}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$, since $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(s_{j-1})$, it has to be $x \not\rightsquigarrow^{F} y = \left(x \not\rightsquigarrow^{F} s_{j-1}\right)[s_{j-1} \mapsto l_k] \in I_1$. Then we have $x \not\rightsquigarrow^{F} s_{j-1} \in I$ and therefore we obtain $x \not\rightsquigarrow_{\omega}^{F'} s_{j-1} \wedge F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_{\omega}^{F'} s_{j-1}$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{y} = \mathtt{l_k}$: by $x \not\rightsquigarrow^{F} y \in \Pi(I)$, since $\rho'(x) = \rho'(y) = \rho(s_{j-1})$, it has to be $\left(x \not\rightsquigarrow^{F} y\right)[s_{j-1} \mapsto l_k] \in I_1$. Hence we have $s_{j-1} \not\rightsquigarrow^{F} s_{j-1} \in I$ and, therefore, we obtain $s_{j-1} \not\rightsquigarrow_{\omega}^{F'} s_{j-1} \wedge F \subseteq F'$ and, by Lemma 7.1, $s_{j-1} \not\rightsquigarrow_{\omega}^{F'} s_{j-1}$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.

Regarding expression (7.2), we distinguish the following cases:

- $\mathtt{z} \neq \mathtt{l_k}$: for each $z \rightsquigarrow^{\not\Delta F} \in \Pi(I)$ we have $z \rightsquigarrow^{\not\Delta F} \in I$, since $\rho'(z) = \rho(z)$ and hence the substitution $[s_{j-1} \mapsto l_k]$ did not affect this variable. By $z \rightsquigarrow^{\not\Delta F} \in I$ we obtain $z \rightsquigarrow_{\omega}^{\not\Delta F'} \wedge F \subseteq F'$ and, by Lemma 7.1, $z \rightsquigarrow_{\omega}^{\not\Delta F'}$ entails $z \rightsquigarrow_{\omega'}^{\not\Delta F'}$.
- $\mathtt{z} = \mathtt{l_k}$: by $z \rightsquigarrow^{\not\Delta F} \in \Pi(I)$, since $\rho'(z) = \rho(s_{j-1})$, it has to be $\left(z \rightsquigarrow^{\not\Delta F}\right)[s_{j-1} \mapsto l_k] \in I_2$. Hence we have $s_{j-1} \rightsquigarrow^{\not\Delta F} \in I$ and therefore we obtain $s_{j-1} \rightsquigarrow_{\omega}^{\not\Delta F'} \wedge F \subseteq F'$ and, by Lemma 7.1, $s_{j-1} \rightsquigarrow_{\omega}^{\not\Delta F'}$ entails $z \rightsquigarrow_{\omega'}^{\not\Delta F'}$.

ins = getfield $\kappa.f$:t. We have $L' = L$ and $S' = S$, $\mu = \mu'$ and for all $v \in \mathsf{dom}(\tau') \setminus \{s_{j-1}\}$ we have $\rho'(v) = \rho(v)$, while $\rho'(s_{j-1}) = (\mu(\rho(s_{j-1})).\phi)(\kappa.f:\mathsf{t})$ and hence $\omega' \in \Xi_{\tau'[s_{j-1}\mapsto \mathsf{t}]}$. By Definition 6.2, $\Pi(I) = I_1 \cup I_2 \cup I_3 \cup I_4 \cup I_5 \cup I_6$ where,

$$
\begin{aligned}
I_1 &= \left\{ a \not\rightsquigarrow^{F_{ab}} b,\, c \rightsquigarrow \not\Diamond^{F_c} \in I \,\middle|\, a, b, c \neq s_{j-1} \right\} \\
I_2 &= \left\{ a \not\rightsquigarrow^{\mathcal{F}} s_{j-1} \,\middle|\, a \in \mathsf{dom}(\tau) \setminus \{s_{j-1}\} \wedge \langle a, s_{j-1}.(\kappa.f:\mathsf{t})\rangle \notin \mathcal{MR}_\tau \right\} \\
I_3 &= \left\{ a \not\rightsquigarrow^{F} s_{j-1} \,\middle|\, \begin{array}{l} a \neq s_{j-1} \wedge \langle a, s_{j-1}.(\kappa.f:\mathsf{t})\rangle \in \mathcal{MR}_\tau \wedge \\ a \not\rightsquigarrow^{F} b \in I \wedge \left(b, s_{j-1}.(\kappa.f:\mathsf{t})\right) \in \mathcal{DA}_\tau \end{array} \right\} \\
I_4 &= \left\{ s_{j-1} \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, b \in \mathsf{dom}(\tau) \setminus \{s_{j-1}\} \wedge \langle s_{j-1}.(\kappa.f:\mathsf{t}), b\rangle \notin \mathcal{MR}_\tau \right\} \\
I_5 &= \left\{ s_{j-1} \not\rightsquigarrow^{F'} b \,\middle|\, \begin{array}{l} b \neq s_{j-1} \wedge \langle s_{j-1}.(\kappa.f:\mathsf{t}), b\rangle \in \mathcal{MR}_\tau \wedge \\ s_{j-1} \not\rightsquigarrow^{F} b \in I \wedge F' = F \cup \{\kappa_k.f_k:\mathsf{t}_k \in \mathcal{F} \,|\mathsf{t}_k \notin \mathsf{T(t)}\} \end{array} \right\} \\
I_6 &= \left\{ s_{j-1} \not\rightsquigarrow^{F} s_{j-1} \,\middle|\, a \not\rightsquigarrow^{F} a \in I \wedge \left(a, s_{j-1}.(\kappa.f:\mathsf{t})\right) \in \mathcal{DA}_\tau \right\} \\
I_7 &= \left\{ s_{j-1} \rightsquigarrow \not\Diamond^{F'} \,\middle|\, \begin{array}{l} s_{j-1} \rightsquigarrow \not\Diamond^{F} \in I \wedge \\ F' = F \cup \{\kappa_k.f_k:\mathsf{t}_k \in \mathcal{F} \,|\mathsf{t}_k \notin \mathsf{T(t)}\} \end{array} \right\}
\end{aligned}
$$

About expression (7.1), we distinguish the following cases:

- $\mathtt{x}, \mathtt{y} \neq \mathtt{s}_{\mathtt{j-1}}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$ we have $x \not\rightsquigarrow^{F} y \in I_1 \subseteq I$. By $x \not\rightsquigarrow^{F} y \in I$ we obtain $x \not\rightsquigarrow_{\omega}^{F'} y \wedge F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_{\omega}^{F'} y$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{s}_{\mathtt{j-1}}, \mathtt{y} \neq \mathtt{s}_{\mathtt{j-1}}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$ we have $x \not\rightsquigarrow^{F} y \in I_4$ or $x \not\rightsquigarrow^{F} y \in I_5$. By $x \not\rightsquigarrow^{F} y \in I_4$ we have $\langle s_{j-1}.(\kappa.f:\mathsf{t}), y\rangle \notin \mathcal{MR}_\tau$ that it is equivalent to $\langle x, y\rangle \notin \mathcal{MR}_{\tau'}$ i.e., the new variable can not reach y in τ'. Thus, by Definition 4.7, we obtain $x \not\rightsquigarrow_{\omega'}^{F'} y$ for every $F' \subseteq \mathcal{F}$.
  On the contrary $x \not\rightsquigarrow^{F} y \in I_5$ means $\langle s_{j-1}.(\kappa.f:\mathsf{t}), y\rangle \in \mathcal{MR}_\tau$ i.e., $\langle x, y\rangle \in \mathcal{MR}_{\tau'}$ and hence a necessary condition for reachability. Practically we could have $\rho(y)$ inside or not the set $\mathsf{L}_{\omega'}(x)$. If $\rho(y) \notin \mathsf{L}_{\omega'}(x)$, then, by Definition 4.7, we obtain $x \not\rightsquigarrow_{\omega'}^{F} y$ for every $F \subseteq \mathcal{F}$. Otherwise if $\rho(y) \in \mathsf{L}_{\omega'}(x)$, since we also have $s_{j-1} \not\rightsquigarrow^{F_1} b \in I$ and $F_1 = F \cup \overline{F}$ with $\overline{F} = \{\kappa_k.f_k:\mathsf{t}_k \in \mathcal{F} \,|\mathsf{t}_k \notin \mathsf{T(t)}\}$, we obtain $s_{j-1} \not\rightsquigarrow_{\omega}^{F_1'} b \wedge F_1 \subseteq F_1'$. Furthermore, by Definition 4.7 and by Lemma 7.2, we can assert $a \not\rightsquigarrow_{\omega'}^{F'} b$ with $F' = F_1' \cup \overline{F}$. Hence, since $F_1 \subseteq F_1'$, we have $F = F_1 \cup \overline{F} \subseteq F_1' \cup \overline{F} = F'$.
- $\mathtt{x} \neq \mathtt{s}_{\mathtt{j-1}}, \mathtt{y} = \mathtt{s}_{\mathtt{j-1}}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$ we have $x \not\rightsquigarrow^{F} y \in I_2$ or $x \not\rightsquigarrow^{F} y \in I_3$. If $x \not\rightsquigarrow^{F} y \in I_2$ we have $\langle x, s_{j-1}.(\kappa.f:\mathsf{t})\rangle \notin \mathcal{MR}_\tau$ that it is equivalent to $\langle x, y\rangle \notin \mathcal{MR}_{\tau'}$ i.e., the new variable cannot be reached by x. Thus, by Definition 4.7, we obtain $x \not\rightsquigarrow_{\omega'}^{F'} y$ for every $F' \subseteq \mathcal{F}$.
  On the contrary $x \not\rightsquigarrow^{F} y \in I_3$ means $\langle x, s_{j-1}.(\kappa.f:\mathsf{t})\rangle \in \mathcal{MR}_\tau$ i.e., $\langle x, y\rangle \in \mathcal{MR}_{\tau'}$ and hence a necessary condition for reachability. Practically we could have $\rho(y)$ inside or not the set $\mathsf{L}_{\omega'}(x)$. If $\rho(y) \notin \mathsf{L}_{\omega'}(x)$, then, by Definition 4.7, we obtain $x \not\rightsquigarrow_{\omega'}^{F'} y$ for every $F' \subseteq \mathcal{F}$. Otherwise if $\rho(y) \in \mathsf{L}_{\omega'}(x)$, since we also have $x \not\rightsquigarrow^{F_1} w \in I$ such that $\rho'(y) = \rho(w)$ i.e., the new top of the stack y and w are alias, we obtain $x \not\rightsquigarrow_{\omega}^{F} w \wedge F \subseteq F'$. Since $\rho'(x) = \rho(x)$ also holds, by Lemma 7.1, $x \not\rightsquigarrow_{\omega}^{F} w$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x}, \mathtt{y} = \mathtt{s}_{\mathtt{j-1}}$: by $x \not\rightsquigarrow^{F} y \in \Pi(I)$, it must be $x \not\rightsquigarrow^{F} y \in I_6$ and hence it has also to exist $a \in \mathsf{dom}(\tau)$ such that $a \not\rightsquigarrow^{F} a \in I$. Thus $a \not\rightsquigarrow_{\omega}^{F'} a$ with $F \subseteq F'$. Moreover, since

s_{j-1} and a are alias i.e., $\rho'(x) = \rho'(y) = \rho(a)$, we have that, by Lemma 7.1, $a \not\rightsquigarrow_{\omega}^{F'} a$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.

Regarding expression (7.2), we distinguish the following cases:

- $\mathtt{z} \neq \mathtt{s_{j-1}}$: for each $z \rightsquigarrow^{\not\cap F} \in \Pi(I)$ we have $z \rightsquigarrow^{\not\cap F} \in I_1 \subseteq I$. If $z \rightsquigarrow^{\not\cap F} \in I$ we obtain $z \rightsquigarrow_{\omega}^{\not\cap F'} \wedge F \subseteq F'$ and, by Lemma 7.1, $z \rightsquigarrow_{\omega}^{\not\cap F'}$ entails $z \rightsquigarrow_{\omega'}^{\not\cap F'}$.
- $\mathtt{z} = \mathtt{s_{j-1}}$: by $z \rightsquigarrow^{\not\cap F} \in \Pi(I)$ we have $z \rightsquigarrow^{\not\cap F} \in I_7$. Hence $s_{j-1} \rightsquigarrow^{\not\cap F_1} \in I$ and $F = F_1 \cup \overline{F}$ with $\overline{F} = \{\kappa_k.f_k : \mathtt{t}_k \in \mathcal{F} \mid \mathtt{t}_k \notin \mathsf{T}(\mathtt{t})\}$. By $s_{j-1} \rightsquigarrow^{\not\cap F_1} \in I$ we obtain $c \rightsquigarrow_{\omega}^{\not\cap F_1'} \wedge F_1 \subseteq F_1'$ and, by Lemma 7.1, $c \rightsquigarrow_{\omega}^{\not\cap F_1'}$ entails $c \rightsquigarrow_{\omega'}^{\not\cap F_1'}$. Furthermore, by Definition 4.8 and by Lemma 7.2, we can also assert $c \rightsquigarrow_{\omega'}^{\not\cap F'}$ with $F' = F_1' \cup \overline{F}$. Hence, since $F_1 \subseteq F_1'$, we have $F = F_1 \cup \overline{F} \subseteq F_1' \cup \overline{F} = F'$.

ins = putfield $\kappa.f$:t. We have $L' = L$, $S = S' \setminus \{s_{j-2}, s_{j-1}\}$, $\rho' = \rho$ and $\mu' = \mu[(\rho(s_{j-2}).\phi)(\kappa.f : \mathtt{t}) \mapsto \rho(s_{j-1})]$. Therefore $\mathsf{L}_{\omega'}(s_{j-1}) = \mathsf{L}_{\omega}(s_{j-1})$, where $\mathsf{L}_{\omega'}(s_{j-2}) = \mathsf{L}_{\omega}(s_{j-2}) \cup \mathsf{L}_{\omega}(s_{j-1})$ and, hence, for all v such that $\rho(s_{j-2}) \notin \mathsf{L}_{\omega}(v)$ or $\rho(s_{j-1}) \in \mathsf{L}_{\omega}(v)$ we have $\mathsf{L}_{\omega}(v) = \mathsf{L}_{\omega'}(v)$. By Definition 6.2, $\Pi(I) = I_1 \cup I_2 \cup I_3 \cup I_4 \cup I_5$ where,

$$I_1 = \left\{ a \not\rightsquigarrow^{F} b \in I \;\middle|\; \begin{array}{l} a, b \notin \{s_{j-1}, s_{j-2}\} \wedge \\ \left(\langle a, s_{j-2}\rangle \notin \mathcal{MR}_\tau \vee \langle s_{j-1}, b\rangle \notin \mathcal{MR}_\tau\right) \end{array} \right\}$$

$$I_2 = \left\{ a \not\rightsquigarrow^{F'} b \;\middle|\; \begin{array}{l} 1.\ a, b \notin \{s_{j-1}, s_{j-2}\} \wedge \langle a, s_{j-2}\rangle, \langle s_{j-1}, b\rangle \in \mathcal{MR}_\tau \\ 2.\ a \not\rightsquigarrow^{F_{ab}} b,\ a \not\rightsquigarrow^{F_{a2}} s_{j-2},\ s_{j-1} \not\rightsquigarrow^{F_{1b}} b \in I \\ 3.\ F' = \{F_{ab} \cap F_{a2} \cap F_{1b}\} \setminus \{\kappa.f : \mathtt{t}\} \end{array} \right\}$$

$$I_3 = \left\{ c \rightsquigarrow^{\not\cap F} \in I \;\middle|\; \begin{array}{l} c \notin \{s_{j-1}, s_{j-2}\} \wedge \\ \langle c, s_{j-1}\rangle, \langle c, s_{j-2}\rangle \notin \mathcal{MR}_\tau \end{array} \right\}$$

$$I_4 = \left\{ c \rightsquigarrow^{\not\cap F'} \;\middle|\; \begin{array}{l} 1.\ c \notin \{s_{j-1}, s_{j-2}\} \wedge \langle s_{j-1}, s_{j-2}\rangle \notin \mathcal{MR}_\tau \\ 2.\ \langle c, s_{j-2}\rangle \in \mathcal{MR}_\tau \vee \langle c, s_{j-1}\rangle \in \mathcal{MR}_\tau \\ 3.\ c \rightsquigarrow^{\not\cap F_c},\ s_{j-1} \rightsquigarrow^{\not\cap F_{j1}} \in I \wedge F' = F_c \cap F_{j1} \end{array} \right\}$$

$$I_5 = \left\{ c \rightsquigarrow^{\not\cap F'} \;\middle|\; \begin{array}{l} 1.\ c \notin \{s_{j-1}, s_{j-2}\} \wedge \langle s_{j-1}, s_{j-2}\rangle \in \mathcal{MR}_\tau \\ 2.\ \langle c, s_{j-1}\rangle \in \mathcal{MR}_\tau \vee \langle c, s_{j-2}\rangle \in \mathcal{MR}_\tau \\ 3.\ s_{j-1} \not\rightsquigarrow^{F_{12}} s_{j-2},\ c \rightsquigarrow^{\not\cap F_c},\ s_{j-1} \rightsquigarrow^{\not\cap F_{j1}} \in I \\ 4.\ F' = \left\{F_c \cap F_{j1} \cap F_{12}\right\} \setminus \{\kappa.f : \mathtt{t}\} \end{array} \right\}$$

About expression (7.1), for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$ we distinguish the following cases:

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I_1}$: since $x \not\rightsquigarrow^{F} y \in I$, we obtain $x \not\rightsquigarrow_{\omega}^{F'} y \wedge F' \subseteq F$ and, since $\langle x, s_{j-2}\rangle$ or $\langle s_{j-1}, y\rangle$ not inside $\mathcal{MR}_\tau$, we have two sufficient conditions for unreachability. In particular $\langle x, s_{j-2}\rangle \notin \mathcal{MR}_\tau$ entails $\rho(s_{j-2}) \notin \mathsf{L}_{\omega}(x)$ and hence, by definition of μ', we satisfy the condition 3 and 4 of Lemma 7.1 i.e., $x \not\rightsquigarrow_{\omega'}^{F'} y$.
  Contrariwise $\langle s_{j-1}, y\rangle \notin \mathcal{MR}_\tau$ entails $\rho(y) \doteq \rho(y) \notin \mathsf{L}_{\omega}(s_{j-1}) \doteq \mathsf{L}_{\omega'}(s_{j-1})$. Therefore for every $v \in \mathsf{dom}(\tau) \setminus \{s_{j-1}, s_{j-2}\} \doteq \mathsf{dom}(\tau')$ and for every path $\mathcal{P}$ such that $v \rightsquigarrow_{\omega'}^{\mathcal{P}} y$, we have that for all $\ell \in \mathsf{dom}(\mu)$, it does not exist a field $\kappa'.f : \mathtt{t}' \in \mathcal{P}$ such that $(\mu(\ell).\phi)(\kappa'.f : \mathtt{t}') = \rho(s_{j-1})$. Thus also for every $\mathcal{P}$ such that $x \rightsquigarrow_{\omega'}^{\mathcal{P}} y$ this assertion holds and, hence, as $\kappa.f : \mathtt{t}$ is such that $(\mu'(\rho(s_{j-2})).\phi)(\kappa.f : \mathtt{t}) = \rho(s_{j-1})$, it cannot be in that $\mathcal{P}$. Hence all paths from x to y remain unchanged in ω' i.e., $x \not\rightsquigarrow_{\omega'}^{F} y$.

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_2$: we have that x may reach s_{j-2}, s_{j-1} may reach y and $x \not\rightsquigarrow_{\omega}^{F'_{xy}} y$ with $F_{xy} \subseteq F'_{xy}$. We note that, as $F \subseteq F_{xy}$, we obtain $F \subseteq F'_{xy}$. If $\rho(s_{j-2}) \notin \mathsf{L}_\omega(x)$ or $\rho(y) \notin \mathsf{L}_\omega(s_{j-1})$, since $F \subseteq F_{xy}$, we can, thanks to Lemma 4.7, follow the same reasoning made with the set I_1 and then conclude $x \not\rightsquigarrow_{\omega'}^{F'_{xy}} y$.
  Contrariwise if $\rho(s_{j-2}) \in \mathsf{L}_\omega(x)$ and $\rho(y) \in \mathsf{L}_\omega(s_{j-1})$, new paths between x and y are established in ω' and all these paths are shaped as $\mathcal{P}_k = \mathcal{P}_i :: \langle \kappa.f : \mathsf{t} \rangle :: \mathcal{P}_h$, where $\mathcal{P}_i, \mathcal{P}_h \subseteq \mathcal{F}$ are such that $x \rightsquigarrow_{\omega}^{\mathcal{P}_i} s_{j-2}$ and $s_{j-1} \rightsquigarrow_{\omega}^{\mathcal{P}_h} y$, respectively. Since every $\mathcal{P}_i$ and every $\mathcal{P}_h$ is obviously different to $\mathcal{P}_k$, it remains unchanged also in ω' i.e., from $x \rightsquigarrow_{\omega'}^{\mathcal{P}_i} s_{j-2}$ and $s_{j-1} \rightsquigarrow_{\omega'}^{\mathcal{P}_h} y$ we obtain, by Definition 4.7, $x \not\rightsquigarrow_{\omega'}^{F'_{x1}} s_{j-2}$ with $F_{x2} \subseteq F'_{x2}$ and $s_{j-1} \not\rightsquigarrow_{\omega'}^{F'_{1y}} y$ with $F_{1y} \subseteq F'_{1y}$. Thus, as $\mathcal{P}_i \cap F'_{x2} = \emptyset$ and $\mathcal{P}_h \cap F'_{1y} = \emptyset$, we also have $\mathcal{P}_k \cap [(F'_{x2} \cap F'_{1y}) \setminus \{\kappa.f : \mathsf{t}\}] = \emptyset$. Furthermore, since for all paths $\mathcal{P}_l \subseteq \mathcal{F}$ such that $x \rightsquigarrow_{\omega}^{\mathcal{P}_l} y$ we also have $\mathcal{P}_l \cap F'_{xy} = \emptyset$, we must preserve in ω' the consistency between all paths, both old ($\mathcal{P}_l$) and new ($\mathcal{P}_k$). Therefore we set up the set $F' = (F'_{xy} \cap F'_{x2} \cap F'_{1y}) \setminus \{\kappa.f : \mathsf{t}\}$, so that $\mathcal{P}_k \cup F' = \mathcal{P}_l \cup F' = \emptyset$. Thus, we can assert $x \not\rightsquigarrow_{\omega'}^{F'} y$ and, by construction of F, also $F \subseteq F'$.

Regarding expression (7.2), for each $z \rightsquigarrow \not\emptyset^F \in \Pi(I)$ we distinguish the following cases:

- $\mathtt{z} \rightsquigarrow \not\emptyset^{\mathtt{F}} \in \mathtt{I}_3$: since $z \rightsquigarrow \not\emptyset^F \in I$, we obtain $z \rightsquigarrow_{\omega}^{\not\emptyset^{F'}} \wedge F' \subseteq F$ and, since $\langle x, s_{j-2} \rangle$ and $\langle s_{j-1}, y \rangle$ are not inside $\mathcal{MR}_\tau$, we have a sufficient condition for unreachability neither s_{j-2} nor s_{j-1} i.e., $\rho(s_{j-2}), \rho(s_{j-1}) \notin \mathsf{L}_\omega(z)$. Therefore, by definition of μ', we satisfy the condition 3 and 4 of Lemma 7.1 i.e., $z \rightsquigarrow_{\omega'}^{\not\emptyset^{F'}}$.
- $\mathtt{z} \rightsquigarrow \not\emptyset^{\mathtt{F}} \in \mathtt{I}_4$: we have $s_{j-1} \rightsquigarrow \not\emptyset^{F_{j1}}, z \rightsquigarrow \not\emptyset^{F_z} \in I$, hence $s_{j-1} \rightsquigarrow_{\omega}^{\not\emptyset^{F'_{j1}}} \wedge F_{j1} \subseteq F'_{j1}$, $z \rightsquigarrow_{\omega}^{\not\emptyset^{F'_z}} \wedge F_z \subseteq F'_z$ and, since $F \subseteq F_z$, also $F \subseteq F'_z$. Furthermore we have a sufficient condition for unreachability of s_{j-2} from s_{j-1} i.e., $\rho(s_{j-2}) \notin \mathsf{L}_\omega(s_{j-1})$. Therefore we can assert that there are not new cycles in ω' i.e., $\kappa.f : \mathsf{t}$ cannot become part of a cyclic-path $\mathcal{P}$, $\ell \rightsquigarrow_{\mu'}^{\mathcal{P}} \ell$.[1] Whereas it may be that $\rho(s_{j-1}) \in \mathsf{L}_\omega(z)$ or $\rho(s_{j-2}) \in \mathsf{L}_\omega(z)$. If $\rho(s_{j-1}), \rho(s_{j-2}) \notin \mathsf{L}_\omega(z)$, since $F \subseteq F_z$, we can, thanks to Lemma 4.7, follow the same reasoning made with the set I_3 and then conclude $z \rightsquigarrow_{\omega'}^{\not\emptyset^{F'_z}}$. Else if $\rho(s_{j-2}) \notin \mathsf{L}_\omega(z)$ or $\rho(s_{j-1}) \in \mathsf{L}_\omega(z)$ then, by Definition of μ', $\mathsf{L}_\omega(z) = \mathsf{L}_{\omega'}(z)$ and hence, by Lemma 7.1, $z \rightsquigarrow_{\omega'}^{\not\emptyset^{F'_z}}$.
  Otherwise if $\rho(s_{j-2}) \in \mathsf{L}_\omega(z)$, but $\rho(s_{j-1}) \notin \mathsf{L}_\omega(z)$ then, as by definition of μ' we have $\rho(s_{j-1}) \in \mathsf{L}_{\omega'}(s_{j-2})$, we obtain $\rho(s_{j-1}) \in \mathsf{L}_{\omega'}(z)$ and $\mathsf{L}_{\omega'}(z) = \mathsf{L}_\omega(z) \cup \mathsf{L}_\omega(s_{j-1})$. Therefore, in ω', z reaches also all the cycles that were already reached by s_{j-1} in ω. For every $\mathcal{P}_i, \mathcal{P}_h \subseteq \mathcal{F}$ such that exist $\ell_1 \in \mathsf{L}_\omega(z)$ and $\ell_2 \in \mathsf{L}_\omega(s_{j-1})$ which ensure $\ell_1 \rightsquigarrow_{\omega}^{\mathcal{P}_i} \ell_1$ and $\ell_2 \rightsquigarrow_{\omega}^{\mathcal{P}_h} \ell_2$, we have, by Definition 4.8, $F'_z \cap \mathcal{P}_i = F'_{j1} \cap \mathcal{P}_h = \emptyset$. Hence we obtain $\mathcal{P}_k \cap (F'_z \cap F'_{j1}) = \emptyset$, where $\mathcal{P}_k$ is such that $\ell_3 \rightsquigarrow_{\omega'}^{\mathcal{P}_k} \ell_3$ with $\ell_3 \in \mathsf{L}_{\omega'}(z)$. Thus, by setting $F' = F'_z \cap F'_{j1}$, we have $z \rightsquigarrow_{\omega'}^{\not\emptyset^{F'}}$ and, by construction of F, also $F \subseteq F'$.
- $\mathtt{z} \rightsquigarrow \not\emptyset^{\mathtt{F}} \in \mathtt{I}_5$: we have $s_{j-1} \not\rightsquigarrow^{F_{12}} s_{j-2}$, $s_{j-1} \rightsquigarrow \not\emptyset^{F_{j1}}, z \rightsquigarrow \not\emptyset^{F_z} \in I$, hence $s_{j-1} \not\rightsquigarrow_{\omega}^{F'_{12}} s_{j-2} \wedge F_{12} \subseteq F'_{12}$, $s_{j-1} \rightsquigarrow_{\omega}^{\not\emptyset^{F'_{j1}}} \wedge F_{j1} \subseteq F'_{j1}$, $z \rightsquigarrow_{\omega}^{\not\emptyset^{F'_z}} \wedge F_z \subseteq F'_z$ and, furthermore,

[1] That because, by Definition 4.4, in every cyclic-path $\mathcal{P}$ with $\kappa.f : \mathsf{t}$ inside, it exist $\ell^h, \ell^k \in \mathsf{L}_\omega(s_{j-1})$ and $\kappa'.f : \mathsf{t}', \kappa''.f : \mathsf{t}'' \in \mathcal{P}$ such that $(\mu'(\rho(s_{j-1})).\phi)(\kappa'.f : \mathsf{t}') = \ell^h$, $\ell^k = \mathsf{L}_\omega(\ell^h)$ and $(\mu'(\ell^k).\phi)(\kappa''.f : \mathsf{t}'') = \rho(s_{j-2})$ i.e., $\rho(s_{j-2})$ should be inside $\mathsf{L}_\omega(s_{j-1})$, which is false by hypothesis.

it may be that $\rho(s_{j-2}) \in \mathsf{L}_\omega(s_{j1})$. If $\rho(s_{j-2}) \notin \mathsf{L}_\omega(s_{j-1})$, since by construction $F \subseteq F_z \cap F_{j1}$, we can, thanks to Lemma 4.7, follow the same reasoning made with I_4 and conclude $z \rightsquigarrow_{\omega'}^{\not\Diamond F'}$ with $F' = F_z \cap F_{j1}$.

Otherwise certainly $\rho(s_{j-2}) \in \mathsf{L}_\omega(s_{j-1})$ and then new cycles, all going through $\kappa.f:\mathsf{t}$, are established in ω'. These new cycles are all shaped as $\mathcal{P}_k = \langle \kappa.f:\mathsf{t}\rangle :: \mathcal{P}_i$, where $\mathcal{P}_i \subseteq \mathcal{F}$ is such that $s_{j-1} \rightsquigarrow_{\omega}^{\mathcal{P}_i} s_{j-2}$. Since every $\mathcal{P}_i$ is obviously different to $\mathcal{P}_k$, it remains unchanged also in ω' i.e., from $s_{j-1} \rightsquigarrow_{\omega'}^{\mathcal{P}_i} s_{j-2}$ we obtain, by Definition 4.7, $s_{j-1} \not\rightsquigarrow_{\omega'}^{F'_{12}} s_{j-2}$ with $F_{12} \subseteq F'_{12}$. Therefore, as $\mathcal{P}_i \cap F'_{12} = \emptyset$, also $\mathcal{P}_k \cap (F'_{12} \setminus \{\kappa.f:\mathsf{t}\}) = \emptyset$.

Furthermore, since for all the cycle-paths $\mathcal{P}_l, \mathcal{P}_h \subseteq \mathcal{F}$ such that $\ell_1 \rightsquigarrow_{\omega}^{\mathcal{P}_l} \ell_1$ and $\ell_2 \rightsquigarrow_{\omega}^{\mathcal{P}_h} \ell_2$, where $\ell_1 \in \mathsf{L}_\omega(z)$ and $\ell_2 \in \mathsf{L}_\omega(s_{j-1})$, we also have $\mathcal{P}_l \cap F'_z = \mathcal{P}_h \cap F'_{j1} = \emptyset$. By following a reasoning like that made with I_4 set, as it may be that $\rho(s_{j-2} \in \mathsf{L}_\omega(z)$ but $\rho(s_{j-1} \notin \mathsf{L}_\omega(z)$, we must preserve in ω' the consistency between all paths, both old ($\mathcal{P}_l$, $\mathcal{P}_h$) and new ($\mathcal{P}_k$). Therefore we set up the set $F' = (F'_z \cap F'_{12} \cap F'_{j1}) \setminus \{\kappa.f:\mathsf{t}\}$, so that $\mathcal{P}_l \cap F' = \mathcal{P}_h \cap F' = \mathcal{P}_k \cap F' = \emptyset$. We can hence assert $z \rightsquigarrow_{\omega'}^{\not\Diamond F'}$ and, by construction of F, also $F \subseteq F'$.

ins = catch, excp_is K, ifne t, ifeq t. We can prove these four instructions together.

About catch and excp_is K instructions we have $L' = L$, $S' = S = s_0$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau')$, $\rho'(v) = \rho(v)$. About ifne t and ifeq t instructions, instead of $S' = S = s_0$ we have $S' = S \setminus s_{j-1}$. In both cases, by Definition 6.2, the propagation rule is

$$\Pi(I) = \left\{ a \not\rightsquigarrow^{F_{ab}} b,\ c \rightsquigarrow^{\not\Diamond F_c} \in I \,\middle|\, a, b, c \in \mathsf{dom}(\tau') \right\}$$

About expression (7.1) for each $x \not\rightsquigarrow^{F_{xy}} y \in \Pi(I)$ we have $x \not\rightsquigarrow^{F_{xy}} y \in I$ and hence also $x \not\rightsquigarrow_{\omega}^{F'_{xy}} y$ with $F_{xy} \subseteq F'_{xy}$. Therefore, by Lemma 7.1, we obtain $x \not\rightsquigarrow_{\omega'}^{F'_{xy}} y$. Regarding expression (7.2), for each $z \rightsquigarrow^{\not\Diamond F_z} \in \Pi(I)$ we have $z \rightsquigarrow^{\not\Diamond F_z} \in I$ and hence also $z \rightsquigarrow_{\omega}^{\not\Diamond F'_z}$ with $F_z \subseteq F'_z$. Therefore, by Lemma 7.1, we obtain $z \rightsquigarrow_{\omega'}^{\not\Diamond F'_z}$.

□

7.2.2 Soundness of Final Arcs

Lemma 7.4 (Final Arcs)**.** *The normalized propagation rules for* final arcs *of Definition 5.3 are correct. That is, consider a final arc from a bytecode* ins *and its normalized propagation rule* Φ*. Assume that* ins *has* static type *information* τ *at its beginning and* τ' *immediately after its execution (its non-exceptional execution if* ins *is a* return t*, its exceptional execution if* ins *is a* throw κ*). Then, for every* $I \in \mathsf{A}_\tau$ *we have*

$$\mathit{ins}\,(\gamma_\tau(I)) \subseteq \gamma_{\tau'}(\Phi(I))$$

(we recall that ins *is the semantic of* ins *bytecode).*

Proof. Since $\gamma_{\tau'}(\Pi(I)) = \gamma_{\tau'}(\Phi(I))$ by Corollary 6.1.1, it is enough to prove

$$\mathit{ins}\,(\gamma_\tau(I)) \subseteq \gamma_{\tau'}(\Pi(I)).$$

Let $\mathsf{dom}(\tau) = L \cup S$ contains i local variables $L = \{l_0, \ldots, l_{i-1}\}$ and j stack elements $S = \{s_0, \ldots, s_{j-1}\}$. Furthermore let $\mathsf{dom}(\tau') = L' \cup S'$, where L' and S' are the local and

stack variables after the execution of ins. Consider an arbitrary abstract element $I \in \mathsf{A}_\tau$ and an arbitrary state $\omega' = \langle \rho', \mu' \rangle \in ins\,(\gamma_\tau\,(I))$. We prove that $\omega' \in \gamma_{\tau'}\,(\Pi\,(I))$ i.e., by Definition 5.2 that

$$\text{for each } x \not\rightsquigarrow^{F} y \in \Pi(I) \text{ we have } x \not\rightsquigarrow_{\omega'}^{F'} y \wedge F \subseteq F' \tag{7.3}$$

$$\text{for each } z \rightsquigarrow \!\!\!\!\diagup^{F} \in \Pi(I) \text{ we have } z \rightsquigarrow \!\!\!\!\diagup_{\omega'}^{F'} \wedge F \subseteq F' \tag{7.4}$$

Note that, by the choice of ω', there exists $\omega = \langle \rho, \mu \rangle \in \gamma_\tau(I)$ such that $\omega' = ins(\omega)$ and, by Definition 5.2, for each $x \not\rightsquigarrow^{F} y \in I$ we obtain $x \not\rightsquigarrow_{\omega}^{F'} y$ with $F \subseteq F'$ and, respectively, for each $z \rightsquigarrow \!\!\!\!\diagup^{F} \in I$ we obtain $z \rightsquigarrow \!\!\!\!\diagup_{\omega}^{F'}$ with $F \subseteq F'$. We analyze different propagation rules corresponding to different types of final arcs.

ins = return void. We have $L' = L$ and $S' = \emptyset$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau')$, $\rho'(v) = \rho(v)$. By Definition 6.2, $\Pi(I) = \left\{ a \not\rightsquigarrow^{F_{ab}} b,\ c \rightsquigarrow \!\!\!\!\diagup^{F_c} \in I \,\middle|\, a, b, c \notin \{s_0, \ldots, s_{j-1}\} \right\}$. About expression (7.3), for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$ we have $x \not\rightsquigarrow^{F} y \in I$ and, hence, $x \not\rightsquigarrow_{\omega}^{F'} y$ with $F \subseteq F'$. Since $\mu = \mu'$, $x \not\rightsquigarrow_{\omega}^{F'} y$ entails, by Lemma 7.1, $x \not\rightsquigarrow_{\omega'}^{F'} y$. Regarding (7.4), for each $z \rightsquigarrow \!\!\!\!\diagup^{F} \in \Pi(I)$ we have $z \rightsquigarrow \!\!\!\!\diagup^{F} \in I$ and, hence, $z \rightsquigarrow \!\!\!\!\diagup_{\omega}^{F'}$ with $F \subseteq F'$. Since $\mu = \mu'$, $z \rightsquigarrow \!\!\!\!\diagup_{\omega}^{F'}$ entails, by Lemma 7.1, $z \rightsquigarrow \!\!\!\!\diagup_{\omega'}^{F'}$.

ins = return t. We have $L' = L$, $S' = \{s_0\}$, $\mu' = \mu$ and, for every $v \in \mathsf{dom}(\tau') \setminus \{s_0\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_0) = \rho(s_{j-1})$. By Definition 6.2,

$$\Pi(I) = \overbrace{\left\{ \left(a \not\rightsquigarrow^{F} b \right) \left[s_{j-1} \mapsto s_0 \right] \,\middle|\, a \not\rightsquigarrow^{F} b \in I \wedge a, b \notin \{s_0, \ldots, s_{j-2}\} \right\}}^{I_1} \cup \underbrace{\left\{ \left(c \rightsquigarrow \!\!\!\!\diagup^{F} \right) \left[s_{j-1} \mapsto s_0 \right] \,\middle|\, c \rightsquigarrow \!\!\!\!\diagup^{F} \in I \wedge c \notin \{s_0, \ldots, s_{j-2}\} \right\}}_{I_2}.$$

About expression (7.3), we distinguish the following cases:

- $\mathtt{x}, \mathtt{y} \neq \mathtt{s_0}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$, we have $x \not\rightsquigarrow^{F} y \in I$, since the substitution $[s_{j-1} \mapsto s_0]$ did not effect these two variables and hence $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. By $x \not\rightsquigarrow^{F} y \in I$, we obtain $x \not\rightsquigarrow_{\omega}^{F'} y$ with $F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_{\omega}^{F'} y$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{s_0}, \mathtt{y} \neq \mathtt{s_0}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$, since $\rho'(x) = \rho(s_{j-1})$ and $\rho'(y) = \rho(y)$, it has to be $x \not\rightsquigarrow^{F} y = \left(s_{j-1} \not\rightsquigarrow^{F} y \right) [s_{j-1} \mapsto s_0] \in I_1$. Then we have $s_{j-1} \not\rightsquigarrow^{F} y \in I_1$ and, therefore, we obtain $s_{j-1} \not\rightsquigarrow_{\omega}^{F'} y$ with $F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} \neq \mathtt{s_0}, \mathtt{y} = \mathtt{s_0}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$, since $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(s_{j-1})$, it has to be $x \not\rightsquigarrow^{F} y = \left(x \not\rightsquigarrow^{F} s_{j-1} \right) [s_{j-1} \mapsto s_0] \in I_1$. Then we have $x \not\rightsquigarrow^{F} s_{j-1} \in I$ and, therefore, we obtain $x \not\rightsquigarrow_{\omega}^{F'} s_{j-1}$ with $F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{y} = \mathtt{s_0}$: by $x \not\rightsquigarrow^{F} y \in \Pi(I)$, since $\rho'(x) = \rho'(y) = \rho(s_{j-1})$, it has to be $x \not\rightsquigarrow^{F} y = \left(s_{j-1} \not\rightsquigarrow^{F} s_{j-1} \right) [s_{j-1} \mapsto s_0] \in I_1$. Hence we have $s_{j-1} \not\rightsquigarrow^{F} s_{j-1} \in I$ and, therefore, we obtain $s_{j-1} \not\rightsquigarrow_{\omega}^{F'} s_{j-1}$ with $F \subseteq F'$ and, by Lemma 7.1, $s_{j-1} \not\rightsquigarrow_{\omega}^{F'} s_{j-1}$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.

Regarding (7.4), we distinguish the following cases:

- $\mathtt{z} \neq \mathtt{s_0}$: for each $z \rightsquigarrow^{\not\Rightarrow F} \in \Pi(I)$, we have $z \rightsquigarrow^{\not\Rightarrow F} \in I$, since $\rho'(z) = \rho(z)$ and hence the substitution $[s_{j-1} \mapsto s_0]$ did not affect this variable. By $z \rightsquigarrow^{\not\Rightarrow F} \in I$ we obtain $z \rightsquigarrow_{\omega}^{\not\Rightarrow F'}$ with $F \subseteq F'$ and, by Lemma 7.1, $z \rightsquigarrow_{\omega}^{\not\Rightarrow F'}$ entails $z \rightsquigarrow_{\omega'}^{\not\Rightarrow F'}$.
- $\mathtt{z} = \mathtt{s_0}$: by $z \rightsquigarrow^{\not\Rightarrow F} \in \Pi(I)$, since $\rho'(z) = \rho(s_{j-1})$, it has to be $z \rightsquigarrow^{\not\Rightarrow F} = \left(s_{j-1} \rightsquigarrow^{\not\Rightarrow F}\right)[s_{j-1} \mapsto s_0] \in I_2$. Hence we have $s_{j-1} \rightsquigarrow^{\not\Rightarrow F} I$ and, therefore, we obtain $s_{j-1} \rightsquigarrow_{\omega}^{\not\Rightarrow F'}$ with $F \subseteq F'$ and, by Lemma 7.1, $s_{j-1} \rightsquigarrow_{\omega}^{\not\Rightarrow F'}$ entails $s_{j-1} \rightsquigarrow_{\omega'}^{\not\Rightarrow F'}$.

ins = throw κ. We have $L' = L$, $S' = \{s_0\}$ and, for every $v \in \mathsf{dom}(\tau') \setminus \{s_0\}$, $\rho'(v) = \rho(v)$. By Definition 6.2,

$$\Pi(I) = \overbrace{\underbrace{\left\{\left(a \not\rightsquigarrow^{F} b\right)\left[s_{j-1} \mapsto s_0\right] \middle| a \not\rightsquigarrow^{F} b \in I \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\right\}}_{I_2} \cup \left\{\left(c \rightsquigarrow^{\not\Rightarrow F}\right)\left[s_{j-1} \mapsto s_0\right] \middle| c \rightsquigarrow^{\not\Rightarrow F} \in I \wedge c \notin \{s_0, \ldots, s_{j-2}\}\right\}}^{I_1}.$$

We have two possibilities: either $\rho'(s_0) = \rho(s_{j-1})$ and $\mu' = \mu$, in which case, with the same proof as for return t above, we conclude our two theses 7.3 and 7.4. Or otherwise $\rho'(s_0) = \ell$ where ℓ is fresh and $\mu' = \mu[\ell \mapsto \mathtt{npe}]$, where npe is a new object of class `NullPointerException` containing only fresh locations $\mathsf{L}_{\mu'}(\ell) \cap \mathsf{dom}(\mu) = \emptyset$. In this latter case, about expression (7.3) we distinguish the following cases:

- $\mathtt{x}, \mathtt{y} \neq \mathtt{s_0}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$, we have $x \not\rightsquigarrow^{F} y \in I$, since the substitution $[s_{j-1} \mapsto s_0]$ did not effect these two variables and hence $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. By $x \not\rightsquigarrow^{F} y \in I$, we obtain $x \not\rightsquigarrow_{\omega}^{F'} y$ with $F \subseteq F'$ and, by Lemma 7.1, $x \not\rightsquigarrow_{\omega}^{F'} y$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{s_0}, \mathtt{y} \neq \mathtt{s_0}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$, we have $\rho'(x) = \rho'(s_0) = \ell$ and $\rho'(y) = \rho(y)$. Hence, since ℓ is fresh, we can state $x \not\rightsquigarrow_{\omega'}^{\mathcal{F}} y$ and, by Lemma 4.7, for each F' such that $F \subseteq F' \subseteq \mathcal{F}$ we have $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} \neq \mathtt{s_0}, \mathtt{y} = \mathtt{s_0}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$, we have $\rho'(x) = \rho'(x)$ and $\rho'(y) = \rho(s_0) = \ell$. Hence, since ℓ is fresh, we can state $x \not\rightsquigarrow_{\omega'}^{\mathcal{F}} y$ and, by Lemma 4.7, for each F' such that $F \subseteq F' \subseteq \mathcal{F}$ we have $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{y} = \mathtt{s_0}$: for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$, we have $\rho'(x) = \rho'(y) = \rho'(s_0) = \ell$. Hence, since ℓ is fresh, we can state $x \not\rightsquigarrow_{\omega'}^{\mathcal{F}} y$ and, by Lemma 4.7, for each F' such that $F \subseteq F' \subseteq \mathcal{F}$ we have $x \not\rightsquigarrow_{\omega'}^{F'} y$.

Regarding (7.4), we distinguish the following cases:

- $\mathtt{z} \neq \mathtt{s_0}$: for each $z \rightsquigarrow^{\not\Rightarrow F} \in \Pi(I)$, we have $z \rightsquigarrow^{\not\Rightarrow F} \in I$, since $\rho'(z) = \rho(z)$ and hence the substitution $[s_{j-1} \mapsto s_0]$ did not affect this variable. By $z \rightsquigarrow^{\not\Rightarrow F} \in I$ we obtain $z \rightsquigarrow_{\omega}^{\not\Rightarrow F'}$ with $F \subseteq F'$ and, by Lemma 7.1, $z \rightsquigarrow_{\omega}^{\not\Rightarrow F'}$ entails $z \rightsquigarrow_{\omega'}^{\not\Rightarrow F'}$.
- $\mathtt{z} = \mathtt{s_0}$: for each $z \rightsquigarrow^{\not\Rightarrow F} \in \Pi(I)$, we have $\rho'(z) = \rho'(s_0) = \ell$. Hence, since ℓ is fresh, we can state $z \rightsquigarrow_{\omega'}^{\not\Rightarrow \mathcal{F}}$ and, by Lemma 4.7, for each F' such that $F \subseteq F' \subseteq \mathcal{F}$ we have $z \rightsquigarrow_{\omega'}^{\not\Rightarrow F'}$.

□

7.2.3 Soundness of Parameter Passing Arcs

Lemma 7.5 (Parameter Passing Arcs)**.** *The normalized propagation rules for* parameter passing arcs *of Definition 5.3 are correct. That is, consider a parameter passing arc from a* call $m_1 \dots m_k$ *to the first bytecode instruction of* m_w*, for some* $w \in [1 \dots k]$*, and its normalized propagation rule* Φ*. Assume that* call $m_1 \dots m_k$ *has* static type *information* τ *at its beginning and* τ' *immediately after its execution. Then, for every* $I \in \mathsf{A}_\tau$ *we have*

$$(makescope\ m_w)(\gamma_\tau(I)) \subseteq \gamma_{\tau'}(\Phi(I))$$

Proof. Since $\gamma_{\tau'}(\Pi(I)) = \gamma_{\tau'}(\Phi(I))$ by Corollary 6.1.1, it is enough to prove

$$(makescope\ m_w)(\gamma_\tau(I)) \subseteq \gamma_{\tau'}(\Pi(I)).$$

Let $\mathsf{dom}(\tau) = L \cup S$ contains i local variables $L = \{l_0, \dots, l_{i-1}\}$ and $j \geq \pi$ stack elements $S = \{s_0, \dots, s_{j-\pi}, \dots, s_{j-1}\}$ where π is the number of the parameters of method m_w, including this. Then $\mathsf{dom}(\tau') = \{l_0, \dots, l_{\pi-1}\}$ i.e., $S' = \epsilon$. Consider an arbitrary abstract element $I \in \mathsf{A}_\tau$ and an arbitrary state $\omega' = \langle \rho', \mu' \rangle \in (makescope\ m_w)(\gamma_\tau(I))$. We prove that $\omega' \in \gamma_{\tau'}(\Pi(I))$ i.e., by Definition 5.2 that

$$\text{for each } x \not\rightsquigarrow^F y \in \Pi(I) \text{ we have } x \not\rightsquigarrow^{F'}_{\omega'} y \wedge F \subseteq F' \tag{7.5}$$

$$\text{for each } z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in \Pi(I) \text{ we have } z \rightsquigarrow\!\!\!\!\!\!\diagup^{F'}_{\omega'} \wedge F \subseteq F' \tag{7.6}$$

Note that, by the choice of ω', there exists $\omega = \langle \rho, \mu \rangle \in \gamma_\tau(I)$ such that $\omega' = (makescope\ m_w)(\omega)$ and, by Definition 5.2, for each $x \not\rightsquigarrow^F y \in I$ we obtain $x \not\rightsquigarrow^{F'}_{\omega} y$ with $F \subseteq F'$ and, respectively, for each $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I$ we obtain $z \rightsquigarrow\!\!\!\!\!\!\diagup^{F'}_{\omega}$ with $F \subseteq F'$.

By Definition 3.6, for every $p \in [0, \pi]$, $\rho'(l_p) = \rho(s_{j-\pi-p})$ and $\mu' = \mu$. Furthermore by Definition 6.2,

$$\Pi(I) = \overbrace{\underbrace{\left\{ \left(a \not\rightsquigarrow^F b \right) \begin{bmatrix} s_{j-\pi} \mapsto l_0 \\ \dots \\ s_{j-1} \mapsto l_{\pi-1} \end{bmatrix} \middle| \, a \not\rightsquigarrow^F b \in I \wedge a, b \in \{s_{j-\pi}, \dots, s_{j-1}\} \right\}}^{I_1} \cup \underbrace{\left\{ \left(c \rightsquigarrow\!\!\!\!\!\!\diagup^F \right) \begin{bmatrix} s_{j-\pi} \mapsto l_0 \\ \dots \\ s_{j-1} \mapsto l_{\pi-1} \end{bmatrix} \middle| \, c \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I \wedge c \in \{s_{j-\pi}, \dots, s_{j-1}\} \right\}}_{I_2}}$$

About expression (7.5), for each $x, y \in \mathsf{dom}(\tau') = L'$ such that $x \not\rightsquigarrow^F y \in I_1 \subseteq \Pi(I)$, there exist $p, q \in [0, \pi]$ such that $x = l_p$ and $y = l_q$; therefore $\rho'(x) = \rho'(l_p) = \rho(s_{j-\pi-p})$ and $\rho'(y) = \rho(l_q) = \rho(s_{j-\pi-q})$. Thus, since $\left(s_{j-\pi-p} \not\rightsquigarrow^F s_{j-\pi-q} \right)[s_{j-\pi-p} \mapsto l_p, s_{j-\pi-q} \mapsto l_q] \in I_1$, it must be $s_{j-\pi-p} \not\rightsquigarrow^F s_{j-\pi-q} \in I$ and hence $s_{j-\pi-p} \not\rightsquigarrow^{F'}_{\omega} s_{j-\pi-q}$ with $F \subseteq F'$. By Lemma 7.1, we have that $s_{j-\pi-p} \not\rightsquigarrow^{F'}_{\omega} s_{j-\pi-q}$ entails $x \not\rightsquigarrow^{F'}_{\omega'} y$.

Regarding expression (7.6), for each $z \in \mathsf{dom}(\tau') = L'$ such that $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I_1 \subseteq \Pi(I)$, there exist $p \in [0, \pi]$ such that $z = l_p$; therefore $\rho'(z) = \rho'(l_p) = \rho(s_{j-\pi-p})$. Thus, since $\left(s_{j-\pi-p} \rightsquigarrow\!\!\!\!\!\!\diagup^F \right)[s_{j-\pi-p} \mapsto l_p] \in I_1$, it must be $s_{j-\pi-p} \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I$ and hence $s_{j-\pi-p} \rightsquigarrow\!\!\!\!\!\!\diagup^{F'}_{\omega}$ with $F \subseteq F'$. By Lemma 7.1, we have that $s_{j-\pi-p} \rightsquigarrow\!\!\!\!\!\!\diagup^{F'}_{\omega}$ entails $z \rightsquigarrow\!\!\!\!\!\!\diagup^{F'}_{\omega'}$. □

7.2.4 Soundness of Side Effect Arcs

Lemma 7.6 (Side Effect Arc)**.** *The normalized propagation rules for* side-effect arc *of Definition 5.3 are correct at method* return *void. That is let* $w \in [1, n]$ *and consider a side-effect arc form nodes* $\mathsf{C} = \boxed{\mathsf{call}\ \kappa_1.m \dots \kappa_n.m}$ *and* $\mathsf{E} = \boxed{\mathsf{exit@}\, m_w}$ *to node* $\mathsf{Q} = \boxed{\mathsf{ins}_q}$ *and its normalize propagation rule* $\Phi^{\#11}$*. Assume that* C*,* E *and* Q *have* static type *information* τ_c*,* τ_e *and* τ_q *respectively and let* d *the denotation of* m_w *i.e., a partial function from a state at its beginning to the corresponding state at its end. Then, for every* $I_c \in \mathsf{A}_{\tau_c}$ *and* $I_e \in \mathsf{A}_{\tau_e}$*, we have*

$$d\left((makescope\ m_w)\left(\gamma_{\tau_c}(I_c)\right)\right) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}\left(\Phi^{\#11}(I_c, I_e)\right).$$

Proof. Since $\gamma_\tau(\Pi(I)) = \gamma_\tau(\Phi(I))$ by Corollary 6.1.1, it is enough to prove

$$d\left((makescope\ m_w)\left(\gamma_{\tau_c}(I_c)\right)\right) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}\left(\Pi^{\#11}(I_c, I_e)\right).$$

Let $\omega_c \in \gamma_{\tau_c}(I_c)$ and $\omega_e \in \gamma_{\tau_e}(I_e)$. Consider two arbitrary abstract elements $I_c \in \mathsf{A}_{\tau_c}$ and $I_e \in \mathsf{A}_{\tau_e}$ and an arbitrary state $\omega_q = \langle \rho_q, \mu_q \rangle \in d\left((makescope\ m_w)(\sigma_c)\right) \cap \Xi_{\tau_q}$. We prove that $\omega_q \in \gamma_{\tau_q}(\Pi^{\#11}(I_c, I_e))$ i.e., by Definition 5.2 that

$$\text{for each } x \not\rightsquigarrow^{F} y \in \Pi^{\#11}(I_c, I_e) \text{ we have } x \not\rightsquigarrow^{F'}_{\omega_q} y \wedge F \subseteq F' \tag{7.7}$$

$$\text{for each } z \rightsquigarrow\!\!\!\!\!\!\diagup^{F} \in \Pi^{\#11}(I_c, I_e) \text{ we have } z \rightsquigarrow\!\!\!\!\!\!\diagup^{F'}_{\omega_q} \wedge F \subseteq F' \tag{7.8}$$

Note that, by the choice of ω_q and by Definition 5.2, for each $x_c \not\rightsquigarrow^{F} y_c \in I_c$ and $x_e \not\rightsquigarrow^{F} y_e \in I_e$ we obtain $x_c \not\rightsquigarrow^{F'_c}_{\omega_c} y_c$ with $F_c \subseteq F'_c$ and $x_e \not\rightsquigarrow^{F'_e}_{\omega_e} y_e$ with $F_e \subseteq F'_e$, respectively. Furthermore for each $z_c \rightsquigarrow\!\!\!\!\!\!\diagup^{F} \in I_c$ and $z_e \rightsquigarrow\!\!\!\!\!\!\diagup^{F_e} \in I_e$ we obtain $z_c \rightsquigarrow\!\!\!\!\!\!\diagup^{F'_c}_{\omega_c}$ with $F_c \subseteq F'_c$ and $z_e \rightsquigarrow\!\!\!\!\!\!\diagup^{F'_e}_{\omega_e}$ with $F_e \subseteq F'_e$, respectively.

We have $L_q = L_c$, $S_q = S_c \cup \{s_{j-\pi}\}$, $\mu_q \subseteq \mu_e$ and $\rho_q(s_{j-\pi}) = \rho_e(s_0)$. By Definition 6.2, $\Pi^{\#11}(I_c, I_e) = I_1 \cup I_2 \cup I_3 \cup I_4 \cup I_5$, where

$$I_1 = \left\{ a \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, a, b \in \mathsf{dom}(\tau_q) \wedge \langle a, b\rangle \notin \mathcal{MR}_{\tau_q} \right\}$$

$$I_2 = \left\{ \begin{array}{l} a \not\rightsquigarrow^{F_1} b, \\ a \rightsquigarrow \not\Diamond^{F_2} \in I_c \end{array} \middle| \begin{array}{l} a, b \in L_c \cup \{s_0, \ldots, s_{max-1}\} \wedge \\ \forall j - \pi \leq p \leq j, (a, s_p) \notin \mathcal{MS}_{\tau_c} \end{array} \right\}$$

$$I_3 = \left\{ \begin{array}{l} a \not\rightsquigarrow^{F_1} b, \\ a \rightsquigarrow \not\Diamond^{F_2} \in I_c \end{array} \middle| \begin{array}{l} a, b \in L \cup \{s_0, \ldots, s_{max-1}\} \wedge \\ \forall j - \pi \leq p < j \,\big|\, (a, s_p) \in \mathcal{MS}_{\tau}, \\ \quad \forall \boxed{ins} \text{ in } m_w \,\big|\, \boxed{ins = \mathsf{putfield}\ \kappa.f{:}\mathsf{t}} \vee \boxed{ins = \mathsf{call}\ m_1 \ldots m_k}, \\ \quad \left[\text{no } \boxed{\mathsf{store}\ \mathsf{l}_{p-j+\pi}} \text{ until } \boxed{ins} \text{ with type information } \tau^* \right] \wedge \\ \quad \text{if } \boxed{ins = \mathsf{putfield}\ \kappa.f{:}\mathsf{t}} \text{ linking } s_{j-2} \text{ to } s_{j-1}, \\ \qquad \langle l_{p-j+\pi}, s_{j-2}\rangle \notin \mathcal{MR}_{\tau *} \\ \quad \text{if } \boxed{ins = \mathsf{call}\ m_1 \ldots m_k} \text{ with actual params } s_k, \ldots, s_{k-\phi}, \\ \qquad \forall q \in [k, k-\phi], (l_{p-j+\pi}, s_q) \notin \mathcal{MS}_{\tau *} \end{array} \right\}$$

$$I_4 = \left\{ a \not\rightsquigarrow^{F} b \,\middle|\, \begin{array}{l} 1.\ a, b \in L_c \cup \{s_0, \ldots, s_{max-1}\} \wedge \langle a, b\rangle \in \mathcal{MR}_{\tau_q} \\ 2.\ \exists j - \pi \leq p_a, p_b < j \,\big|\, (a, s_{p_a}), (b, s_{p_b}) \in \mathcal{DA}_{\tau_c} \wedge \\ \quad \left[\text{no } \boxed{\mathsf{store}\ \mathsf{l}_{\mathsf{pa}-j+\pi}} \text{ nor } \boxed{\mathsf{store}\ \mathsf{l}_{\mathsf{p_b}-j+\pi}} \text{ occurs in } m_w \right] \\ 3.\ l_{p_a-j+\pi} \not\rightsquigarrow^{F} l_{p_b-j+\pi} \in I_q \end{array} \right\}$$

$$I_5 = \left\{ c \rightsquigarrow \not\Diamond^{F} \,\middle|\, \begin{array}{l} 1.\ c \in L_c \cup \{s_0, \ldots, s_{max-1}\} \\ 2.\ \exists j - \pi \leq p < j \,\big|\, (c, s_p) \in \mathcal{DA}_{\tau_c} \wedge \\ \quad \left[\text{no } \boxed{\mathsf{store}\ \mathsf{l}_{p-j+\pi}} \text{ occurs in } m_w \right] \\ 3.\ l_{p-j+\pi} \rightsquigarrow \not\Diamond^{F} \in I_q \end{array} \right\}$$

About expression (7.7), for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$ we distinguish the following cases:

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_1$: we have that $x, y \in \mathsf{dom}(\tau_q)$ and x cannot reaches y in τ_q i.e., $\rho_q(y) \notin \mathsf{L}_{\omega_q}(x)$. Therefore, by Definition 4.7, we obtain $x \not\rightsquigarrow^{F'}_{\omega_q} y$ for every $F' \subseteq \mathcal{F}$.

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_2$: we have that $x, y \in \mathsf{dom}(\tau_c)$ and, since $x \not\rightsquigarrow^{F} y \in I_c$, $x \not\rightsquigarrow^{F'}_{\omega_c} y$ with $F \subseteq F'$. Furthermore $\rho_c(x) = \rho_q(x), \rho_c(y) = \rho_q(y)$ and x does not share with any stack parameter passed to the method m_w. Therefore inside m_w no paths through objects in locations that share with $\rho_c(x)$ are modified and, hence, also after the execution of m_w the part of the heap reachable from $\rho_c(x)$ remains unchanged. Thus $\rho_q(x) \not\rightsquigarrow^{F'}_{\mu_q} \rho_q(y)$ i.e., $x \not\rightsquigarrow^{F'}_{\omega_q} y$.

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_3$: we have that $x, y \in \mathsf{dom}(\tau_c)$ and, since $x \not\rightsquigarrow^{F} y \in I_c$, $x \not\rightsquigarrow^{F'}_{\omega_c} y$ with $F \subseteq F'$. Let s_p a stack elements such that x may share with s_p at τ_c. This stack element, at the beginning of the method m_w, has been popped from the stack and placed in position $p - j - \pi$ of L.

  Furthermore, let $\boxed{\mathsf{putfield}\ \kappa.f{:}\mathsf{t}}$ occurred in a program point with type environment τ^* inside m_w, if no $\boxed{\mathsf{store}\ \mathsf{l}_{p-j+\pi}}$ occurred until this point, we can state $\rho^*(l_{p-j+\pi}) = \rho_c(s_p)$ and hence $\rho_c(x)$ may reach $\rho^*(l_{p-j+\pi})$. The $\boxed{\mathsf{putfield}\ \kappa.f{:}\mathsf{t}}$ at τ^* does not affect $\rho^*(l_{p-j+\pi})$ because $l_{p-j+\pi}$ cannot surely reach the variable s_{j-2} below the top of the stack i.e., $\rho^*(s_{j-2}) \notin \mathsf{L}_{\omega^*}(l_{p_y-j+\pi})$. Therefore, also if $\rho_c(x)$ reaches $\rho^*(l_{p-j+\pi})$ in ω^*, it cannot reach $\rho(s_{j-2})$ and

hence, by Lemma 7.1, we can assert that also $\rho_c(x) \not\rightsquigarrow^{F'}_{\mu_*} \rho_c(y)$ with μ_* the heap after the putfield $\kappa.f$:t.

By Definition of I_8, this reasoning can be done with all the stack variables such that $\langle x, s_p \rangle \in \mathcal{MR}_{\tau_c}$ and for all the putfield $\kappa.f$:t in m_w. Moreover , by bytecode semantics in Figure 3.1, the putfield $\kappa.f$:t is the only instruction that can modify some paths in memory μ, we have, at the program point E, $\rho_c(x) \not\rightsquigarrow^{F'}_{\mu_e} \rho_c(y)$. Hence, since $\rho_c(x) = \rho_q(x)$, $\rho_c(y) = \rho_q(y)$ and $\mu_e = \mu_q$, we can conclude $x \not\rightsquigarrow^{F'}_{\omega_q} y$.

We can do the same reasoning with every call $m_1 \dots m_k$ occurring in m_w with the difference that this call cannot affect $\rho^*(l_{p-j+\pi})$ because $l_{p-j+\pi}$ does not share with any actual parameter $s_k, \dots, s_{k-\phi}$ of that call $m_1 \dots m_k$ and, hence, also after the execution of that method the part of the heap reachable from $\rho^*(l_{p-j+\pi})$ remains unchanged.

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I_4}$: we have that $x, y \in \mathsf{dom}(\tau_q)$ and x may reach y in τ_q. Furthermore x and y are alias, in τ_c, with two stack elements s_{p_x} and s_{p_y}, respectively. These two elements, at the beginning of the method m_w, have been popped from the stack and placed in position $p_x - j - \pi$ and $p_y - j - \pi$ of L. If also neither store $\mathsf{l}_{\mathsf{px}-\mathsf{j}+\pi}$ nor store $\mathsf{l}_{\mathsf{py}-\mathsf{j}+\pi}$ occurs in m_w, we can assert:

$$\rho_q(x) \overset{\text{by Def. 3.8}}{=} \rho_c(x) \overset{\langle x, s_{p_x} \rangle \in \mathcal{DA}_{\tau_c}}{=} \rho_c(s_{p_x}) \overset{\text{no store } \mathsf{l}_{\mathsf{px}-\mathsf{j}+\pi} \text{ in } m_w}{=} \rho_e(l_{p_x - j + \pi})$$

and,

$$\rho_q(y) \overset{\text{by Def. 3.8}}{=} \rho_c(y) \overset{\langle s_{p_y}, y \rangle \in \mathcal{DA}_{\tau_c}}{=} \rho_c(s_{p_y}) \overset{\text{no store } \mathsf{l}_{\mathsf{py}-\mathsf{j}+\pi} \text{ in } m_w}{=} \rho_e(l_{p_y - j + \pi})$$

Moreover, since $l_{p_x-j+\pi} \not\rightsquigarrow^{F} l_{p_y-j+\pi} \in I_e$, we have $l_{p_x-j+\pi} \not\rightsquigarrow^{F'}_{\omega_e} l_{p_y-j+\pi}$ with $F \subseteq F'$ and then also $\rho_c(x) \not\rightsquigarrow^{F'}_{\mu_e} \rho_c(y)$. Hence, since $\mu_q = \mu_e$, by Lemma 7.1, we obtain $x \not\rightsquigarrow^{F'}_{\omega_q} y$.

Regarding expression (7.8), for each $z \not\rightsquigarrow^{\mathtt{F}} \in \Pi(I)$ we distinguish the following cases:

- $\mathtt{z} \not\rightsquigarrow^{\mathtt{F}} \in \mathtt{I_2}$: we have that $z \in \mathsf{dom}(\tau_c)$ and, since $z \not\rightsquigarrow^{\mathtt{F}} \in I_c$, $z \not\rightsquigarrow^{F'}_{\omega_c}$ with $F \subseteq F'$. Furthermore, z does not share with any stack parameter that is passed to the method m_w. Therefore inside m_w no paths through objects in locations that share with $\rho_c(z)$ are modified and, hence, also after the execution of m_w the part of the heap reachable from $\rho_c(z)$ remains unchanged. Thus $\rho_q(z) \not\rightsquigarrow^{F'}_{\mu_q}$ i.e., $z \not\rightsquigarrow^{F'}_{\omega_q}$.

- $\mathtt{z} \not\rightsquigarrow^{\mathtt{F}} \in \mathtt{I_3}$: we have that $z \in \mathsf{dom}(\tau_c)$ and, since $z \not\rightsquigarrow^{\mathtt{F}} \in I_c$, $z \not\rightsquigarrow^{F'}_{\omega_c}$ with $F \subseteq F'$. Let s_p a stack elements such that z may share with s_p at τ_c. This stack element, at the beginning of the method m_w, has been popped from the stack and placed in position $p - j - \pi$ of L.

  Furthermore, let putfield $\kappa.f$:t occurred in a program point with type environment τ^* inside m_w, if no store $\mathsf{l}_{\mathsf{p}-\mathsf{j}+\pi}$ occurred until this point, we can state $\rho^*(l_{p-j+\pi}) = \rho_c(s_p)$ and hence $\rho_c(x)$ may reach $\rho^*(l_{p-j+\pi})$. The putfield $\kappa.f$:t at τ^* does not affect $\rho^*(l_{p-j+\pi})$ because $l_{p-j+\pi}$ cannot surely reach the variable s_{j-2} below the top of the stack i.e., $\rho^*(s_{j-2}) \notin \mathsf{L}_{\omega^*}(l_{p_y-j+\pi})$. Therefore, also if $\rho_c(x)$ reaches $\rho^*(l_{p-j+\pi})$ in ω^*, it cannot reach $\rho(s_{j-2})$ and hence, by Lemma 7.1, we can assert that also $\rho_c(z) \not\rightsquigarrow^{F'}_{\mu_*}$ with μ_* the heap after the putfield $\kappa.f$:t.

  By Definition of I_3, this reasoning can be done with all the stack variables such that $\langle x, s_p \rangle \in \mathcal{MR}_{\tau_c}$ and for all the putfield $\kappa.f$:t in m_w. Moreover , by bytecode semantics in

Figure 3.1, the $\boxed{\text{putfield } \kappa.f{:}\mathsf{t}}$ is the only instruction that can modify some paths in memory μ, we have, at the program point E, $\rho_c(z) \rightsquigarrow_{\mu_e}^{\not\partial F'}$. Hence, since $\rho_c(x) = \rho_q(x)$, $\rho_c(y) = \rho_q(y)$ and $\mu_e = \mu_q$, we can conclude $z \rightsquigarrow_{\omega_q}^{\not\partial F'} y$.

We can do the same reasoning with every $\boxed{\text{call } m_1 \dots m_k}$ occurring in m_w with the difference that this call cannot affect $\rho * (l_{p-j+\pi})$ because $l_{p-j+\pi}$ does not share with any actual parameter $s_k, \dots, s_{k-\phi}$ of that call $m_1 \dots m_k$ and, hence, also after the execution of that method the part of the heap reachable from $\rho^*(l_{p-j+\pi})$ remains unchanged.

- $\mathtt{z} \rightsquigarrow^{\not\partial \mathtt{F}} \in \mathtt{I_5}$: we have that $z \in \mathsf{dom}(\tau_q)$. Furthermore z is alias, in τ_c, with a stack element s_p which, at the beginning of the method m_w, has been popped from the stack and placed in position $p - j - \pi$ of L. If also no $\boxed{\text{store } \mathsf{l}_{p-j+\pi}}$ occurs in m_w, we can assert:

$$\rho_q(z) \overset{\text{by Def. 3.8}}{=} \rho_c(z) \overset{\langle s_p, z\rangle \in \mathcal{DA}_{\tau_c}}{=} \rho_c(s_p) \overset{\text{no store } \mathsf{l}_{p-j+\pi} \text{ in } m_w}{=} \rho_e(l_{p-j+\pi})$$

Moreover, since $l_{p-j+\pi} \rightsquigarrow^{\not\partial F} \in I_e$, we have $l_{p-j+\pi} \rightsquigarrow_{\omega_e}^{\not\partial F'}$ with $F \subseteq F'$ and then also $\rho_c(z) \rightsquigarrow_{\mu_e}^{\not\partial F'}$. Hence, since $\mu_q = \mu_e$, by Lemma 7.1, we obtain $z \rightsquigarrow_{\omega_q}^{\not\partial F'}$.

□

7.2.5 Soundness of Return Value Arcs

Lemma 7.7 (Return Value Arc)**.** *The normalized propagation rules for* return value arc *of Definition 5.3 are correct at method* return t*. That is let* $w \in [1, n]$ *and consider a return value and form nodes* $\mathsf{C} = \boxed{\text{call } \kappa_1.m \dots \kappa_n.m}$ *and* $\mathsf{E} = \boxed{\text{exit@}m_w}$ *to node* $\mathsf{Q} = \boxed{\text{ins}_q}$ *and its normalize propagation rule* $\Phi^{\#12}$*. Assume that* C*,* E *and* Q *have* static type *information* τ_c*,* τ_e *and* τ_q *respectively and let* d *the denotation of* m_w *i.e., a partial function from a state at its beginning to the corresponding state at its end. Then, for every* $I_c \in \mathsf{A}_{\tau_c}$ *and* $I_e \in \mathsf{A}_{\tau_e}$*, we have*

$$d\left((makescope\ m_w)\left(\gamma_{\tau_c}(I_c)\right)\right) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}\left(\Phi^{\#12}(I_c, I_e)\right).$$

Proof. Since $\gamma_\tau(\Pi(I)) = \gamma_\tau(\Phi(I))$ by Corollary 6.1.1, it is enough to prove

$$d\left((makescope\ m_w)\left(\gamma_{\tau_c}(I_c)\right)\right) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}\left(\Pi^{\#12}(I_c, I_e)\right).$$

Let $\omega_c \in \gamma_{\tau_c}(I_c)$ and $\omega_e \in \gamma_{\tau_e}(I_e)$. Consider two arbitrary abstract elements $I_c \in \mathsf{A}_{\tau_c}$ and $I_e \in \mathsf{A}_{\tau_e}$ and an arbitrary state $\omega_q = \langle \rho_q, \mu_q \rangle \in d\left((makescope\ m_w)(\sigma_c)\right) \cap \Xi_{\tau_q}$. We prove that $\omega_q \in \gamma_{\tau_q}(\Pi^{\#12}(I_c, I_e))$ i.e., by Definition 5.2 that

$$\text{for each } x \not\rightsquigarrow^F y \in \Pi^{\#12}(I_c, I_e) \text{ we have } x \not\rightsquigarrow_{\omega_q}^{F'} y \wedge F \subseteq F' \tag{7.9}$$

$$\text{for each } z \rightsquigarrow^{\not\partial F} \in \Pi^{\#12}(I_c, I_e) \text{ we have } z \rightsquigarrow_{\omega_q}^{\not\partial F'} \wedge F \subseteq F' \tag{7.10}$$

Note that, by the choice of ω_q and by Definition 5.2, for each $x_c \not\rightsquigarrow^F y_c \in I_c$ and $x_e \not\rightsquigarrow^F y_e \in I_e$ we obtain $x_c \not\rightsquigarrow_{\omega_c}^{F'_c} y_c$ with $F_c \subseteq F'_c$ and $x_e \not\rightsquigarrow_{\omega_e}^{F'_e} y_e$ with $F_e \subseteq F'_e$, respectively. Furthermore for each $z_c \rightsquigarrow^{\not\partial F} \in I_c$ and $z_e \rightsquigarrow^{\not\partial F_e} \in I_e$ we obtain $z_c \rightsquigarrow_{\omega_c}^{\not\partial F'_c}$ with $F_c \subseteq F'_c$ and $z_e \rightsquigarrow_{\omega_e}^{\not\partial F'_e}$ with $F_e \subseteq F'_e$, respectively.

We have $L_q = L_c$, $S_q = S_c \cup \{s_{j-\pi}\}$, $\mu_q \subseteq \mu_e$ and $\rho_q(s_{j-\pi}) = \rho_e(s_0)$. By Definition 6.2, $\Pi^{\#12}(I_c, I_e) = I_1 \cup I_2 \cup I_3 \cup I_4 \cup I_5 \cup I_6$, where

$$
\begin{aligned}
I_1 &= \left\{ s_{j-\pi} \not\rightsquigarrow^{F_1} s_{j-\pi}, s_{j-\pi} \rightsquigarrow\!\!\!\!\!\diagup^{F_2} \;\middle|\; s_0 \not\rightsquigarrow^{F_1} s_0, s_0 \rightsquigarrow\!\!\!\!\!\diagup^{F_2} \in I_e \right\} \\
I_2 &= \left\{ a \not\rightsquigarrow^{\mathcal{F}} s_{j-\pi} \,\middle|\, a \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\} \wedge \langle a, s_{j-\pi} \rangle \notin \mathcal{MR}_{\tau_q} \right\} \\
I_3 &= \left\{ a \not\rightsquigarrow^{F} s_{j-\pi} \;\middle|\; \begin{array}{l} 1.\ a \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\} \wedge \langle a, s_{j-\pi} \rangle \in \mathcal{MR}_{\tau_q} \\ 2.\ \exists j - \pi \leq p < j \,|\, (s_p, a) \in \mathcal{DA}_{\tau_c} \wedge \\ \quad \left[\text{no } \boxed{\text{store } \mathsf{l}_{p-j+\pi}} \text{ occurs in } m_w \right] \\ 3.\ l_{p-j+\pi} \not\rightsquigarrow^{F} s_0 \in I_e \end{array} \right\} \\
I_4 &= \left\{ s_{j-\pi} \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, b \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\} \wedge \langle s_{j-\pi}, b \rangle \notin \mathcal{MR}_{\tau_q} \right\} \\
I_5 &= \left\{ s_{j-\pi} \not\rightsquigarrow^{F} b \;\middle|\; \begin{array}{l} 1.\ b \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\} \wedge \langle s_{j-\pi}, b \rangle \in \mathcal{MR}_{\tau_q} \\ 2.\ \exists j - \pi \leq p < j \,|\, (s_p, b) \in \mathcal{DA}_{\tau_c} \wedge \\ \quad \left[\text{no } \boxed{\text{store } \mathsf{l}_{p-j+\pi}} \text{ occurs in } m_w \right] \\ 3.\ s_0 \not\rightsquigarrow^{F} l_{p-j+\pi} \in I_e \end{array} \right\} \\
I_6 &= \Pi^{\#\#11}(I_c, I_e, j - \pi)
\end{aligned}
$$

About expression (7.9), for each $x \not\rightsquigarrow^{F} y \in \Pi(I)$ we distinguish the following cases:

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_1$: we have that $x, y = s_{j-\pi}$ and $s_0 \not\rightsquigarrow^{F} s_0 \in I_e$. Therefore $s_0 \not\rightsquigarrow^{F'}_{\omega_e} s_0$ with $F \subseteq F'$ and, since $\rho_q(x) = \rho_q(y) = \rho_e(s_0)$ and $\mu_q = \mu_e$, by Lemma 7.1, we can conclude $x \not\rightsquigarrow^{F'}_{\omega_q} y$.

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_2$: we have that $x \in \mathsf{dom}(\tau_q)$, $y = s_{j-\pi}$ and x cannot reaches y in τ_q i.e., $\rho_q(y) \notin \mathsf{L}_{\omega_q}(x)$. Therefore, by Definition 4.7, we obtain $x \not\rightsquigarrow^{F'}_{\omega_q} y$ for every $F' \subseteq \mathcal{F}$.

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_3$: we have that $x \in \mathsf{dom}(\tau_q)$, $y = s_{j-\pi}$ and x may reach y in τ_q. Furthermore x is alias, in τ_c, with a stack element s_p which, at the beginning of the method m_w, has been popped from the stack and placed in position $p - j - \pi$ of L. If also no $\boxed{\text{store } \mathsf{l}_{p-j+\pi}}$ occurs in m_w, we can assert:

$$
\rho_q(x) \overset{\text{by Def. 3.8}}{=} \rho_c(x) \overset{\langle x, s_{j-\pi} \rangle \in \mathcal{DA}_{\tau_c}}{=} \rho_c(s_p) \overset{\text{no store } \mathsf{l}_{p-j+\pi} \text{ in } m_w}{=} \rho_e(l_{p-j+\pi})
$$

  Moreover, since $l_{p-j+\pi} \not\rightsquigarrow^{F} s_0 \in I_e$, we have $l_{p-j+\pi} \not\rightsquigarrow^{F'}_{\omega_e} s_0$ with $F \subseteq F'$ and then also $\rho_c(x) \not\rightsquigarrow^{F'}_{\mu_e} \rho_c(y)$. Hence, since $\mu_q = \mu_e$, by Lemma 7.1, we obtain $x \not\rightsquigarrow^{F'}_{\omega_q} y$.

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_4$: we have that $x = s_{j-\pi}$, $y \in \mathsf{dom}(\tau_q)$ and x cannot reaches y in τ_q i.e., $\rho_q(y) \notin \mathsf{L}_{\omega_q}(x)$. Therefore, by Definition 4.7, we obtain $x \not\rightsquigarrow^{F'}_{\omega_q} y$ for every $F' \subseteq \mathcal{F}$.

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_5$: we have that $x = s_{j-\pi}$, $y \in \mathsf{dom}(\tau_q)$ and x may reach y in τ_q. Furthermore y is alias, in τ_c, with a stack element s_p which, at the beginning of the method m_w, has been popped from the stack and placed in position $p - j - \pi$ of L. If also no $\boxed{\text{store } \mathsf{l}_{p-j+\pi}}$ occurs in m_w, we can assert:

$$
\rho_q(y) \overset{\text{by Def. 3.8}}{=} \rho_c(y) \overset{\langle s_{j-\pi}, y \rangle \in \mathcal{DA}_{\tau_c}}{=} \rho_c(s_p) \overset{\text{no store } \mathsf{l}_{p-j+\pi} \text{ in } m_w}{=} \rho_e(l_{p-j+\pi})
$$

  Moreover, since $s_0 \not\rightsquigarrow^{F} l_{p-j+\pi} \in I_e$, we have $s_0 \not\rightsquigarrow^{F'}_{\omega_e} l_{p-j+\pi}$ with $F \subseteq F'$ and then also $\rho_c(x) \not\rightsquigarrow^{F'}_{\mu_e} \rho_c(y)$. Hence, since $\mu_q = \mu_e$, by Lemma 7.1, we obtain $x \not\rightsquigarrow^{F'}_{\omega_q} y$.

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_6$: see Proof of unreachability tokens in Lemma 7.6.

Regarding expression (7.10), for each $z \rightsquigarrow\!\!\!\!\!\!\diagup^{F} \in \Pi(I)$ we distinguish the following cases:

- $\mathtt{z} \rightsquigarrow\!\!\!\!\!\!\diagup^{\mathtt{F}} \in \mathtt{I}_1$: we have that $z = s_{j-\pi}$ and $s_0 \rightsquigarrow\!\!\!\!\!\!\diagup^{F} \in I_e$. Therefore $s_0 \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega_e}^{F'}$ with $F \subseteq F'$ and, since $\rho_q(x) = \rho_q(y) = \rho_e(s_0)$ and $\mu_q = \mu_e$, by Lemma 7.1, we can conclude $z \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega_q}^{F'}$.
- $\mathtt{z} \rightsquigarrow\!\!\!\!\!\!\diagup^{\mathtt{F}} \in \mathtt{I}_6$: see Proof of non-cyclicity tokens in Lemma 7.6.

□

7.2.6 Soundness of Exceptional Arcs

Lemma 7.8 (Exceptional Arcs)**.** *The normalized propagation rules for* exceptional arcs *of Definition 5.3 are correct. That is, consider an exceptional arc from a bytecode* ins *distinct from* call *and its normalized propagation rule* Φ*. Assume that* ins *has* static type *information* τ *at its beginning and* τ' *immediately after its execution. Then, for every* $I \in \mathsf{A}_\tau$ *we have*

$$\mathit{ins}\,(\gamma_\tau\,(I)) \cap \Xi_{\tau'} \subseteq \gamma_{\tau'}\,(\Phi\,(I))$$

(we recall that ins *is the semantic of* ins *bytecode).*

Proof. Since $\gamma_{\tau'}\,(\Pi\,(I)) = \gamma_{\tau'}\,(\Phi\,(I))$ by Corollary 6.1.1, it is enough to prove

$$\mathit{ins}\,(\gamma_\tau\,(I)) \cap \Xi_{\tau'} \subseteq \gamma_{\tau'}\,(\Pi\,(I))\,.$$

Let $\mathsf{dom}(\tau) = L \cup S$ contains i local variables $L = \{l_0, \ldots, l_{i-1}\}$ and j stack elements $S = \{s_0, \ldots, s_{j-1}\}$. Furthermore let $\mathsf{dom}(\tau') = L' \cup S'$, where L' and S' are the local and stack variables after the execution of ins. Consider an arbitrary abstract element $I \in \mathsf{A}_\tau$ and an arbitrary state $\omega' = \langle \rho', \mu' \rangle \in \mathit{ins}\,(\gamma_\tau\,(I)) \cap \Xi_{\tau'}$. We prove that $\omega' \in \gamma_{\tau'}\,(\Pi\,(I))$ i.e., by Definition 5.2 that

$$\text{for each } x \not\rightsquigarrow^{F} y \in \Pi(I) \text{ we have } x \not\rightsquigarrow_{\omega'}^{F'} y \wedge F \subseteq F' \tag{7.11}$$

$$\text{for each } z \rightsquigarrow\!\!\!\!\!\!\diagup^{F} \in \Pi(I) \text{ we have } z \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega'}^{F'} \wedge F \subseteq F' \tag{7.12}$$

Note that, by the choice of ω', there exists $\omega = \langle \rho, \mu \rangle \in \gamma_\tau(I)$ such that $\omega' = \mathit{ins}(\omega)$ and, by Definition 5.2, for each $x \not\rightsquigarrow^{F} y \in I$ we obtain $x \not\rightsquigarrow_{\omega}^{F'} y$ with $F \subseteq F'$ and, respectively, for each $z \rightsquigarrow\!\!\!\!\!\!\diagup^{F} \in I$ we obtain $z \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega}^{F'}$ with $F \subseteq F'$. We analyze different propagation rules corresponding to different types of exceptional arcs.

ins = throw κ. This proof is analogous to the proof of Lemma 7.4 for throw κ, when $\rho'(s_0) = \ell$, where ℓ is a fresh location.

ins = new κ, getfield $\kappa.f$:t, putfield $\kappa.f$:t. We have $L' = L$ and $S' = \{s_0\}$. Furthermore, for every $v \in \mathsf{dom}(\tau') \setminus \{s_0\}$, $\rho(v) = \rho'(v)$, while $\rho'(s_0) = \ell \in \mathbb{L}$, where ℓ is a fresh location and $\mu' = \mu[\ell \mapsto \mathsf{o}]$ with o a new instance of `OutOfMemoryException` (in case of new κ) or of `NullPointerException` (in case of getfield $\kappa.f$:t or putfield $\kappa.f$:t) containing only fresh locations i.e., $\mathsf{L}_{\mu'}(\ell) \cap \mathsf{dom}(\mu) = \emptyset$. By Definition 6.2,

$$\Pi(I) \;=\; \overbrace{\left\{ a \not\rightsquigarrow^{F_{ab}} b,\, c \rightsquigarrow\!\!\!\!\!\!\diagup^{F_c} \in I \,\middle|\, a, b, c \notin \{s_0, \ldots, s_{j-1}\} \right\}}^{I_1} \cup \underbrace{\left\{ a \not\rightsquigarrow^{\mathcal{F}} s_0,\, s_0 \not\rightsquigarrow^{\mathcal{F}} a \,|\, a \in L \right\} \cup \left\{ s_0 \not\rightsquigarrow^{\mathcal{F}} s_0,\, s_0 \rightsquigarrow\!\!\!\!\!\!\diagup^{\mathcal{F}} \right\}}_{I_2}$$

About expression (7.11) we distinguish the following cases:

- $\mathtt{x}, \mathtt{y} \neq \mathtt{s_0}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$, we have $x \not\rightsquigarrow^F y \in I_1$ and, hence, $x \not\rightsquigarrow^F y \in I$. Therefore $x \not\rightsquigarrow_\omega^{F'} y$ with $F \subseteq F'$. Since for every $\ell \in \mathsf{dom}(\mu), \mu'(\ell) = \mu(\ell)$, we obtain, by Lemma 7.1, that $x \not\rightsquigarrow_\omega^{F'} y$ entails $x \not\rightsquigarrow_{\omega'}^{F'} y$.
- $\mathtt{x} = \mathtt{s_0}, \mathtt{y} \neq \mathtt{s_0}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$ we have $s_0 \not\rightsquigarrow^{\mathcal{F}} y \in I_2$ and, hence, $\rho'(x) = \rho'(s_0) = \ell$ and $\rho(y) \in \mathsf{dom}(\mu)$. Furthermore, as ℓ is fresh, $\mathsf{L}_{\mu'}(\rho'(x)) \cap \mathsf{dom}(\mu) = \emptyset$. Therefore for each $\ell \in \mathsf{dom}(\mu)$, ℓ certainly does not reaches $\rho'(x)$ and vice-versa. Since $\rho'(x) = \ell$ is fresh, by Definition 4.7, we can assert $x \not\rightsquigarrow_{\omega'}^{\mathcal{F}} w$, for each $w \in \mathsf{dom}(\tau') \setminus \{s_0\}$; then also for y.
- $\mathtt{x} \neq \mathtt{s_0}, \mathtt{y} = \mathtt{s_0}$: for each $x \not\rightsquigarrow^F y \in \Pi(I)$ we have $x \not\rightsquigarrow^{\mathcal{F}} s_0 \in I_2$ and, hence, $\rho'(x) \in \mathsf{dom}(\mu)$ and $\rho(y) = \rho'(s_0) = \ell$. Furthermore, as ℓ is fresh, $\mathsf{L}_{\mu'}(\rho'(x)) \cap \mathsf{dom}(\mu) = \emptyset$. Therefore for each $\ell \in \mathsf{dom}(\mu)$, ℓ certainly does not reaches $\rho'(y)$ and vice-versa. Since $\rho'(y) = \ell$ is fresh, by Definition 4.7, we can assert $w \not\rightsquigarrow_{\omega'}^{\mathcal{F}} y$, for each $w \in \mathsf{dom}(\tau') \setminus \{s_0\}$; then also for x.
- $\mathtt{x} = \mathtt{y} = \mathtt{s_0}$: by $x \not\rightsquigarrow^F y \in \Pi(I)$ we obtain $s_0 \not\rightsquigarrow^{\mathcal{F}} s_0 \in I_2$. Since $\rho'(x) = \rho'(y) = \ell$ is fresh, by Definition 4.7 we can assert $s_0 \not\rightsquigarrow_{\omega'}^{\mathcal{F}} s_0 = x \not\rightsquigarrow_{\omega'}^{\mathcal{F}} y$.

Regarding expression (7.12), we distinguish the following cases:

- $\mathtt{z} \neq \mathtt{s_0}$: for each $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in \Pi(I)$ we have $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I_1$ and, hence, $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in I$. Therefore $z \rightsquigarrow\!\!\!\!\!\!\diagup_\omega^{F'}$ with $F \subseteq F'$. Since for every $\ell \in \mathsf{dom}(\mu), \mu'(\ell) = \mu(\ell)$, we obtain, by Lemma 7.1, that $z \rightsquigarrow\!\!\!\!\!\!\diagup_\omega^{F'}$ entails $z \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega'}^{F'}$.
- $\mathtt{z} = \mathtt{s_0}$: by $z \rightsquigarrow\!\!\!\!\!\!\diagup^F \in \Pi(I)$ we obtain $s_0 \rightsquigarrow\!\!\!\!\!\!\diagup^{\mathcal{F}} \in I_2$. Since $\rho'(s_0) = \ell$ is fresh, by Definition 4.7, we can assert $s_0 \rightsquigarrow\!\!\!\!\!\!\diagup_{\omega'}^{\mathcal{F}}$.

□

Lemma 7.9 (Exceptional Arc of the call)**.** *The normalized propagation rule for* exceptional arcs *of the* call *of Definition 5.3 are correct when a method throw an exception. That is let* $w \in [1, n]$ *and consider a multi-exceptional arc form nodes* $\mathsf{C} = \boxed{\mathsf{call}\ \kappa_1.m \ldots \kappa_n.m}$ *and* $\mathsf{E} = \boxed{\mathsf{exception@}\, m_w}$ *to node* $\mathsf{Q} = \boxed{\mathsf{catch}}$ *and its normalize propagation rule* $\Phi^{\#15}$*. Assume that* C, E *and* Q *have* static type *information* τ_c, τ_e *and* τ_q *respectively and let* d *the denotation of* m_w *i.e., a partial function from a state at its beginning to the corresponding state at its end. Then, for every* $I_c \in \mathsf{A}_{\tau_c}$ *and* $I_e \in \mathsf{A}_{\tau_e}$*, we have*

$$d\left((makescope\ m_w)\left(\gamma_{\tau_c}(I_c)\right)\right) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}\left(\Phi^{\#15}(I_c, I_e)\right).$$

Proof. Since $\gamma_\tau(\Pi(I)) = \gamma_\tau(\Phi(I))$ by Corollary 6.1.1, it is enough to prove

$$d\left((makescope\ m_w)\left(\gamma_{\tau_c}(I_c)\right)\right) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}\left(\Pi^{\#15}(I_c, I_e)\right).$$

Let $\omega_c = \langle\langle l_c \parallel v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0\rangle, \mu_c\rangle \in \gamma_{\tau_c}(I_c)$ and $\omega_e = \underline{\langle\langle l_e \parallel \ell\rangle, \mu_e\rangle} \in \gamma_{\tau_e}(I_e)$. Consider two arbitrary abstract elements $I_c \in \mathsf{A}_{\tau_c}$ and $I_e \in \mathsf{A}_{\tau_e}$ and a state $\underline{\omega_q = d((makescope\ m_w)}$ $(\omega_c))\cap\Xi_{\tau_q}$. This state must have the form $\omega_q = \underline{\langle\langle l_c \parallel \ell\rangle, \mu_e\rangle}$. We prove that $\omega_q \in \gamma_{\tau_q}(\Pi^{\#15}(I_c, I_e))$ i.e., by Definition 5.2 that

$$\text{for each } x \not\rightsquigarrow^F y \in \Pi^{\#15}(I_c, I_e) \text{ we have } x \not\rightsquigarrow_{\omega_q}^{F'} y \wedge F \subseteq F' \tag{7.13}$$

$$\text{for each } z \rightsquigarrow^{\not\Delta F} \in \Pi^{\#15}(I_c, I_e) \text{ we have } z \rightsquigarrow^{\not\Delta F'}_{\omega_q} \wedge F \subseteq F' \tag{7.14}$$

Note that, by the choice of ω_q and by Definition 5.2, for each $x_c \not\rightsquigarrow^F y_c \in I_c$ and $x_e \not\rightsquigarrow^F y_e \in I_e$ we obtain $x_c \not\rightsquigarrow^{F'_c}_{\omega_c} y_c$ with $F_c \subseteq F'_c$ and $x_e \not\rightsquigarrow^{F'_e}_{\omega_e} y_e$ with $F_e \subseteq F'_e$, respectively. Furthermore for each $z_c \rightsquigarrow^{\not\Delta F} \in I_c$ and $z_e \rightsquigarrow^{\not\Delta F_e} \in I_e$ we obtain $z_c \rightsquigarrow^{\not\Delta F'_c}_{\omega_c}$ with $F_c \subseteq F'_c$ and $z_e \rightsquigarrow^{\not\Delta F'_e}_{\omega_e}$ with $F_e \subseteq F'_e$, respectively.

We have $L_q = L_c$, $S_q = s_0$, $\mu_q = \mu_e$ (since only the location pointed by s_0 is created) and $\rho_q(s_0) = \rho_e(s_0)$. By Definition 6.2, $\Pi^{\#15}(I_c) = I_1 \cup I_2 \cup I_3 \cup I_4 \cup I_5$, where

$$\begin{aligned}
I_1 &= \Pi^{\#11}(I_c, I_e, 0) \\
I_2 &= \left\{ a \not\rightsquigarrow^{\mathcal{F}} s_0 \,\middle|\, a \in \mathsf{dom}(\tau_q) \setminus \{s_0\} \wedge \langle a, s_0 \rangle \notin \mathcal{MR}_{\tau_q} \right\} \\
I_3 &= \left\{ s_0 \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, b \in \mathsf{dom}(\tau_q) \setminus \{s_0\} \wedge \langle s_0, b \rangle \notin \mathcal{MR}_{\tau_q} \right\} \\
I_4 &= \left\{ s_0 \not\rightsquigarrow^{\mathcal{F}} s_0 \right\} \\
I_5 &= \left\{ s_0 \rightsquigarrow^{\not\Delta \mathcal{F}} \,\middle|\, \nexists a \in \mathsf{dom}(\tau_q) \setminus \{s_0\} \,\middle|\, \langle s_0, a \rangle \in \mathcal{MR}_{\tau_q} \right\}
\end{aligned}$$

About expression (7.13), for each $x \not\rightsquigarrow^F y \in \Pi(I)$ we distinguish the following cases:

- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_1$: see Proof of unreachability tokens in Lemma 7.6.
- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_2$: we have $x \in \mathsf{dom}(\tau_q)$ and $y = s_0$ and x cannot reaches s_0 in τ_q i.e., $\rho_q(s_0) \notin \mathsf{L}_{\omega_q}(x)$. Therefore, by Definition 4.7, we obtain $x \not\rightsquigarrow^{F'}_{\omega_q} y$ for every $F' \subseteq \mathcal{F}$.
- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_3$: we have $x = s_0$ and $y \in \mathsf{dom}(\tau_q)$ and s_0 cannot reaches y in τ_q i.e., $\rho_q(y) \notin \mathsf{L}_{\omega_q}(s_0)$. Therefore, by Definition 4.7, we obtain $x \not\rightsquigarrow^{F'}_{\omega_q} y$ for every $F' \subseteq \mathcal{F}$.
- $\mathtt{x} \not\rightsquigarrow^{\mathtt{F}} \mathtt{y} \in \mathtt{I}_4$: we have $x \not\rightsquigarrow^{\mathcal{F}} y \in I_4$, $x, y = s_0$ and hence $\rho_q(x) = \rho_q(y)$. Since the thrown object pointed by s_0 cannot be `null`, by Definition 4.7, we can state $s_0 \not\rightsquigarrow^{\mathcal{F}}_{\omega_q} s_0$ i.e., $x \not\rightsquigarrow^{\mathcal{F}}_{\omega_q} y$.

About expression (7.14), for each $x \not\rightsquigarrow^F y \in \Pi(I)$ we distinguish the following cases:

- $\mathtt{z} \rightsquigarrow^{\not\Delta \mathtt{F}} \in \mathtt{I}_1$: see Proof of non-cyclicity tokens in Lemma 7.6.
- $\mathtt{z} \rightsquigarrow^{\not\Delta \mathtt{F}} \in \mathtt{I}_5$: we have $z \rightsquigarrow^{\not\Delta \mathcal{F}} \in I_5$, $z = s_0$ and s_0 cannot reach any variable in τ_q apart itself i.e., for every $a \in \mathsf{dom}(\tau_q) \setminus \{s_0\}$ we have $\rho_q(a) \notin \mathsf{L}_{\omega_q}(s_0)$. Therefore, by Definition 4.8, we obtain $z \rightsquigarrow^{\not\Delta F'}_{\omega_q}$ for every $F' \subseteq \mathcal{F}$.

□

7.3 Soundness of the Analysis

Theorem 7.1 (Soundness)**.** *Let "*$\langle b_{first(\mathtt{main})} \parallel \xi \rangle \Rightarrow^* \langle \boxed{\begin{matrix}\text{ins} \\ \text{rest}\end{matrix}} \begin{matrix}\rightarrow b_1 \\ \cdots \\ \rightarrow b_m\end{matrix} \parallel \sigma \rangle :: a$*" be an execution of our operational semantics, from the first bytecode instruction of the method* `main` *i.e., first*(`main`) *and an initial state* ξ *containing no reachability among variables. Suppose that* $\sigma \in \Sigma_\tau$ *and let* $I_{\mathsf{ins}} \in A_\tau$ *be the approximation on the actual unreachability and non-cyclicity w.r.t. a set of fields at the* `ACG` *node corresponding to* ins*. Then,* $\sigma \in \gamma_\tau(I_{\mathsf{ins}})$.

Proof. We prove the Soundness by induction on the length n of the execution $\langle b_{first(\texttt{main})} \parallel \xi\rangle \Rightarrow^* \langle \boxed{\begin{array}{c} ins \\ rest \end{array}} \begin{array}{l}\rightarrow b_1 \\ \cdots \\ \rightarrow b_m\end{array} \parallel \sigma\rangle :: a$.

`Base Case:` $n = 0$; the execution is just $\langle b_{first(\texttt{main})} \parallel \xi\rangle$. In this case, $\sigma = \xi \in \Sigma_\tau$ and $I_{\mathsf{ins}} = \left\{ a \not\rightsquigarrow^{\mathcal{F}} b, c \rightsquigarrow^{\not\circlearrowright\mathcal{F}} | a, b, c \in \mathsf{dom}(\tau) \right\}$. Since, by definition, ξ contains no reachability between variables, for each pair of variables $\langle a, b\rangle \in \mathsf{dom}(\tau)$ we can assert, by Definition 4.7, $a \not\rightsquigarrow^{\mathcal{F}}_{\xi} b$ and, respectively, for each variable $c \in \mathsf{dom}(\tau)$ we can assert, by Definition 4.8, $c \rightsquigarrow^{\not\circlearrowright\mathcal{F}}_{\xi}$. Therefore $\xi \in \gamma_\tau(I_{\mathsf{ins}})$.

`Inductive Step:` we assume that thesis holds for any such execution of length $k \leq n$. Consider an execution $\langle b_{first(\texttt{main})} \parallel \xi\rangle \Rightarrow^{n+1} \langle \underbrace{\boxed{\begin{array}{c} ins_q \\ rest_q \end{array}}}_{b_q} \begin{array}{l}\rightarrow b_1 \\ \cdots \\ \rightarrow b_m\end{array} \parallel \sigma_q\rangle :: a_q$, with $ins_q(\sigma_q)$ defined. This execution must have the form

$$\langle b_{first(\texttt{main})} \parallel \xi\rangle \Rightarrow^{n_p} \langle \overbrace{\boxed{\begin{array}{c} ins_p \\ rest_p \end{array}}}^{b_p} \begin{array}{l}\rightarrow b'_1 \\ \cdots \\ \rightarrow b'_{m'}\end{array} \parallel \sigma_p\rangle :: a_p \Rightarrow^{n+1-n_p} \langle b_q \parallel \sigma_q\rangle :: a_q \tag{7.15}$$

with $0 \leq n_p \leq n$, that is, it must have a strict prefix of length n_p whose final activation stack has the topmost configuration with a non-empty block b_p. Let such n_p be maximal. Given a bytecode ins_a, let τ_a and I_a be the static type information and the approximation of the unreachability and non-cyclicity information w.r.t. a set of fields at the `ACG` node $\boxed{ins_a}$ respectively.

By inductive hypothesis we know that $\sigma_p \in \gamma_{\tau_p}(I_p)$ and we are going to prove that also $\sigma_q \in \gamma_{\tau_q}(I_q)$ holds. We distinguish on the basis of the rule of the operational semantics that is applied at the beginning of the derivation $\Rightarrow^{n+1-n_p}$ in Relation 7.15.

Rule (1). We have that $ins_p(\sigma_p)$ is defined and ins_p is not a call. We distinguish the following cases.

a) $\mathsf{ins_p} \notin \{\mathsf{return\ t}, \mathsf{throw}\ \kappa\}$. If $rest_p$ is not empty, then, by the maximality of n_p, the Relation 7.15 must be

$$\langle b_{first(\texttt{main})} \parallel \xi\rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\begin{array}{c} ins_p \\ ins_q \\ rest_q \end{array}}}_{b_p} \begin{array}{l}\rightarrow b_1 \\ \cdots \\ \rightarrow b_m\end{array} \parallel \sigma_p\rangle :: a_p$$

$$\overset{\text{Rule (1)}}{\Rightarrow} \langle \underbrace{\boxed{\begin{array}{c} ins_q \\ rest_q \end{array}}}_{b_q} \begin{array}{l}\rightarrow b_1 \\ \cdots \\ \rightarrow b_m\end{array} \parallel \underbrace{ins_p(\sigma_p)}_{\sigma_q}\rangle :: a_q$$

Otherwise $m' \geq 1$ (legal Java bytecode can only end with a $\mathsf{return\ t}$ or a $\mathsf{throw}\ \kappa$) and, by the maximality of n_p, it must be $b_q = b'_h$ for a suitable $1 \leq h \leq m'$, so that Relation 7.15

must have the form

$$\langle b_{first(\texttt{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{ins_p}\rightarrow}_{b_p} \begin{matrix} b_1 \\ \cdots \\ b_m \end{matrix} \parallel \sigma_p \rangle :: a_p$$

$$\overset{\text{Rule (1)}}{\Rightarrow} \langle \underbrace{\boxed{}\rightarrow}_{b_q} \begin{matrix} b_1 \\ \cdots \\ b_m \end{matrix} \parallel \underbrace{ins_p(\sigma_p)}_{\sigma_q} \rangle :: a_q \overset{\text{Rule (6)}}{\Rightarrow} \langle b_q \parallel \sigma_q \rangle :: a_q$$

In both cases, the ACG contains a sequential arc form $\boxed{ins_p}$ to $\boxed{ins_q}$, and $I_q = \Phi(I_p)$, where Φ is the propagation rule of the arc in Definition 6.3. Hence we have

$$\begin{aligned}
\sigma_q &= ins_p(\sigma_p) \\
&= ins_p(\sigma_p) \cap \Xi_{\tau_q} && [\sigma_q \in \Xi_{\tau_q}] \\
&\subseteq ins_p(\gamma_{\tau_q}(I_p)) \cap \Xi_{\tau_q} && [\text{by In. Hyp. and monotonicity of } ins_p] \\
&\subseteq \gamma_{\tau_q}(\Phi(I_p)) = \gamma_{\tau_q}(I_q) && [\text{by Lemma 7.3 and 7.8}]
\end{aligned}$$

b) $\mathsf{ins_p}$ = return t. If t = void there is no return value to consider. Otherwise, if t ≠ void, $rest_p$ is empty and $m' = 0$, since no code is executed after a return t in legal Java bytecode, but the method terminates. Furthermore, by Definition of the return t semantics, we have $ins_p(\sigma_p) \in \Xi$, and hence the Relation 7.15, depending on the emptiness of block b in Rule 4, must have one of this two forms:

$$\langle b_{first(\texttt{main})} \parallel \xi \rangle$$

$$\Rightarrow^{n_p} \langle \underbrace{\boxed{\text{return t}}}_{b_p} \parallel \underbrace{\langle\langle l_p \parallel top :: s_p \rangle, \mu_p \rangle\rangle}_{\sigma_p} :: \overbrace{\underbrace{\langle b_q \parallel \langle\langle l_c \parallel s_c \rangle, \mu_c \rangle\rangle}_{a_p} :: a_q}^{\text{call-time}}$$

$$\overset{\text{Rule (1)}}{\Rightarrow} \langle \boxed{} \parallel \langle\langle l_p \parallel top \rangle, \mu_p \rangle\rangle :: a_p \overset{\text{Rule (4)}}{\Rightarrow} \langle b_q \parallel \overbrace{\langle\langle l_c \parallel top :: s_c \rangle, \mu_p \rangle\rangle}^{\sigma_q} \rangle :: a_q$$

or

$$\langle b_{first(\texttt{main})} \parallel \xi \rangle$$

$$\Rightarrow^{n_p} \langle \underbrace{\boxed{\text{return t}}}_{b_p} \parallel \underbrace{\langle\langle l_p \parallel top :: s_p \rangle, \mu_p \rangle\rangle}_{\sigma_p} :: \overbrace{\underbrace{\langle \boxed{}\rightarrow \begin{matrix} b'_1 \\ \cdots \\ b'_{m'} \end{matrix} \parallel \langle\langle l_c \parallel s_c \rangle, \mu_c \rangle\rangle :: a_q}_{a_p}}^{\text{call-time}}$$

$$\overset{\text{Rule (1)}}{\Rightarrow} \langle \boxed{} \parallel \langle\langle l_p \parallel top \rangle, \mu_p \rangle\rangle :: a_p \overset{\text{Rule (4)}}{\Rightarrow} \langle \boxed{}\rightarrow \begin{matrix} b'_1 \\ \cdots \\ b'_{m'} \end{matrix} \parallel \langle\langle l_c \parallel top :: s_c \rangle, \mu_p \rangle\rangle :: a_q$$

$$\overset{\text{Rule (6)}}{\Rightarrow} \langle b_q \parallel \langle\langle l_c \parallel top :: s_c \rangle, \mu_p \rangle\rangle :: a_q$$

where, in the latter case, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \leq h \leq m'$. We only prove the case for the former, the latter being similar. Consider the configuration at call-time. Since only Rule 2 can stack configurations, it was on top when a call was

executed and, from a suitable $1 \leq w \leq n$, it has to be the form

$$\langle b_{first(\texttt{main})} \parallel \xi \rangle$$
$$\Rightarrow^{n_c} \langle \boxed{\begin{array}{c}\textsf{call } k_1.m \ldots k_n.m \\ ins_q \\ rest_q\end{array}} \begin{array}{l}\rightarrow b'_1 \\ \cdots \\ \rightarrow b'_{m'}\end{array} \parallel \underbrace{\langle\langle l_c \parallel [v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0]\rangle, \mu_c\rangle\rangle}_{\sigma_c} :: a_q$$
$$\overset{\text{Rule (2)}}{\Rightarrow} \langle first(\texttt{main}) \parallel \langle\langle [v_{j-\pi} :: \ldots :: v_{j-1}] \parallel \epsilon\rangle, \mu_c\rangle\rangle :: a_p$$
$$\Rightarrow^{n_p - n_c - 1} \langle b_p \parallel \sigma_p\rangle :: a_p \overset{\text{Rule (1)}}{\Rightarrow} \langle \boxed{} \parallel \langle\langle l_p \parallel top\rangle, \mu_p\rangle\rangle :: a_p$$
$$\overset{\text{Rule (5)}}{\Rightarrow} \langle b_q \parallel \sigma_q\rangle :: a_q$$

where j is the number of stack elements before $C = \textsf{call} k_1.m \ldots k_n.m$ is executed, π is the number of the parameters of method m and the rules in the portion $\Rightarrow^{n_p - n_c - 1}$ never make the stack lower than at the beginning of that portion.

We consider $\sigma_c = \langle\langle l_c \parallel [v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0]\rangle, \mu_c\rangle$ and $\sigma_p = \langle\langle l_p \parallel top\rangle, \mu_p\rangle$. By inductive hypothesis for n_c and n_p we know that $\sigma_c \in \gamma_{\tau_c}(I_c)$ and $\sigma_p \in \gamma_{\tau_p}(I_p)$. It is worth noting that, in this case, $\sigma_q = d((makescope\ m_w)(\sigma_c)) \cap \Xi_{\tau_q}$ and, since $\sigma_c \in \gamma_{\tau_c}(I_c)$, we obtain

$$\sigma_q \subseteq d((makescope\ m_w)(\gamma_{\tau_c}(I_c))) \cap \Xi_{\tau_q} \tag{7.16}$$

Let $\sigma_e = \textsf{return t}\ (\sigma_p) = \langle\langle l_p \parallel top\rangle, \mu_q\rangle$, the ACG contains a final arc from $\boxed{\textsf{return t}}$ to $\boxed{\textsf{exit@}m_w}$, for a suitable $1 \leq w \leq n$, and $I_q = \Phi^{\#9}(I_p)$. The following relation holds

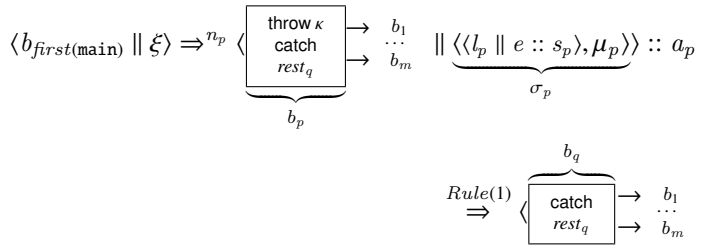
$$\begin{array}{lll}\sigma_e = \textsf{return t}\ (\sigma_p) & & \\ \quad \subseteq \textsf{return t}\ (\gamma_{\tau_p}(I_p)) & & \text{[by In. Hyp. and monotonicity of return t]} \\ \quad \subseteq \gamma_{\tau_q}(\Phi(I_p)) = \gamma_{\tau_q}(I_q) & & \text{[by Lemma 7.4]}\end{array}$$

In this case there is a return value multi-arcs going into $\boxed{ins_q}$ and I_c and I_e represents the correct approximation of our properties information at the sources of the arcs: $I_q = \Phi^{\#12}(I_c, I_e)$. By Relation 7.17 and Lemma 7.7, we have $\sigma_q \in \gamma_{\tau_q}(I_q)$.

c) $\textsf{ins}_\textsf{p} = \textsf{throw}\ \kappa$. If $rest_p$ is empty and $m' > 0$, the execution 7.15 must have the form:

$$\langle b_{first(\texttt{main})} \parallel \xi\rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\textsf{throw}\ \kappa}}_{b_p} \begin{array}{l}\rightarrow b'_1 \\ \cdots \\ \rightarrow b'_{m'}\end{array} \parallel \underbrace{\langle\langle l_p \parallel e :: s_p\rangle, \mu_p\rangle\rangle}_{\sigma_p} :: a_p$$
$$\overset{Rule(1)}{\Rightarrow} \langle \boxed{} \begin{array}{l}\rightarrow b'_1 \\ \cdots \\ \rightarrow b'_{m'}\end{array} \parallel \overbrace{\langle\langle l_p \parallel e\rangle, \mu_p\rangle\rangle}^{\sigma_q} :: a_p \overset{Rule(6)}{\Rightarrow} \langle b_q \parallel \sigma_q\rangle :: \overbrace{a_p}^{a_q},$$

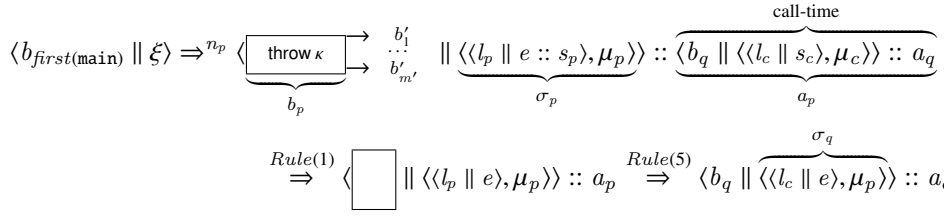
where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \leq h \leq m'$. Contrariwise if $rest_q$ is not empty and $m' > 0$, the execution 7.15 must have the form



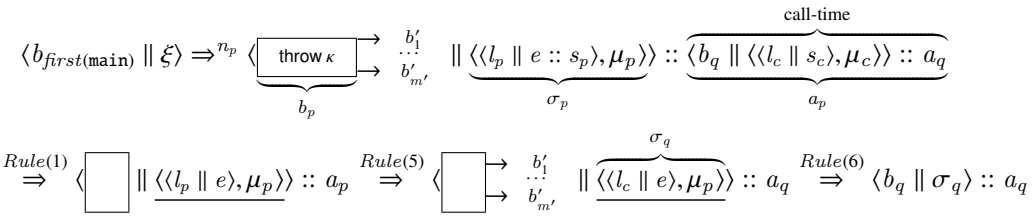
since catch is the only bytecode whose semantics can be defined on the exceptional state σ_q (see Figure 3.1). In both these cases, by inductive hypothesis we have $\sigma_p \in \gamma_{\tau_p}(I_p)$, the ACG contains an exceptional arc from $\boxed{\text{throw } \kappa}$ to $\boxed{\text{catch}}$, and $I_q = \Phi^{\#13}(I_p)$. In this case, $\sigma_q \in \Xi_{\tau_q}$ and we have

$$
\begin{aligned}
\sigma_q &= \mathsf{throw}\ \kappa\ (\sigma_p) \\
&= \mathsf{throw}\ \kappa\ (\sigma_p) \cap \Xi_{\tau_q} && [\text{since } \sigma_q \in \Xi_{\tau_q}] \\
&\subseteq \mathsf{throw}\ \kappa\ (\gamma_{\tau_p}(I_p)) \cap \Xi_{\tau_q} && [\text{by In. Hyp. and monotonicity of } \mathsf{throw}\ \kappa] \\
&\subseteq \gamma_{\tau_q}(\Phi(I_p)) = \gamma_{\tau_q}(I_q). && [\text{by Lemma 7.8}]
\end{aligned}
$$

If $rest_p$ is empty, the execution 7.15 must have the form, depending on the emptiness of block b in Rule (5):



or



where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \leq h \leq m'$. We are going to prove only the former, the latter being similar.

Consider the configuration at call-time. Since only Rule (2) can stack configurations, it was on top whe a call was executed and 7.15 must have the form

$$\langle b_{first(\texttt{main})} \parallel \xi \rangle$$
$$\Rightarrow^{n_c} \langle \boxed{\begin{array}{c}\textsf{call } k_1.m \ldots k_n.m \\ ins_q \\ rest_q\end{array}} \begin{array}{l}\rightarrow b'_1 \\ \cdots \\ \rightarrow b'_{m'}\end{array} \parallel \overbrace{\langle\langle l_c \parallel [v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0]\rangle, \mu_c\rangle\rangle}^{\sigma_c} :: a_q$$
$$\overset{\text{Rule (2)}}{\Rightarrow} \langle first(\texttt{main}) \parallel \langle\langle [v_{j-\pi} :: \ldots :: v_{j-1}] \parallel \epsilon\rangle, \mu_c\rangle\rangle :: \langle b_q \parallel \langle\langle l_q \parallel s_q\rangle, \mu_q\rangle\rangle :: a_q$$
$$\Rightarrow^{n_p - n_c - 1} \langle b_p \parallel \sigma_p\rangle :: a_p \overset{\text{Rule (1)}}{\Rightarrow} \langle \boxed{} \parallel \underline{\langle\langle l_p \parallel top\rangle, \mu_p\rangle\rangle} :: a_p \overset{\text{Rule (5)}}{\Rightarrow} \langle b_q \parallel \sigma_q\rangle :: a_q$$

where j is the number of stack elements before $C = \textsf{call } k_1.m \ldots k_n.m$ is executed, π is the number of the parameters of method m and the rules in the portion $\Rightarrow^{n_p - n_c - 1}$ never make the stack lower than at the beginning of that portion. Since $\sigma_q \in \underline{\Xi}_{\tau_q}$, the only possibility for ins_q is to be a catch.

We consider $\sigma_c = \langle\langle l_c \parallel [v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0]\rangle, \mu_c\rangle$ and $\sigma_p = \langle\langle l_p \parallel top\rangle, \mu_p\rangle$. By inductive hypothesis for n_c and n_p we know that $\sigma_c \in \gamma_{\tau_c}(I_c)$ and $\sigma_p \in \gamma_{\tau_p}(I_p)$. It is worth noting that, in this case, $\sigma_q = d((makescope\ m_w)(\sigma_c)) \cap \underline{\Xi}_{\tau_q}$ and, since $\sigma_c \in \gamma_{\tau_c}(I_c)$, we obtain

$$\sigma_q \subseteq d((makescope\ m_w)(\gamma_{\tau_c}(I_c))) \cap \underline{\Xi}_{\tau_q} \tag{7.17}$$

Let $\sigma_e = \textsf{throw } \kappa\ (\sigma_p) = \underline{\langle\langle l_p \parallel top\rangle, \mu_q\rangle}$, the ACG contains an exceptional arc from $\boxed{\textsf{throw } \kappa}$ to $\boxed{\textsf{exception@}m_w}$, for a suitable $1 \leq w \leq n$, and $I_q = \Phi^{\#9}(I_p)$. The following relation holds

$$\begin{array}{lr}
\sigma_e = \textsf{throw } \kappa\ (\sigma_p) & \\
\quad = \textsf{throw } \kappa\ (\sigma_p) \cap \underline{\Xi}_{\tau_q} & [\text{since } \sigma_q \in \underline{\Xi}_{\tau_q}] \\
\quad \subseteq \textsf{throw } \kappa\ (\gamma_{\tau_p}(I_p)) \cap \underline{\Xi}_{\tau_q} & [\text{by In. Hyp. and monotonicity of } \textsf{throw } \kappa] \\
\quad \subseteq \gamma_{\tau_q}(\Phi(I_p)) = \gamma_{\tau_q}(I_q). & [\text{by Lemma 7.4}]
\end{array}$$

In this case there is a side-effect multi-arcs going into $\boxed{\textsf{catch}}$ and I_c and I_e represents the correct approximation of our properties information at the sources of the arcs: $I_q = \Phi^{\#15}(I_c, I_e)$. By Relation 7.17 and Lemma 7.9, we have $\sigma_q \in \gamma_{\tau_q}(I_q)$.

Rule (2). By Definition of $makescope$, the Relation 7.15 must have the form

$$\langle b_{first(\texttt{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\textsf{call } k_1.m \ldots k_n.m} \begin{array}{l}\rightarrow b'_1 \\ \cdots \\ \rightarrow b'_{m'}\end{array}}_{b_p} \parallel \underbrace{\langle\langle l_c \parallel [v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0]\rangle, \mu_p\rangle\rangle}_{\sigma_p} :: a_p$$
$$\overset{\text{Rule (2)}}{\Rightarrow} \overbrace{\langle first(\mathtt{k}_w.m)}^{b_q} \parallel \overbrace{\langle\langle [v_{j-\pi} :: \ldots :: v_{j-1}] \parallel \epsilon\rangle, \mu_c\rangle\rangle}^{\sigma_q} :: a_q$$

where j is the number of stack elements before $C = \textsf{call } k_1.m \ldots k_n.m$ is executed, π is the number of the parameters of method m. In this case, the ACG contains the parameter passing

arc from $\boxed{\mathsf{call} k_1.m \ldots k_n.m}$ to $\boxed{first(\mathtt{k}_w.m)}$, for a suitable $w \in [1 \ldots n]$ and $I_q = \Phi^{\#10}(I_p)$. We have

$$\begin{aligned}
\sigma_q &= makescope(\sigma_p) \\
&\subseteq makescope(\gamma_{\tau_p}(I_p)) && \text{[by In. Hyp. and monotonicity of } makescope\text{]} \\
&\subseteq \gamma_{\tau_q}(\Phi(I_p)) = \gamma_{\tau_q}(I_q). && \text{[by Lemma 7.5]}
\end{aligned}$$

Rule (3). Let i and j be the number of local variables and stack elements before $C = \mathsf{call}\ k_1.m \ldots k_n.m$ is executed and π be the number of parameters of method m. In this case, the Relation 7.15 must have the form

$$\langle b_{first(\mathtt{main})} \parallel \xi \rangle$$

$$\Rightarrow^{n_p} \langle \underbrace{\boxed{\begin{array}{c}\mathsf{call}\ k_1.m \ldots k_n.m \\ rest_p\end{array}} \rightarrow \begin{array}{c} b'_1 \\ \cdots \\ b'_{m'} \end{array}}_{b_p} \parallel \underbrace{\langle\langle l_c \parallel [v_{j-1} :: \ldots :: v_{j-\pi+1} :: \mathtt{null} :: \ldots :: v_0]\rangle, \mu_p\rangle}_{\sigma_p}\rangle :: a_p$$

$$\overset{\text{Rule (3)}}{\Rightarrow} \langle \overbrace{\boxed{rest_p} \rightarrow \begin{array}{c} b'_1 \\ \cdots \\ b'_{m'} \end{array}}^{b_q} \parallel \overbrace{\underline{\langle\langle l_p \parallel \ell\rangle, \mu_p[\ell \mapsto npe\rangle\rangle}}^{\sigma_q} :: a_q$$

where $rest_p$ is non-empty, while otherwise it has the form

$$\langle b_{first(\mathtt{main})} \parallel \xi \rangle$$

$$\Rightarrow^{n_p} \langle \underbrace{\boxed{\mathsf{call}\ k_1.m \ldots k_n.m} \rightarrow \begin{array}{c} b'_1 \\ \cdots \\ b'_{m'} \end{array}}_{b_p} \parallel \underbrace{\langle\langle l_c \parallel [v_{j-1} :: \ldots :: v_{j-\pi+1} :: \mathtt{null} :: \ldots :: v_0]\rangle, \mu_p\rangle}_{\sigma_p}\rangle :: a_p$$

$$\overset{\text{Rule (3)}}{\Rightarrow} \langle \overbrace{\boxed{} \rightarrow \begin{array}{c} b'_1 \\ \cdots \\ b'_{m'} \end{array}}^{b_q} \parallel \overbrace{\underline{\langle\langle l_p \parallel \ell\rangle, \mu_p[\ell \mapsto npe\rangle\rangle}}^{\sigma_q} :: a_q \overset{\text{Rule (6)}}{\Rightarrow} \langle b_q \parallel \sigma_q\rangle :: a_q$$

where, by maximality of n_p, we have $b_p = b'_h$ for a suitable $1 \leq h \leq m'$. In both cases, the ACG contains an exceptional multi-arc from $\boxed{\mathsf{call}\ \kappa_1.m \ldots \kappa_n.m}$ and $\boxed{\mathsf{exception@}\, m_w}$ to $\boxed{\mathsf{catch}}$, with I_p and I_e the correct approximation of our properties information at the sources: $I_q = \Phi^{\#15}(I_p, I_e)$. We have $\sigma_q = \langle\langle l_p \parallel \ell\rangle, \mu_p[\ell \mapsto npe]\rangle = \langle\langle l_p \parallel \ell\rangle, \mu_e\rangle$ and therefore, since it is the same state used to prove Lemma 7.9, we can exploit its proof to assert $\sigma_q \in \gamma_{\tau_q}(I_q)$.

Rule (4), (5), (6). These rules can not be applied at the beginning of the derivation $\Rightarrow^{n+1-n_p}$, since they work only with an empty block and b_p is not-empty by hypothesis. □

Chapter 8

Conclusions

To say what we have done in one sentence, we have introduced a new abstract domain for the field-sensitive unreachability and non-cyclcicity static analysis of program variables. Hence the main contribution of this thesis is the definition of 15 propagation rules i.e., 15 functions explaining how the information is propagated in the abstract constraint graph of the program under analysis, depending on the considered Java Bytecode.

Furthermore, we have proved the soundness of each propagation rule applied to an abstract element I of our domain. Namely, it has been shown that the concretization set of each propagation rule contains all the possible states resulting from the application of the correspondent instruction to the concretized states obtained by I.

Finally we have proved the soundness of the analysis with respect to the small-step semantics previously provided by induction on the length of the derivation.

We note that this analysis confirms yet again the theoretical result obtained by Logozzo and Fähndrich [11] i.e., in order to achieve a precision-enough analysis of a low-level code we need to use ever more finer domains: in that sense our domain is a refinement of the reachability and non-cyclicity ones introduced respectively in [13, 14] and [18]. However a finer domain entails a more complex abstract transfer function with more cases to handle (and hence to prove), mainly due to the presence of object manipulation instructions, such as $\mathsf{getfield}\ \kappa.f:\mathsf{t}$ and $\mathsf{putfield}\ \kappa.f:\mathsf{t}$, and side effects derived from a method call.

Another notable topic arisen in this paper, always related with the precision/complexity trade-off, is the massive use of pre-processed information deriving from other static analyses, often those that we are trying to refine. It is worth noting that these results can always be exploited as definite information of the program behavior: also considering possible analyses, we can affirm definite statements of the corresponding denied properties. For instance we can definitely assert the unreachability between two variables when this pair is not inside the possible reachability abstract set.

Since our unreachability definition is based on the *material implication*, we have also exploited the positive results of the possible reachability: even if its approximation provides incorrect reachability relations, our unreachability property remains sound with respect to whatever set of fields because the antecedent is false. For example this approach has been followed to accomplish a more precise propagation rule relative to the $\mathsf{getfield}\ \kappa.f:\mathsf{t}$ instruction, in particular to build the set with respect to which the variables are unreachable from the new top of the stack.

Future work deals with the implementation of the analysis in the `Julia` tool [23, 24] by adding the abstract domain and solving the abstract constraint graph through its fix-point en-

gine. In this way, it will be possible to evaluate the real cost of our analysis and to verify whether it actually improves the precision of the termination checker by testing it both on samples of the termination competition[1] and on real Java/Android programs.

[1] http://termination-portal.org/wiki/Termination_Competition

Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley series in computer science. Addison-Wesley Pub. Co., 1986 (cit. on p. 15).

[2] A.R. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 2010 (cit. on p. 1).

[3] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000 (cit. on p. 1).

[4] P. Cousot and R. Cousot. "Systematic design of program analysis frameworks". In: *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Antonio, Texas: ACM Press, New York, NY, 1979, pp. 269–282 (cit. on pp. 4, 12).

[5] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '77. Los Angeles, California: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973 (cit. on pp. 1, 29).

[6] Samir Genaim and Damiano Zanardini. "Reachability-based Acyclicity Analysis by Abstract Interpretation". In: *CoRR* abs/1206.2188 (2012) (cit. on pp. 3, 4).

[7] Roberto Giacobazzi and Francesco Ranzato. "Refining and Compressing Abstract Domains". In: *ICALP*. 1997, pp. 771–781 (cit. on p. 4).

[8] M. Hind. "Pointer Analysis: Haven't We Solved This Problem Yet?" In: *Proceedings of PASTE'01*. Snowbird, Utah, United States: ACM, 2001, pp. 54–61 (cit. on p. 3).

[9] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3nd edition*. Pearson/Addison Wesley, 2007 (cit. on pp. 1, 29).

[10] T. Lindholm and F. Yellin. *The Java virtual machine specification*. Java series. Addison-Wesley, 1999 (cit. on pp. 4, 15, 17–19).

[11] Francesco Logozzo and Manuel Fähndrich. "On the relative completeness of bytecode analysis versus source code analysis". In: *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*. CC'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 197–212 (cit. on pp. 3, 5, 13, 79).

[12] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2010 (cit. on pp. 1, 7, 29, 31, 35).

[13] Đurica Nikolić and Fausto Spoto. *Reachability Analysis of Program Variables*. `http://profs.sci.univr.it/~nikolic/download/IJCAR2012/IJCAR2012Ext.pdf` (cit. on pp. 3, 15, 25, 34, 53, 79).

[14] Đurica Nikolić and Fausto Spoto. "Reachability Analysis of Program Varibles". In: *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR'12)*. Ed. by Bernhard Gramlich, Dale Miller, and Ulrike Sattler. Vol. 7364. Lecture Notes in Artificial Intelligence. Springer-Verlag Berlin Heidelberg, 2012, pp. 423–438 (cit. on pp. 3, 15, 34, 79).

[15] Durica Nikolic and Fausto Spoto. "Definite Expression Aliasing Analysis for Java Bytecode". In: *ICTAC*. 2012, pp. 74–89 (cit. on pp. 3, 4, 34).

[16] J. Palsberg and M. I. Schwartzbach. "Object-oriented Type Inference". In: *Proc. of OOPSLA'91*. Vol. 26(11). ACM SIGPLAN Notices. ACM Press, 1991, pp. 146–161 (cit. on p. 21).

[17] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. "Efficient field-sensitive pointer analysis for C". In: *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. PASTE '04. Washington DC, USA: ACM, 2004, pp. 37–42. DOI: `10.1145/996821.996835` (cit. on p. 3).

[18] Stefano Rossignoli and Fausto Spoto. "Detecting non-cyclicity by abstract compilation into boolean functions". In: *Proceedings of the 7th international conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI'06. Charleston, SC: Springer-Verlag, 2006, pp. 95–110. DOI: `10.1007/11609773_7` (cit. on pp. 3, 79).

[19] M. Sagiv, T. Reps, and R. Wilhelm. "Parametric Shape Analysis via 3-Valued Logic". In: *ACM Trans. Program. Lang. Syst.* 24 (3 2002), pp. 217–298 (cit. on p. 2).

[20] David A. Schmidt. "Underapproximating predicate transformers". In: *Proceedings of the 13th international conference on Static Analysis*. SAS'06. Seoul, Korea: Springer-Verlag, 2006, pp. 127–143 (cit. on p. 4).

[21] Stefano Secci and Fausto Spoto. "Pair-Sharing Analysis of Object-Oriented Programs". In: *Proceedings of the 12th SAS*. Vol. 3672. LNCS. London, UK: Springer, 2005, pp. 320–335 (cit. on pp. 3, 35).

[22] F. Spoto, F. Mesnard, and É. Payet. "A Termination Analyzer for Java Bytecode Based on Path-Length". In: *ACM Trans. on Programming Languages and Systems* 32.3 (2010), pp. 1–70 (cit. on p. 2).

[23] Fausto Spoto. "JULIA: A Generic Static Analyser for the Java Bytecode". In: *Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs - FTfJP'2005*. 2005 (cit. on pp. 34, 79).

[24] Fausto Spoto. *Julia in a Nutshell*. Tech. rep. (cit. on pp. 34, 35, 79).

[25] Fausto Spoto and Michael D. Ernst. "Inference of field initialization". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 231–240. DOI: `10.1145/1985793.1985826` (cit. on p. 13).

[26] Alfred Tarski. "A Lattice-Theoretical Fixpoint Theorem and its Applications". In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 285–309 (cit. on p. 11).