Đurica Nikolić

A General Framework for Constraint-Based Static Analyses of Java Bytecode Programs

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
prof. Nicola Fausto Spoto

Series N°: **TD-06-13**

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

Посвећено мојим родитељима Петру и Емилији
који су омогућили остварење и овог сна.

Dedicated to my parents Petar and Emilija
who have made this dream come true.

Summary

The present thesis introduces a *generic parameterized framework for static analysis of Java bytecode programs*, based on constraint generation and solving. This framework is able to deal with the exceptional flows inside the program and the side-effects induced by calls to non-pure methods. It is generic in the sense that different instantiations of its parameters give rise to different static analyses which might capture complex memory-related properties at each program point. Different properties of interest are represented as abstract domains, and therefore the static analyses defined inside the framework are abstract interpretation-based. The framework can be used to generate *possible* or *may* approximations of the property of interest, as well as *definite* or *must* approximations of that property. In the former case, the result of the static analysis is an over-approximation of what might be true at a given program point; in the latter, it is an under-approximation. This thesis provides a set of conditions that different instantiations of framework's parameters must satisfy in order to have a sound static analysis. When these conditions are satisfied by a parameter's instantiation, the framework guarantees that the corresponding static analysis is sound. It means that the designer of a novel static analysis should only show that the parameters he or she instantiated actually satisfy the conditions provided by the framework. This way the framework simplifies the proofs of soundness of the static analysis: instead of showing that the overall analysis is sound, it is enough to show that the provided instantiation describing the actual static analyses satisfies the conditions mentioned above. This a very important feature of the present approach.

Then the thesis introduces two novel static analyses dealing with memory-related properties: the *Possible Reachability Analysis Between Program Variables* and the *Definite Expression Aliasing Analysis*. The former analysis is an example of a possible analysis which determines, for each program point p, which are the ordered pairs of variables $\langle v, w \rangle$ available at p, such that v might reach w at p, i.e., such that starting from v it is possible to follow a path of memory locations that leads to the object bound to w. The latter analysis is an example of a definite analysis, and it determines, for each program point p and each variable v available at that point, a set of expressions which are always aliased to v at p. Both analyses have been formalized and proved sound by using the theoretical results of the framework. These analyses have been also implemented inside the Julia tool (`www.juliasoft.com`), which is a static analyzer for Java and Android. Experimental evaluation of these analyses on real-life benchmarks shows how the precision of Julia's principal checkers (nullness and termination checkers) increased compared to

the previous version of Julia where these two analyses were not implemented. Moreover, this experimental evaluation showed that the presence of the reachability analysis actually decreased the total run-time of Julia. On the other hand, the aliasing analysis takes more time, but the number of possible warnings produced by the principal checkers drastically decreased.

Acknowledgements

It would not have been possible to write this doctoral thesis without the immense help and support of the most important persons in my life - my parents Petar and Emilija and my brother Milan. I cannot thank you enough for all the support, advices, patience and love that you have given me.

I would like to express my sincere gratitude to my supervisor and friend, Prof. Nicola Fausto Spoto, for his advices, guidance, and encouragement throughout the course of this work. It has been a pleasure working with him.

Another person I would like to thank is Prof. Roberto Giacobazzi who has taught me a lot of things and who has given me a lot of valuable advices during last seven years. Professors Nicola Fausto Spoto, Roberto Giacobazzi and Graziano Pravadelli have followed my work during last three years and has given me a lot of advices on how to present my ideas better and how to improve my talks. I really appreciated that.

I would like to acknowledge the financial, academic and technical support of the University of Verona.

Last but not least, I would like to thank my “sisters” Becky, Nata, Vaća and Vida who are always there for me, as well as Wayne, Leo, Maja, Silvia, Sara, Valerio and Emad for their support and friendship.

Contents

List of Figures

1

Introduction

1.1 Static Analysis, for Real

Static analysis of computer programs allows one to gather information about the run-time behavior of such programs, before they are run. Hence, it is possible to prove that programs do not perform illegal operations, such as a division by zero or a dereference of `null`, or do not lead to erroneous executions, such as infinite loops, or do not divulge information in incorrect ways, such as security authorizations or GPS position in mobile devices. This must be performed without executing those programs. Companies such as banks and insurance companies are interested in the application of static analysis to their software, that must not break or hang unexpectedly. Software houses and freelance developers are attracted by the idea of verifying their software before it gets into production, so that a large class of bugs and inefficiencies can be removed before hitting the market.

Static analysis of real software is extremely difficult. There are many reasons for this:

- the precision of the static analysis must be very high, which makes the latter very complex. Moreover, for the users of static analysis, the ideal situation is when the technique can be applied without any help from the programmer. That is, the code is analyzed as it is, without the addition of annotations that might help the analyzer;
- current programming languages have complex semantics. They deal with data structures or objects dynamically allocated in the heap, rather than just primitive, numerical values. This makes the analysis complex, since data updates and method calls might affect other data indirectly, by side-effect;
- current programming languages use exceptions to implement exceptional executions. This introduces execution paths that do not follow the normal execution order of statements and that must be taken into account for a sound static analysis;
- current programming languages come with a large standard library, that is heavily used by even the smallest program. As a consequence, the analysis of a very small program might require the analysis of thousands of lines of code and consequently become very expensive;
- for all these reasons, the development of new static analyses is complex and error-prone. Once a formal specification of the analysis is provided, its implementation might require months and the debugging and optimization of that implementation might take up to one year before being industrially strong, from our experience. This

makes the development of sound and precise static analyses impractical and economically uninteresting.

This thesis describes an approach to the development of new static analyses for memory-related properties of a complex programming language such as Java bytecode. The technique is based on the construction of a system of constraints from the program under analysis, which takes the form of a graph, and on the propagation of information tokens along the arcs of the graph. There is complete freedom in the kind of propagation that gets applied here, the only requirement is that it has to be monotonic.

This is a generic technique, in the sense that it can be applied to the development of many, very different static analyses, both in possible and definite form. The framework has been formalized, developed, debugged and optimized only once, which took around two years of programming work. Its implementation has been optimized heavily, by using bitsets to represent sets of tokens of information and by using a compact representation for the graphs. New static analyses are implemented by subclassing the code already implemented in its generic form. Hence, new analyses can be developed in a couple of weeks and inherit the highly debugged and efficient generic implementation. As a consequence, the development of new analyses becomes practical and makes economical sense.

This thesis provides the theory underlying the definition of constraint-based static analyses for Java bytecode and shows two concrete instantiations of the framework for a possible and a definite analysis, respectively. The implementation of the framework has been performed by Julia Srl (`http://www.juliasoft.com`), a spin-off company of the University of Verona, specialized in the development of static analyses for Java and Android. The implementation of the two concrete instantiations of analysis has been performed in collaboration with the same company, which currently owns the code.

The importance of this thesis is that it shows how very different static analyses for a complex programming languages such as Java bytecode can be defined inside a single framework and inherit, automatically, properties such as correctness and efficiency of implementation. The considered analyses are related to the heap memory and hence very complex since they must take into account the side-effects of method calls and of field updates. Again, the framework in this thesis gives a general solution for this kind of analyses, by providing a technique for dealing with side-effects. Similarly, the solution for exception handling can be applied, automatically, to all static analyses, present and future.

1.2 General Idea of This Thesis

This thesis introduces a *general parameterized framework for constraint-based static analyses of Java bytecode programs*. That framework is abstract interpretation-based and can be used to formalize static analyses that deal with both numerical and memory-related properties. Moreover, the framework's structure allows one to define static analyses which deal with both the side-effects of the methods, and their exceptional executions. Finally, it simplifies proofs of correctness of the static analyses formalized in the framework.

In the following, the contribution of the thesis is explained in more detail. First of all, a *Java bytecode-like language* used by this formalization, as well as its *operational semantics* are formally defined. The crucial notion there is the notion of *state*, representing a system configuration. Namely, a state assigns a concrete value to each variable available

at a program point p, and to each field of each object available in memory at p. For each program point, this is the most concrete information about the execution of that program concerned with that point. The set of all possible states that might be related to a given program point is called the *concrete domain*, and it is denoted by C.

Let P be a program under analysis, composed of a set of `.class` files, and let L be the set of libraries that P uses. Suppose that P's classes as well as libraries in L are archived in a `.jar` file, representing one of the inputs the present framework requires. Moreover, let $\mathcal{P}$ be a generic property of interest. The actions performed by the framework are listed below.

1. The framework extracts, from the `.jar` archive, an extended control flow graph (eCFG), which contains a node for each bytecode instruction available in P and L, some special nodes which deal with the side-effects of non-pure methods, as well as with exceptional and non-exceptional method ends, and different types of arcs which connect those nodes. There are some simple arcs connecting one source node with one sink node, but there are also some special arcs, composed of two source nodes and one sink node: their main purpose is handling of the method's side effects in both exceptional and non-exceptional executions of those methods and they represent one of the actual contributions of the present thesis. Fig. 1.1 graphically illustrates this step. More details about eCFG can be found in Section 4.2. It is worth noting that this step does not depend on any particular property of interest, and is always done automatically by the framework.
2. Suppose that A is a generic abstract domain (Section 2.2) representing the property $\mathcal{P}$. Moreover suppose that, for each arc, there exists a generic propagation rule $\Pi : \mathsf{A} \rightarrow \mathsf{A}$ representing the behavior of the bytecode instruction corresponding to the source node of the arc with respect to the abstract domain A, and therefore with respect to the property $\mathcal{P}$. Both the abstract domain and the propagation rules are property-dependent and represent the actual parameters of the framework. When this parameters are instantiated, the framework annotate the arcs of the eCFG, obtaining the *abstract constraints graph* (ACG). An example of an ACG is given in Fig. 1.2. More details about this step are given in Sections 4.3 and 4.4.
3. From the annotated graph the framework extracts a *system of constraints* which represents the actual definition of a new constraint-based static analysis and its solution represents the approximation provided by that static analysis. This step is both property-independent and property-dependent, i.e., the construction of the system of constraints does not depend on the property of interest, while its form does depend on that property and, in particular, on the propagation rules. The extraction of constraints from the ACG is explained in Section 4.5.

In order to define sound static analyses, the framework introduces a set of *requirements that framework's parameters, i.e., the abstract domain and the propagation rules, must satisfy*. For example, the framework requires that the concrete domain C and the abstract domain A are related in terms of abstract interpretation, i.e., it is necessary to show the correspondence between each abstract element and its concrete counterparts and vice versa. Another requirement is, for instance, the monotonicity of the propagation rules needed for the existence of a solution of the corresponding static analysis. The other, more complicated requirements and their purpose are explained later in Section 4.4. The results provided in Section 4.6 guarantee that when an instantiation of framework's

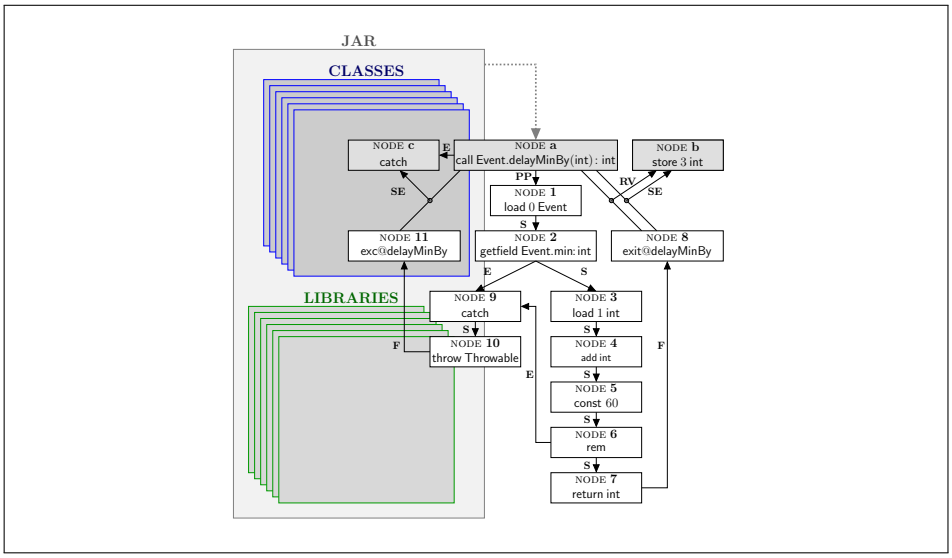


Fig. 1.1. Extraction of CFG from a program and the libraries it uses

parameters satisfies the requirements mentioned above, the static analysis determined by this instantiation is sound. This is a very important result, since it allows one to show that a static analysis of a huge, real-life program, written in Java bytecode, is sound. In order to show that, it is not necessary to consider program's structure or the libraries it uses. Another important result of this thesis is the fact that the satisfiability of the requirements

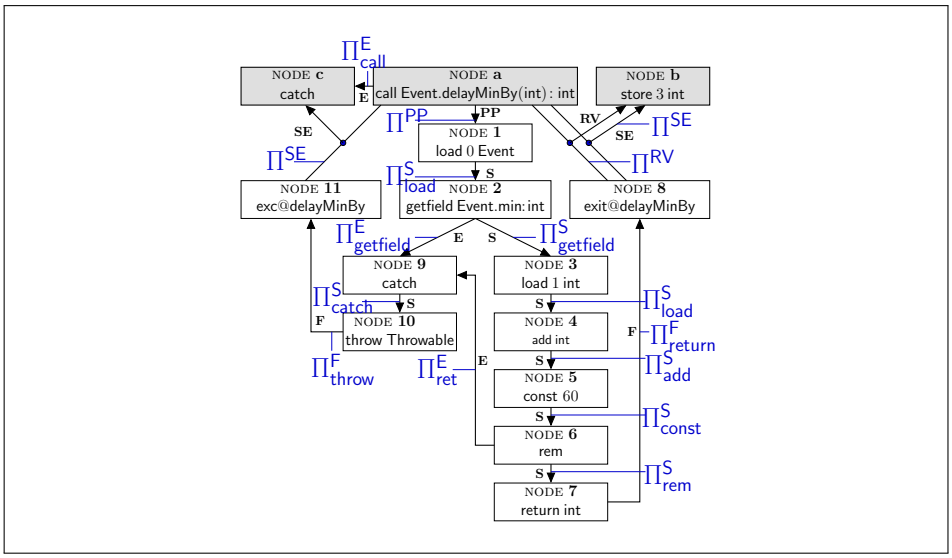


Fig. 1.2. An example of an ACG

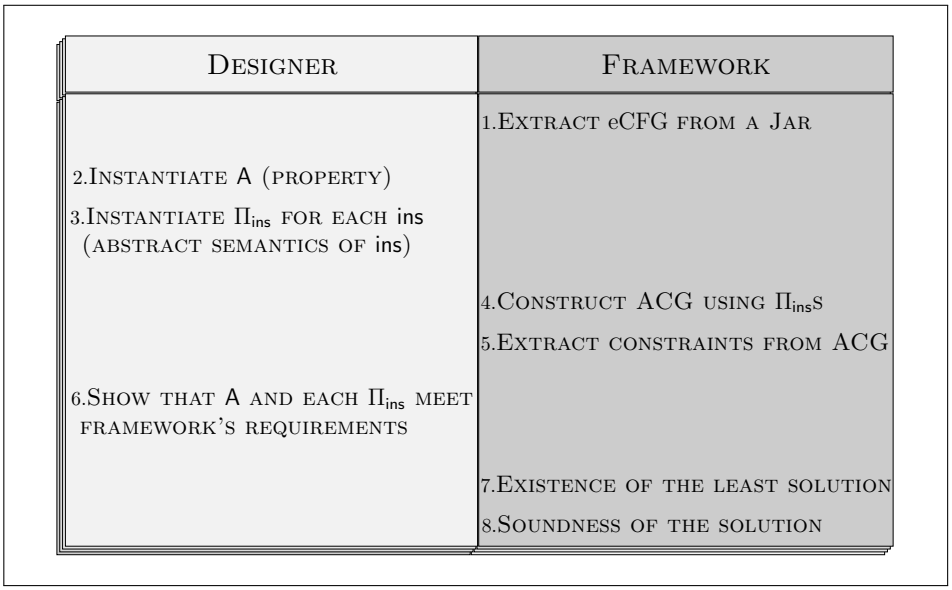


Fig. 1.3. Process of formalization of a constraint-based static analysis

implies that there always exists a solution of the system of constraints obtained from the ACG.

When the designer of a new static analysis wants to formalize a sound static analysis using the present framework, he or she has to complete the following tasks:

1. *formally define the property of interest* $\mathcal{P}$ and express it as an *abstract domain* A in terms of the abstract interpretation framework [32];
2. *define a propagation rule* $\Pi : \mathsf{A} \rightarrow \mathsf{A}$, for each bytecode instruction of the target language, and each possible behavior of that instruction (exceptional and non-exceptional);
3. show that A and the propagation rules satisfy all of the requirements specified by the framework, which is not a simple task, but it is drastically easier to show that, for example, each propagation rule soundly mimics the behavior of the corresponding bytecode instruction, than to show that the abstract semantics of whole program soundly approximates the operational semantics of that program.

Fig. 1.3 lists the sequence of actions automatically done by the framework (at the right) and required by the user (on the left). The actions are performed following the order specified in the figure, i.e., first of all the framework extracts the eCFG, then the designer instantiate the abstract domain and the propagation rules, then the framework constructs the ACG, etc.

There is also another important benefit of using this framework. From an implementational point of view, Julia contains an abstract class (in terms of Java) implementing the generic engine for eCFG and ACG creation, constraint generation and solving. This implementation represents an implementation of the framework and is not a contribution of the present thesis. Each new specific static analysis is a concrete subclass of the abstract class mentioned above, providing an implementation for a few methods, where the

specific static analyses deviates from the general framework. This largely simplifies the implementation of new static analyses. For instance, the developer need not bother about the implementation of the constraints and the strategy for their solution.

1.3 Does This Really Work?

The present framework has been used to *define*, *formally prove sound* and *implement* seven static analyses dealing with memory-related properties:

- *Possible Creation Point Analysis*,
- *Possible Field Nulnness Analysis*,
- *Possible Rawness of Variables and Fields*,
- *Possible Reachability Analysis of Program Variables*,
- *Possible Sharing Analysis*,
- *Definite Expression Aliasing Analysis*,
- *Definite Non-Null Expressions Analysis*.

All these analyses statically determine some memory-related run-time properties and have been implemented in Julia. They are fast analyses which allow one to analyse real-life benchmarks composed of hundreds of thousands of lines of code. This thesis pays particular attention to the *Possible Reachability Analysis of Program Variables* and the *Definite Expression Aliasing Analysis*.

Possible Reachability Analysis of Program Variables is an example of a *possible analysis*. Reachability from a program variable v to a program variable w states that starting from v it is possible to follow a path of memory locations that leads to the object bound to w. This useful piece of information is important for improving the precision of other static analyses, such as side-effects, field initialization, cyclicity and path-length analysis, as well as more complex analyses built upon them, such as nullness and termination analysis. It determines, for each program point p, a set of ordered pairs of variables of reference type $\langle v, w \rangle$ such that v *might reach* w at p when the program is executed on an arbitrary input. Seen the other way round, if a pair $\langle v, w \rangle$ is not present in our over-approximation at p, it means that v *definitely does not reach* w at p.

On the other hand, the *Definite Expression Aliasing Analysis* infers, for each variable v at each program point p, a set of expressions whose value at p is equal to the value of v at p, for every possible execution of the program. Namely, it determines which expressions *must* be aliased to local variables and stack elements of the Java Virtual Machine. The approximation produced by this analysis is another useful piece of information for an inter-procedural static analyzer, since it can refine other static analyses at conditional statements or assignments.

Both Possible Reachability Analysis of Program Variables and Definite Expression Aliasing Analysis have been implemented in the Julia static analyzer for Java and Android. A collection of real-life benchmarks has been analyzed by Julia in order to experimentally evaluate these novel static analyses and we show the results of those evaluations: both analyses improved the precision of Julia's principal checkers, which are *nullness* and *termination* checkers. In the case of reachability analysis, the run-time of Julia decreased with respect to the previous version of the tool, and the reason of this fact is explained.

On the other hand, the aliasing analysis increased the run-times of Julia, but it drastically decreased the number of warnings produced by the principal tools, which is a good trade-off.

1.4 Overview of the Thesis

This thesis is structured as follows. Chapter 2 provides notation and the basic algebraic notions that are used in the remainder of the thesis, together with a brief introduction to abstract interpretation. Chapter 3 formally introduces the operational semantics of a target, Java bytecode-like language, used also in [63, 64, 86] and inspired by the standard informal semantics [52].

Chapter 4 presents a *generic parameterized static analysis framework for Java bytecode*, based on constraint generation and solving. This framework is able to deal with the exceptional flows inside the program and the side-effects induced by calls to non-pure methods. It is generic in the sense that different instantiations of its parameters give rise to different static analyses which might capture complex properties of the heap at each program point. This framework can be used to generate both *possible* or *may* static analyses, as well as *definite* or *must* static analyses dealing with the property of interest. The formalization also characterizes the sufficient requirements that the instantiated parameters have to satisfy in order to have a sound static analysis. The satisfiability of those requirements entails the existence of a solution of that static analysis. Chapter 4 is based on [59].

Chapters 5 and 6 show two novel static analyses obtained as instantiations of the framework mentioned above. Chapter 5 introduces our Possible Reachability Analysis of Program Variables, which is an example of a possible analysis. That chapter is based on [61, 64]. On the other hand, Chapter 6 introduces our Definite Expression Aliasing Analysis, a definite static analysis which determines, for each program point, and each variable available at that point, a set of expressions that must be aliased to that variable, for any possible execution of the program. That chapter is based on [60, 63]. Both Chapters 5 and 6 show how these two novel static analyses can be obtained by instantiating the parameters of the framework introduced in Chapter 4 and that these instantiations satisfy the requirements specified by the framework, which entails the soundness of those static analyses. Moreover, we show the experimental evaluations of the implementations of both analyses inside the Julia static analyzer [4]. Chapter 7 discusses relater work, and finally, Chapter 8 concludes.

2

Background

In this chapter, we introduce the basic algebraic notions that we are going to use in the thesis. In Section 2.1 we describe the mathematical background, recalling the basic notions of sets, functions and relations, followed by an overview of fixpoint theory [33, 88]. We also give a brief presentation of lattice theory [36,40,41], recalling the basic algebraic ordered structures. Moreover, we formally define Galois connections and show some their properties which will be used in the rest of this thesis. In Section 2.2 we introduce abstract interpretation [32, 34], we characterize abstract domains in terms of Galois connections and we describe the property of soundness.

2.1 Basic notions

Sets

Sets are one of the most fundamental concepts in mathematics and represent collections of objects (or elements). We write $x \in C$ to denote that x is an element of the set C, i.e., that x belongs to C. The number of elements of C is called the *cardinality* of C and we denote it as $|C|$. Given two sets C and D, we say that:

- C is a subset of D, denoted as $C \subseteq D$, if every element of C belongs to D.
- C and D are equal, denoted as $C = D$, if C is a subset of D and viceversa, i.e., $C \subseteq D$ and $D \subseteq C$.
- C and D are different, denoted as $C \neq D$, if there exists at least one element belonging to one of the sets but not belonging to the the other one.

The *empty set* is the set without any element, and we denote it as $\varnothing$. Therefore, for every element x we have that $x \notin \varnothing$ and for every set C we have that $\varnothing \subseteq C$. The *union* of two sets C and D, denoted as $C \cup D$, represents the set of elements belonging to C or to D, and is defined as $C \cup D = \{x \mid x \in C \vee x \in D\}$. The *intersection* of tho sets C and D, denoted as $C \cap D$, represents the sets of elements belonging to both C and D, and is defined as $C \cap D = \{x \mid x \in C \wedge x \in D\}$. The *relative complement* of D in C (also called the *set-theoretic difference* of C and D), denoted as $C \setminus D$, represents the set of all elements which are members of C but not members of D, and is defined as $C \setminus D = \{x \mid x \in C \wedge x \notin D\}$. The *powerset* $\wp(C)$ of a set C is defined as the set of all possible subsets of C: $\wp(C) = \{D \mid D \subseteq C\}$.

Relations

A *relation* is used to describe certain properties of things and it specifies how certain things may be connected. Let x and y be two elements of a set C, we call a *non-ordered* pair the element (x, y) such that $(x, y) = (y, x)$, and an *ordered* pair the element $\langle x, y\rangle$, such that $\langle x, y\rangle \neq \langle y, x\rangle$, when $x \neq y$. This notion can be extended to the one of ordered n-tuple of n elements $x_1, \ldots, x_n$, with $n \geq 2$, by $\langle \ldots \langle\langle x_1, x_2\rangle, x_3\rangle \ldots\rangle$, denoted by $\langle x_1, \ldots, x_n\rangle$.

Definition 2.1. *Given n sets $\{C_i\}_{1\leq i\leq n}$. We define the* cartesian product *of the n sets C_i as the set of ordered n-tuples:*

$$C_1 \times \ldots \times C_n = \{\langle x_1, \ldots, x_n\rangle \mid \forall 1 \leq i \leq n . x_i \in C_i\}\,.$$

In particular, when $C_1 = \ldots = C_n = C$, we use C^n to denote the cartesian product $C \times \ldots \times C$.

Given two non-empty sets C and D, any subset of the cartesian product $C \times D$ defines a relation between the elements of C and the elements of D. Given a relation $\mathcal{R}$ between C and D, i.e., $\mathcal{R} \subseteq C \times D$, and two elements $x \in C$ and $y \in D$, then $\langle x, y\rangle \in \mathcal{R}$ and $x\mathcal{R}y$ are equivalent notations denoting that x is in relation $\mathcal{R}$ with y.

Definition 2.2. *A binary relation $\leq$ on a set C is a* partial order *on C if it is:*

- reflexive*, i.e., for each $x \in C$, $x \leq x$ holds;*
- antisymmetric*, i.e., for each pair $x, y \in C$ if $x \leq y$ and $y \leq x$, then $x = y$;*
- transitive*, i.e., given $x, y, z \in C$, if $x \leq y$ and $y \leq z$, then also $x \leq z$.*

Functions

Given two sets C and D, we define a *function* f from C to D as a relation between C and D such that for each $x \in C$ there exists exactly one $y \in D$ such that $\langle x, y\rangle \in f$, and in this case we write $f(x) = y$. We write $f : C \to D$ to denote a function f from C to D, where C is called the *domain* of f and is denoted by $\mathsf{dom}(f)$, while D is called the *co-domain* of f. The set $f(X) = \{f(x) \mid x \in X\}$ is the image of $X \subseteq C$ under f. In particular, the image of the domain, i.e., f(C), is called the *range* of f and is denoted by $\mathsf{rng}(f)$. The *composition* $g \circ f : C \to E$ of two functions $f : C \to D$ and $g : D \to E$, is defined as $g \circ f(x) = g(f(x))$. When it is clear from the context the symbol $\circ$ may be omitted and the composition can simply be denoted as gf. Sometimes, function f on variable x is denoted as $\lambda x.f(x)$.

Ordered Structures

We consider some basic ordered structures which, unlike sets, embody the relations among their elements. A set C with ordering relation $\leq$ is a *partially ordered set*, also called *poset*, and it is denoted as $\langle C, \leq\rangle$. A *chain* is a poset $\langle C, \leq\rangle$ which satisfies the following property: for any pair $x, y \in C$, $x \leq y$ or $y \leq x$. In that case, $\leq$ is a total order over C, but not necessarily over all sets.

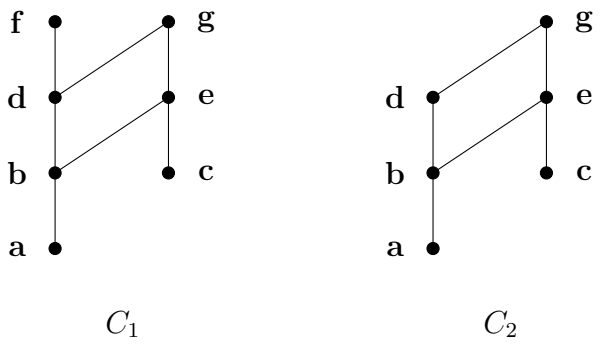


Fig. 2.1. Examples of ordered sets

Definition 2.3. *Let $\langle C, \leq \rangle$ be a poset, and let $X \subseteq C$. An element a is an* upper bound *of X if $\forall x \in X,\ x \leq a$. When $a \in X$, it is* maximal. *The smallest element of the set of upper bounds of X, when it exists, is called the* least upper bound *(*lub*) of X, and it is denoted as $\bigvee X$. When the* lub *belongs to C it is called* maximum *(or* top*) and it is usually denoted as $\top$.*

Consider the ordered sets C_1 and C_2 in Fig. 2.1. We observe that the set $\{a, b, c\}$ has *lub* e and C_1 has maximals f and g and no greatest element, while C_2 has greatest element g. The notions of *lower bound*, *minimal element*, *greatest lower bound* (*glb*) of a set X, denoted $\bigwedge X$, and *minimum* (or *bottom*), denoted by $\bot$ are dually defined. If a poset has a top (or bottom) element from the antisymmetry property of the ordering relation, it is unique. In the following we use $x \wedge y$ and $x \vee y$ to denote respectively the elements $\wedge\{x, y\}$ and $\vee\{x, y\}$.

Algebraic ordered structures can be further characterized. A poset $\langle C, \leq \rangle$ is a *direct set* if each non-empty finite subset of C has the least upper bound in C. A typical example of a direct set is a chain. A *complete partial order* (or *cpo*) is a poset $\langle C, \leq \rangle$ such that $\bot \in C$ and for each direct set D in C, there exists $\bigvee D$. It is clear that every finite poset is a cpo. Moreover, it holds that a poset C is a cpo if and only if each chain in C has the least upper bound.

Definition 2.4. *A poset $\langle C, \leq \rangle$, with $C \neq \emptyset$, is a* lattice *if for each $x, y \in C$, their* lub *$(x \vee y)$ and* glb *$x \wedge y$ both belong to C. A lattice is* complete *if for every subset $S \subseteq C$ we have that $\bigvee S \in C$ and $\bigwedge S \in C$.*

As usual, a complete lattice C with ordering relation $\leq$, *lub* $\vee$, *glb* $\wedge$, top element $\top = \vee C$ and bottom element $\bot = \wedge C$ is denoted as $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$. Often, $\leq_C$ is used to denote the underlying ordering of poset C, and $\vee_C$, $\wedge_C$, $\top_C$ and $\bot_C$ denote the basic operations and elements of a complete lattice C. It is worth noting that the ordered sets in Fig. 2.1 are not lattices since elements a and c do not have *glb*. The set $\mathbb{N}$ of natural numbers with the standard ordering relation is a lattice where the *glb* and the *lub* of a set are given respectively by its minimum and maximum element. However, $\langle \mathbb{N}, \leq \rangle$ is not complete because any infinite subset of $\mathbb{N}$, as for example $\{n \in \mathbb{N} \mid n > 100\}$, has no *lub*. On the other hand, an example of complete lattice often used in this thesis, is the powerset

$\wp(X)$, where X is any set. In this case the ordering is given by set inclusion, the *glb* by the intersection of sets and the *lub* by the union of sets.

Definition 2.5. *A sequence* $\{x_i\}_{i\in\mathbb{N}}$ *of elements in a poset* $\langle C, \leq\rangle$ *is an* ascending chain *if* $n \leq m \Rightarrow x_n \leq x_m$. *That sequence* eventually stabilizes *iff* $\exists n_0 \in \mathbb{N}.\forall n \in \mathbb{N}.n \geq n_0 \Rightarrow x_n = x_{n_0}$. *The poset* $\langle C, \leq\rangle$ *satisfies the* ascending chain contidition *(ACC) if every ascending chain of* C *eventually stabilizes.*

For instance, the ordered set of even numbers $\{n \in \mathbb{N} \mid n \bmod 2 = 0\}$ does not satisfy the ACC condition, since the ascending chain of even numbers does not converge.

Functions on Domains

Definition 2.6. *Let* $\langle C, \leq_C\rangle$ *and* $\langle D, \leq_D\rangle$ *be two cpos, and let* $f : C \to D$ *be a function. Then, we say:*

- f *is* monotonic *if for each* $x, y \in C$ *such that* $x \leq_C y$, *we have that* $f(x) \leq_D f(y)$;
- f *is (*Scott*)*-continuous *if it is monotonic and if it preserves the limits of direct sets, i.e., if for each direct set* X *of* C, *we have* $f(\bigvee_C X) = \bigvee_D f(X)$;
- f *is (*completely*)* additive *if for each subset* $X \subseteq C$, *we have that* $f(\bigvee_C X) = \bigvee_D f(X)$;
- f *is (*completely*)* co-additive *if for each subset* $X \subseteq C$, *we have that* $f(\bigwedge_C X) = \bigwedge_D f(X)$.

It is worth noting that an additive function preserves the limits (*lub*) of all subsets of C (empty set included), meaning that an additive function is also continuous.

Fixpoints

Definition 2.7. *Let* $f : C \to C$ *be a function on a poset* C. *An element* $x \in C$ *is a* fixpoint *of* f *if* $f(x) = x$. *Let* $\text{Fix}(f) = \{x \in C \mid f(x) = x\}$ *be the set of all fixpoints of function* f.

Thanks to the ordering relation $\leq_C$ on C, we can define the least fixpoint of f, denoted $lfp_{\leq_C}(f)$ (or simply $lfp(f)$ when the ordering relation is clear from the context), as the unique element $x \in \text{Fix}(f)$ such that for all $y \in \text{Fix}(f)$, $x \leq_C y$. The notion of the greatest fixpoint, denoted $gfp_{\leq_C}(f)$ (or simply $gfp(f)$ when the ordering relation is clear from the context), is dually defined. Let us recall the well-known Knaster-Tarski's fixpoint theorem [88].

Theorem 2.8 (Knaster-Tarski). *Given a complete lattice* $\langle C, \leq, \vee, \wedge, \top, \bot\rangle$ *and a monotone function* $f : C \to C$, *then the set of fixpoints of* f *is a complete lattice with ordering* $\leq$. *In particular, if* f *is continuous, the least fixpoint can be characterized as:*

$$lfp(f) = \bigvee_{n\leq\omega} f^n(\bot),$$

where, given $x \in C$, *the* i*-th power of* f *in* x *is inductively defined as follows:*

$$f^i(x) = \begin{cases} x & \text{if } i = 0 \\ f(f^{i-1}(x)) & \text{otherwise.} \end{cases}$$

Hence, the least fixpoint of a continuous function on a complete lattice can be computed as the limit of the iteration sequence obtained starting from the bottom of the lattice. Dually, the greatest fixpoint of a co-continuous function f on a complete lattice C, can be computed staring from the top of the lattice, namely $gfp(f) = \bigwedge_{n \leq \omega} f^n(\top)$.

Galois Connections

Another notion typically used in abstract interpretation is the Galois connection.

Definition 2.9. *Two posets $\langle C, \leq_C \rangle$ and $\langle D, \leq_D \rangle$ and two monotonic functions $\alpha : C \to D$ and $\gamma : D \to C$ such that:*

- *for each $c \in C$, $c \leq_C \gamma\alpha(c)$ and*
- *for each $d \in D$, $\alpha\gamma(d) \leq_D d$,*

form a Galois connection*, denoted by $\langle C, \alpha, \gamma, D \rangle$.*

The definition of Galois connection is equivalent to the one of *adjunction* between C and D, where $\langle C, \alpha, \gamma, D \rangle$ is an adjunction if:

$$\forall c \in C, \forall d \in D. \alpha(c) \leq_D d \Leftrightarrow c \leq_C \gamma(d).$$

In this case α (respectively γ) is called the *right adjoint* (respectively *left adjoint*) of γ (respectively α).

It is possible to prove that given a Galois connection $\langle C, \alpha, \gamma, D \rangle$, each function can be uniquely determined by the other one. In fact given $c \in C$ and $d \in D$ we have that:

- $\alpha(c) = \wedge_D \{y \in D \mid c \leq_C \gamma(y)\}$;
- $\gamma(d) = \vee_C \{x \in C \mid \alpha(x) \leq_D d\}$.

Thus, in order to specify a Galois connection it is enough to provide the right or left adjoint since the other one is uniquely determined by the above equalities. Moreover, it has been proved that given a Galois connection $\langle C, \alpha, \gamma, D \rangle$, the function α preserves existing *lub* (i.e., if $X \subseteq C$ and $\exists \bigvee_C X \in C$ then $\exists \bigvee_D \alpha(X) \in D$ and $\alpha(\bigvee_C X) = \bigvee_D \alpha(X)$) and γ preserves existing *glb*. In particular, when C and D are complete lattices we have that α is additive and γ is co-additive. Thus, given two complete lattices C and D, each additive function $\alpha : C \to D$ or co-additive function $\gamma : D \to C$ determines a Galois connection $\langle C, \alpha, \gamma, D \rangle$ where:

- for each $y \in D$, $\gamma(y) = \bigvee_C \{x \in C \mid \alpha(x) \leq_D y\}$;
- for each $x \in C$, $\alpha(x) = \bigwedge_D \{y \in D \mid x \leq_C \gamma(y)\}$.

This means that, α maps each element $c \in C$ in the smallest element in D whose image under γ is greater than c with respect to $\leq_C$. Vice versa, γ maps each element $d \in D$ in the greatest element in C whose image under α is lower than d with respect to $\leq_D$.

2.2 Abstract Interpretation

According to a widely recognized description, "*Abstract interpretation is a general theory for approximating the semantics of discrete dynamic systems*" [31]. The key idea of abstract interpretation is that the behaviour of a program at different levels of abstraction is

an approximation of its (concrete) semantics. Let $\mathbf{S} : \mathbb{P} \to \mathsf{C} \to \mathsf{C}$ denote a formal definition of the semantics of programs in $\mathbb{P}$ written in a certain programming language, where C is the semantic domain on which $\mathbf{S}$ is computed. Let us denote with $\mathbf{S}^\sharp : \mathbb{P} \to \mathsf{A} \to \mathsf{A}$ an abstract semantics expressing an approximation of the concrete semantics $\mathbf{S}$. The definition of the abstract semantics $\mathbf{S}^\sharp$ is given by the definition of the concrete semantics $\mathbf{S}$ where the domain C has been replaced by an approximated semantic domain A in Galois connection with C, i.e., $\langle \mathsf{C}, \alpha, \gamma, \mathsf{A} \rangle$. Then, the abstract semantics is obtained by replacing any function $F : \mathsf{C} \to \mathsf{C}$, used to compute $\mathbf{S}$, with an approximated function $F^\sharp : \mathsf{A} \to \mathsf{A}$ that correctly mimics the behaviour of F in the domain properties expressed by A.

Concrete and Abstract Domains

Let $\mathbf{S}[\![P]\!] : \mathsf{C} \to \mathsf{C}$ denote the concrete semantics $\mathbf{S}$ of a program $P \in \mathbb{P}$, which is computed on the so-called *concrete domain*, i.e., the poset $\langle \mathsf{C}, \leq_\mathsf{C} \rangle$ of mathematical objects used by P. The ordering relation encodes relative precision: $c_1 \leq_\mathsf{C} c_2$ means that $c_1 \in \mathsf{C}$ is a more precise (concrete) description than $c_2 \in \mathsf{C}$. For instance, the concrete domain for a program with integer variables is simply given by the powerset of integer numbers ordered by subset inclusion $\langle \wp(\mathbb{Z}), \subseteq \rangle$.

Approximation is encoded by an *abstract domain* $\langle \mathsf{A}, \leq_\mathsf{A} \rangle$, which is a poset of abstract values that represent some approximated properties of concrete objects from C. The ordering relation $\leq_\mathsf{A}$ models relative precision too: $a_1 \leq_\mathsf{A} a_2$ means that a_1 is a better approximation (i.e., more precise) than a_2. For example, we may be interested in the sign of an integer variable, so that a simple abstract domain for this property may be $\mathit{Sign} = \{\top, 0-, 0, 0+, \bot\}$, where $\top$ gives no sign information, $0-$, 0 and $0+$ state that the integer variable is negative, zero and positive, while $\bot$ represents an uninitialized variable or an error for a variable (e.g., division by zero): thus, we have that $\bot \leq 0 \leq 0- \leq \top$ and $\bot \leq 0 \leq 0+ \leq \top$, so that, in particular, the abstract values $0-$ and $0+$ are incomparable.

We recall that the concrete domain C and the abstract domain A are related through a Galois connection $\langle \mathsf{C}, \alpha, \gamma, \mathsf{A} \rangle$. In this case, $\alpha : \mathsf{C} \to \mathsf{A}$ is called the *abstraction map* and $\gamma : \mathsf{A} \to \mathsf{C}$ is called the *concretization map*. Moreover, we say that A is an *abstraction* (or *abstract interpretation*) of C, and that C is a *concretization* of A. The abstraction and concretization maps express the meaning of the abstraction process: $\alpha(c)$ is the abstract representation of c, and $\gamma(a)$ represents the concrete meaning of a. Thus, $\alpha(c) \leq_\mathsf{A} a$ and, equivalently, $c \leq_\mathsf{C} \gamma(a)$ mean that a is a sound approximation in A of c. Galois connections ensure that $\alpha(c)$ actually provides the best possible approximation in the abstract domain A of the concrete value $c \in \mathsf{C}$. In the abstract domain Sign, for example, we have that $\alpha(\{1, 5\}) = 0-$ while $\alpha(\{1, +1\}) = \top$.

Abstract domains can be compared with respect to their relative degree of precision: if A_1 and A_2 are both abstract domains of a common concrete domain C, we have that A_1 is more precise than A_2, when for any $a_2 \in \mathsf{A}_2$ there exists $a_1 \in \mathsf{A}_1$ such that $\gamma_1(a_1) = \gamma_2(a_2)$, i.e., when $\gamma_2(\mathsf{A}_2) \subseteq \gamma_1(\mathsf{A}_1)$. This ordering relation on the set of all possible abstract domains defines the lattice of abstract interpretations.

Soundness

In abstract interpretation, a concrete semantic operation is formalized as any (possibly n-ary) function $f : \mathsf{C} \to \mathsf{C}$ on the concrete domain C. For example, an integer squaring

operation sq on the concrete domain $\wp(\mathbb{Z})$ is given by $sq(X) = \{x^2 \in \mathbb{Z} \mid x \in X\}$, while an integer increment (by one) operation inc is given by $inc(X) = \{x + 1 \in \mathbb{Z} \mid x \in X\}$. A concrete semantic operation must be approximated on some abstract domain A by a sound abstract operation $f^\sharp : \mathsf{A} \to \mathsf{A}$. This means that $f^\sharp$ must be a correct approximation of f in A: for any $c \in \mathsf{C}$ and $a \in \mathsf{A}$, if a approximates c then $f^\sharp(a)$ must approximate $f(c)$. This is therefore encoded by the condition:

$$\forall c \in \mathsf{C}.\alpha(f(c)) \leq_{\mathsf{A}} f^\sharp(\alpha(c)).$$

For example, a *sound* approximation $sq^\sharp$ of sq on the abstract domain $Sign$ can be defined as follows:

$$sq^\sharp(a) = \begin{cases} \bot & \text{if } a = \bot \\ 0 & \text{if } a = 0 \\ 0+ & \text{if } a \in \{0-, 0+\} \\ \top & \text{if } a = \top, \end{cases}$$

while a correct approximation $inc^\sharp$ of inc on $Sign$ is given by:

$$inc^\sharp(a) = \begin{cases} \bot & \text{if } a = \bot \\ 0+ & \text{if } a \in \{0, 0+\} \\ \top & \text{if } a \in \{0-, \top\}. \end{cases}$$

Soundness can be also equivalently stated in terms of the concretization map:

$$\forall a \in \mathsf{A}.f(\gamma(a)) \leq_{\mathsf{C}} \gamma(f^\sharp(a)).$$

Given a concrete operation $f : \mathsf{C} \to \mathsf{C}$, we can order the correct approximations of f with respect to $\langle \mathsf{C}, \alpha, \gamma, \mathsf{A} \rangle$: let $f_1^\sharp$ and $f_2^\sharp$ be two correct approximations of f in A, then $f_1^\sharp$ is a better approximation of $f_2^\sharp$ if for every $a \in \mathsf{A}$, $f_1^\sharp(a) \leq_{\mathsf{A}} f_2^\sharp(a)$. Hence, if $f_1^\sharp$ is better than $f_2^\sharp$, it means that, given the same input, the output of $f_1^\sharp$ is more precise than the one of $f_2^\sharp$. It is well known that, given a concrete function $f : \mathsf{C} \to \mathsf{C}$ and a Galois connection $\langle \mathsf{C}, \alpha, \gamma, \mathsf{A} \rangle$, there exists the *best correct approximation* of f on A. Namely, it is possible to show that $\alpha \circ f \circ \gamma : \mathsf{A} \to \mathsf{A}$ is a correct approximation of f on A, and that for every correct approximation $f^\sharp$ of f we have that:

$$\forall a \in \mathsf{A}.\alpha(f(\gamma(a))) \leq_{\mathsf{A}} f^\sharp(a).$$

It is worth noting that the definition of the best correct approximation only depends on the structure of the underlying abstract domain, namely the best correct approximation of any concrete function is uniquely determined by the Galois connection $\langle \mathsf{C}, \alpha, \gamma, \mathsf{A} \rangle$. This is a theoretical result since, very often, in the real-life settings the best correct approximation is not computable, so we accept some other, less precise, but still sound approximations.

Abstract Semantics

One of the most important applications of abstract interpretation theory is the systematic design of approximate semantics of programs. Consider a Galois connection $\langle \mathsf{C}, \alpha, \gamma, \mathsf{A} \rangle$

and the concrete semantics $\mathbf{S}$ of programs $\mathbb{P}$ computed on the concrete domain C. As usual, the semantics obtained by replacing C with one of its abstractions A, and each function F defined on C with a corresponding correct approximation $F^\sharp$ on A, is called the *abstract semantics*. The abstract semantics $\mathbf{S}^\sharp$, as well as abstract functions, has to be correct with respect to the concrete semantics $\mathbf{S}$, that is for every program $P \in \mathbb{P}$, $\alpha(\mathbf{S}[\![P]\!])$ has to be an approximation of $\mathbf{S}^\sharp[\![P]\!]$. Let us consider the concrete semantics $\mathbf{S}[\![P]\!]$ of program P given, as usual, in fixpoint form $\mathbf{S}[\![P]\!] = \mathit{lfp}_{\leq_\mathsf{C}} F(P)$, where the semantic transformer $F : \mathsf{C} \to \mathsf{C}$ is monotonic and defined on the concrete domain of objects C. The abstract semantics $\mathbf{S}^\sharp[\![P]\!]$ can be computed as $\mathit{lfp}_{\leq_\mathsf{A}} F^\sharp$, where $F^\sharp = \alpha \circ F \circ \gamma$ is given by the best correct approximation of F in A. In this case soundness is guaranteed, i.e., $\alpha(\mathit{lfp}_{\leq_\mathsf{C}}(F)) \leq_\mathsf{A} \mathit{lfp}_{\leq_\mathsf{A}} F^\sharp$, or equivalently $\alpha(\mathbf{S}[\![P]\!]) \leq_\mathsf{A} \mathbf{S}^\sharp[\![P]\!]$. Thus, a correct approximation of the concrete semantics $\mathbf{S}$ can be systematically derived by computing the least fixpoint of the best correct approximation of F on the abstract domain A. The following well known result (see e.g. [20,34]) states that if the monotonic function $F : \mathsf{C} \to \mathsf{C}$ is sound with respect to an abstract domain A, then the abstract semantics is sound as well.

Theorem 2.10 (Fixpoint transfer). *Given a Galois connection $\langle \mathsf{C}, \alpha, \gamma, \mathsf{A} \rangle$ and a concrete monotonic function $F : \mathsf{C} \to \mathsf{C}$, if $\alpha \circ F \leq_\mathsf{A} F^\sharp \circ \alpha$, then $\alpha(\mathit{lfp}_{\leq_\mathsf{C}} F) \leq_\mathsf{A} \mathit{lfp}_{\leq_\mathsf{A}} F^\sharp$.*

3

Syntax and Semantics of a Java bytecode-like language

This chapter formally introduces the operational semantics of a Java bytecode-like language inspired by its standard informal semantics in [52]. This is the same semantics used in [63, 64, 86]. A similar formalization, but in a denotational form, has been used in [69, 84, 87]. Another representation of bytecode instructions in an operational setting, similar to ours, has been used in [17], although, there, Prolog clauses encode the graph, while we work directly on it.

There exist some other formal semantics for Java bytecode, but this choice has been dictated by the desire of a semantics suitable for abstract interpretation: there is only one concrete domain to abstract (the domain of states) while the bytecode instructions are always state transformers, even in the case of the conditional bytecode instructions and those dealing with dynamic dispatch and exception handling. This is exactly the purpose of the semantics in [86] whose form simplifies the definition of the abstract interpretation and its proof of soundness.

This formalization analyzes programs at bytecode level for several reasons:

- there is a small number of bytecode instructions, compared to varieties of source statements;
- bytecode lacks complexities such as inner classes;
- implementations of our analyses are at bytecode level as well, which brings formalism, implementation, and correctness proofs closer.

In order to formally define the operational semantics of the analyzed programs, Section 3.1 introduces types and values supported by this formalization, Section 3.2 introduces the notion of frame and type environment, widely used in the remainder of this thesis, and Section 3.3 introduces the bytecode instructions of the target language, representing an abstraction of Java virtual machine's bytecode instructions. The notion of state, semantics of bytecode instructions as well as the operational semantics of the target language are introduced in Section 3.4.

3.1 Types and Values

Like both the Java programming language and the Java virtual machine, this formalization deals with two kinds of types: *primitive types* and *reference types*. Consequently, there

are two kinds of values that can be stored inside the program variables, and that can be used for the computations inside the program: *primitive values* and *reference values*. This section introduces the primitive and the reference types supported by the target language.

The only supported primitive type is `int`, since this thesis is mainly concerned with the memory-related static analyses of Java bytecode programs. The other primitive types have the same behavior as the `int` type when it is dealt with the memory-related properties and therefore, for simplicity, they are not considered here[1].

For simplicity, the only reference type in this formalization are *classes* containing *instance fields* and *instance methods* only and *array types*[2]. It is worth noting that, on the other hand, the implementation of different analyses obtained from this framework handles all of the Java virtual machine's primitive and reference types.

The following definition formally defines the primitive and reference types the present formalization deals with, as well as the partial ordering among these types.

Definition 3.1 (Types). *Let* $\mathbb{K}$ *be the set the set of* classes *of a program. Every class has at most one* direct superclass *and an arbitrary number of* direct subclasses. *Let* $\mathbb{A}$ *be the set of all the* array types *of the program. A* type *is an element of* $\mathbb{T} = \{\texttt{int}\} \cup \mathbb{K} \cup \mathbb{A}$. *A class* $\kappa \in \mathbb{K}$ *has* instance fields $\kappa.f : \mathsf{t}$ *(a field* f *of type* $\mathsf{t} \in \mathbb{T}$ *defined in* κ*), where* κ *and* t *are often omitted. We let* $\mathsf{F}(\kappa) = \{\kappa'.f : \mathsf{t}' \mid \kappa \leq \kappa'\}$ *denote the fields defined in* κ *or in any of its superclasses. A class* $\kappa \in \mathbb{K}$ *has* instance methods $\kappa.m(\vec{\mathsf{t}}) : \mathsf{t}$ *(a method* m*, defined in* κ*, with parameters 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 with the special name* init*, which return* void*. Elements of an array type* $\alpha = \mathsf{t}[\,]$ *are subtypes of* t.

The set of types are ordered by a partial order $\leq$ introduced in the following definition.

Definition 3.2 (Partial ordering). *Given two types* $\mathsf{t}, \mathsf{t}' \in \mathbb{T}$*, we say that* t *is a* subtype *of* t'*, or equivalently that* t' *is a* supertype *of* t *and we denote it by* $\mathsf{t} \leq \mathsf{t}'$ *if one of the following conditions is satisfied:*

- $\mathsf{t} = \mathsf{t}'$ *or*
- $\mathsf{t}, \mathsf{t}' \in \mathbb{K}$ *and* t *is a subclass of* t' *or*
- $\mathsf{t} = \mathsf{t}_1[\,], \mathsf{t}' = \mathsf{t}'_1[\,] \in \mathbb{A}$*, and* $\mathsf{t}_1 \leq \mathsf{t}'_1$.

In the following we show some interesting properties of the subtype relation $\leq$. These properties will be used in the following chapters. First of all, we show that two supertypes of the same type must be related through $\leq$.

[1] According to [52], the primitive data types supported by the Java virtual machine are the *numeric types*, which may be *integral* and *floating-point*, the `boolean` type and the `returnAddress` type. The integral types are `byte`, `short`, `int`, `long`, whose values are respectively 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers. The floating-point types are `float` and `double`, whose values are taken from the float and the double value set respectively. The values of the `boolean` type are *true* and *false*, while the values of the `returnAddress` type are pointers to the operation codes (opcodes) of Java virtual machine instructions.

[2] The reference types supported by the Java virtual machine are *class types*, *array types* and *interface types*. The values of these types are, respectively, references to dynamically created class instances, arrays and class instances or arrays that implement interfaces. Values of type reference can be thought of as pointers to objects. There exists another reference value, the special `null` reference, pointing to no object. This value has no run-time type and can be cast to any reference type. It is also the default value of all reference types.

Lemma 3.3. *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 at least one of the following relations holds:* $\mathsf{t}' \leq \mathsf{t}''$ *or* $\mathsf{t}'' \leq \mathsf{t}'$.

Proof. We distinguish the following cases:

- If $\mathsf{t} = \texttt{int}$ then $\mathsf{t}' = \mathsf{t}'' = \texttt{int}$ which falsifies the hypothesis $\mathsf{t}' \neq \mathsf{t}''$.
- If t, t' and t'' are classes, then 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 latter case we have $\mathsf{t}' \leq \mathsf{t}''$, and in the former case $\mathsf{t}'' \leq \mathsf{t}'$.
- If $\mathsf{t} = \mathsf{t}_1 \underbrace{[\,]\ldots[\,]}_{n}, \mathsf{t}' = \mathsf{t}'_1 \underbrace{[\,]\ldots[\,]}_{n}, \mathsf{t}'' = \mathsf{t}''_1 \underbrace{[\,]\ldots[\,]}_{n} \in \mathbb{A}$, where $\mathsf{t}_1, \mathsf{t}'_1, \mathsf{t}''_1 \in \mathbb{K}$ and $\mathsf{t}'_1 \neq \mathsf{t}''_1$. Since $\mathsf{t} \leq \mathsf{t}'$ and $\mathsf{t} \leq \mathsf{t}''$ we have, by Definition 3.2, $\mathsf{t}_1 \leq \mathsf{t}'_1$ and $\mathsf{t}_1 \leq \mathsf{t}''_1$. We have already shown that in this case $\mathsf{t}'_1 \leq \mathsf{t}''_1$ or $\mathsf{t}''_1 \leq \mathsf{t}'_1$ holds. Again by Definition 3.2, we obtain that also $\mathsf{t}' \leq \mathsf{t}''$ or $\mathsf{t}'' \leq \mathsf{t}'$ holds.

■

We say that two types are compatible if they are related through $\leq$.

Definition 3.4 (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:*

$$\mathsf{compatible}(\mathsf{t}) = \{\mathsf{t}' \mid \mathsf{t} \leq \mathsf{t}' \ \vee \ \mathsf{t}' \leq \mathsf{t}\}.$$

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 3.5. *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 3.3 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 3.6. *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 3.3 we have $\mathsf{t} \leq \mathsf{t}''$ or $\mathsf{t}'' \leq \mathsf{t}$ i.e., $\mathsf{t} \in \mathsf{compatible}(\mathsf{t}'')$.

■

3.2 Frames

In this section we define a notion of *frame*. Frames are used to store data and partial results, to handle methods' return values and to manage exceptions. Any time a method is invoked, a new frame is created, and it is destroyed at the end of the method's execution, whether it is normal or exceptional end. Only one frame can be active at any point of the program's execution. That frame concerns the method which is being executed and it ceases to be active if it invokes another method or terminates, normally or exceptionally. In the case the method terminates, it passes back the result to the frame from which it has been invoked, if any, reactivate that frame, and it is definitely deactivated.

Each frame contains its own *array of local variables* L and its own *operand stack* S, whose sizes are determined at compile-time. Local variables ($L = \{l_0, l_1, \ldots\}$) are addressed by indexing: the index of the first local variable is zero. The Java virtual machine uses local variables to pass parameters on method invocation. When an instance method is invoked, local variable 0 (l_0) is always used to pass a reference to the object on which the instance method is being invoked (this in the Java programming language). Any parameters are subsequently passed in consecutive local variables starting from local variable 1 (l_1). Operand stack is a LIFO stack, which is empty when its corresponding frame is created. Some instructions of our target Java bytecode-like language load constants or values from the local variables or from the fields onto the operand stack, while some other instructions take operands from the operand stack, operate on them and push the result back onto the operand stack. It is also used to prepare parameters to be passed to methods and to receive method results.

At any point of execution, we know the exact length of both array of local variables and operand stack. Moreover, a standard algorithm [52] infers their static types. These static types are returned by the *type environment* map.

Definition 3.7 (Type environment). *Each program point is enriched with a* type environment τ*, i.e., a map from all the variables available at that point (*$\mathsf{dom}(\tau)$*) to their static types. We distinguish between* local variables $L = \{l_0, \ldots\}$ *and* operand stack elements $S = \{s_0, \ldots\}$*. For simplicity, we write* $V = \{v_0, \ldots, v_{|L|+|S|-1}\}$*, where* $v_r = l_r$ *if* $0 \leq r < |L|$ *and* $v_r = s_{r-|L|}$ *if* $|L| \leq r < |L| + |S|$*. Moreover, we let* $|\tau|$ *denote* $|\mathsf{dom}(\tau)| = |L| + |S|$ *and we let* $\mathcal{T}$ *denote the set of all type environments.*

3.3 Syntax

Instructions of our Java bytecode-like target language are called *bytecode instructions*. They consist of an opcode of the actual instruction, representable by one byte, followed by zero or more parameters. These parameters might be the arguments passed to the instruction itself, data that the instruction should operate on, or just static types of the parameters. In Fig. 3.1 we give the list of all the bytecode instructions of our target language divided in 7 categories.

3.3.1 Load and Store Instructions

The load and store instructions transfer values between the local variables and the operand stack of a frame:

LOAD AND STORE INSTRUCTIONS:	const x, load k t, store k t
ARITHMETIC INSTRUCTIONS:	inc k x, add, sub, mul, div, rem
OBJECT CREATION AND MANIPULATION INSTRUCTIONS:	new κ, getfield f, putfield f
ARRAY CREATION AND MANIPULATION INSTRUCTIONS:	arraynew α, arraylength α, arrayload α, arraystore α
OPERAND STACK MANAGEMENT INSTRUCTIONS:	dup t
CONTROL TRANSFER INSTRUCTIONS:	ifeq t, ifne t
METHOD INVOCATION AND RETURN INSTRUCTIONS:	call, return void, return t
EXCEPTION HANDLING INSTRUCTIONS:	throw κ, catch, exception_is K

Fig. 3.1. Instruction set summary

- bytecode instruction load k t loads the local variable l_k whose static type is t onto the operand stack;
- bytecode instruction store k t stores a value from the operand stack whose static type is t into the local variable l_k;
- bytecode const x loads an integer constant or `null` onto the operand stack.

These bytecode instructions abstract whole classes of Java bytecode instructions such as iload, iload_⟨n⟩, aload, aload_⟨n⟩, istore, istore_⟨n⟩, astore, astore_⟨n⟩, bipush, sipush, iconst_⟨i⟩, aconst_null.

3.3.2 Arithmetic Instructions

The arithmetical bytecode instructions supported by our target language are functions which pop two topmost integer values from the operand stack, apply the corresponding arithmetic operation on them, and push back the result on the operand stack. They are:

- add, representing addition;
- sub, representing subtraction;
- mul, representing multiplication;
- div, representing division;
- rem, representing remainder operation;
- inc k x, representing incrementation of the value from the local variable l_k by x.

These bytecode instructions correspond to the following Java bytecode instructions: iadd, isub, imul, idiv, irem and iinc.

3.3.3 Object Creation and Manipulation Instructions

Our target languages supports the following object creation and manipulation bytecode instructions:

- bytecode instruction new κ creates a new instance of class κ;
- bytecode instructions getfield f reads a value from the field f belonging to the class κ and whose static type is t;
- bytecode instructions putfield f writes a value into the field f belonging to the class κ and whose static type is t.

These bytecode instructions correspond to the following Java bytecode instructions: new, getfield, putfield.

3.3.4 Array Creation and Manipulation Instructions

Our target languages supports the following array creation and manipulation bytecode instructions:

- bytecode instruction arraynew α creates a new array of array type α;
- bytecode instructions arraylength α puts the length of the array of array type α onto the operand stack;
- bytecode instructions arrayload α reads the value from the k-th element of an array of array type α, where an integer k and a reference to the array are popped from the operand stack, and puts that value onto the operand stack;
- bytecode instructions arraystore α pops a value from the operand stack and writes it into the k-th element of an array of array type α, where an integer k and a reference to the array are popped from the operand stack.

3.3.5 Operand Stack Management Instructions

The only operand stack management bytecode instruction supported by our target language is dup t which duplicates the topmost value of the operand stack.

3.3.6 Control Transfer Instructions

The control transfer bytecode instructions conditionally cause the Java virtual machine to continue execution with an instruction other than the one following the control transfer instruction. Our formalization supports the following bytecode instructions:

- ifeq t which pops a value from the operand stack whose static type is t and controls whether it is equal to the default value for that type;
- ifne t which pops a value from the operand stack whose static type is t and controls whether it is different from the default value for that type.

3.3.7 Method Invocation and Return Instructions

In our formalization, we use the bytecode instruction call to invoke an instance method of an object, dispatching on the static type of the object. This is the normal method dispatch in the Java programming language.

There are two types of the method return instructions:

- return t, which returns a value whose static type is t from the invoked instance method;
- return void, which is used to return from the instance methods declared to be void and from the class initialization methods (constructors).

3.3.8 Exception Handling Instructions

In our formalization, an exception is thrown programmatically using the throw κ bytecode instruction. Exceptions can also be thrown by various other bytecode instructions if they detect an abnormal condition.

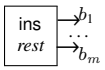
```
1  public class List {
2    public Object head;
3    public List tail;
4
5    public List() {
6      head = tail = null;
7    }
8
9    public List(Object head, List tail) {
10     this.head = head;
11     this.tail = tail;
12   }
13
14   public Object getFirst() {
15     return head;
16   }
17
18   public Object removeFirst() {
19     Object result = head;
20     if (tail != null) {
21       head = tail.head;
22       tail = tail.tail;
23     } else {
24       head = null;
25     }
26     return result;
27   }
28
29   public static void main(String[] args) {
30     ...
31     int n = Integer.valueOf(args[0]);
32     ...
33     List list = new List();
34     for (int i = 1; i <= n; i++) {
35       Object o = new Object();
36       List tmp = new List(o, list.tail);
37       list.tail = tmp;
38     }
39   }
40 }
```

Fig. 3.2. A list of objects

Bytecode catch starts an exception handler. It takes an exceptional state and transforms it into a normal one, subsequently used by the handler. After catch, bytecode exception_is K can be used to select an appropriate handler depending on the run-time class of the top of the stack: it filters those states whose top of the stack is an instance of a class in $K \subseteq \mathbb{K}$.

3.3.9 Control Flow Graph

We analyze bytecode preprocessed into a control flow graph (CFG), i.e., a directed graph of *basic blocks*, with no jumps inside them. We graphically write



to denote a block of code starting with a bytecode instruction ins, possibly followed by a sequence of instructions *rest* and linked to m subsequent blocks $b_1, \ldots, b_m$.

Example 3.8. Fig. 3.3 shows the basic blocks of the second constructor in Fig. 3.2 (lines 9-12). There is a branch at the *implicit* 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 of the constructor. Otherwise, the execution continues with 2 blocks storing the formal parameters (locals 1 and 2) into the fields of this (held in l_0) and then returns. Note that each bytecode instruction except return and throw has always one or more immediate successors, while return and throw are placed at the end of a method or constructor and have no successors. □

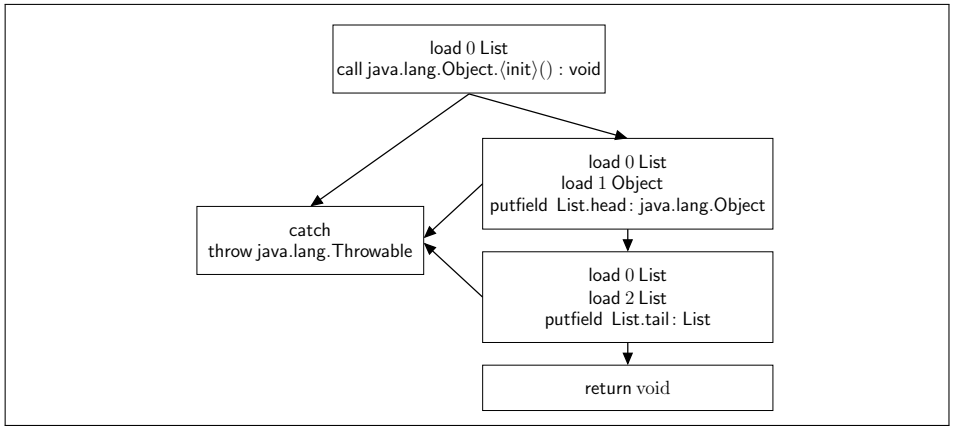
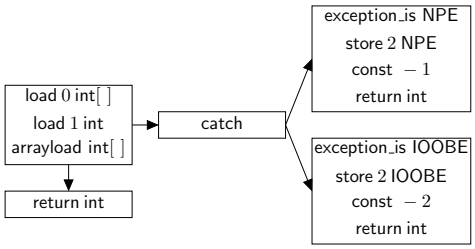


Fig. 3.3. Our representation of the code of the second constructor in Fig. 3.2

Example 3.9. In Fig. 3.4 a) we show a simple Java method `get` whose parameters are an array a of type `int[ ]` and an integer k, and which returns $a[k]$ if no exception occurs. This method handles two possible exceptions: `NullPossibleException` (`NPE` for short) when a is `null` and `IndexOutOfBoundsException` (`IOOBE` for short) when k is greater or equal to a's length. These two exceptions are not subclasses one of another, i.e., neither `NPE` $\leq$ `IOOBE` nor `NPE` $\leq$ `IOOBE` hold. In Fig. 3.4 b) we show our representation of the `get` method. There are two separate blocks handling the two exceptions: one starting with exception_is NPE handling the case of `NPE` and the other one starting with exception_is IOOBE handling the case of `IOOBE`. □

```
public static int get(int[] a, int k) {
 try {
  return a[k];
 } catch (NullPointerException e1) {
  return -1;
 } catch (IndexOutOfBoundsException e2) {
  return -2;
 }
}
```

a)



b)

Fig. 3.4. a) A simple Java method `get` handling two exceptions; **b)** Our representation of `get`, where NPE and IOOBE represent classes `NullPointerExcepiton` and `IndexOutOfBoundsException` respectively

3.4 Semantics

Our semantics keeps a *state* that maps program variables to values. An *activation stack* of *states* models the method call mechanism, exactly as in the actual implementation of the Java virtual machine [52].

3.4.1 States

Definition 3.10 (Values). *The set of all possible values that our formalization supports is* $\mathbb{V} = \mathbb{Z} \cup \mathbb{L} \cup \{\mathtt{null}\}$*, where for simplicity we use* $\mathbb{Z}$ *instead of* 32*-bit two's-complement integers as in the actual Java virtual machine (this choice is irrelevant in this thesis) and where* $\mathbb{L}$ *is an infinite set of* memory locations*.*

Objects are particular instances of classes. The way we represent them in this thesis is explained by the following definition.

Definition 3.11 (Object representation). *Given an object* o*, its type is determined by invoking* $o.\mathtt{type}$*, i.e., we can state that* o *is an* instance *of* $o.\mathtt{type}$*. Each object* o *contains its* internal environment $o.\phi$ *that maps every field available from that object* $\kappa'.f : \mathsf{t}' \in \mathsf{F}(o.\mathtt{type})$ *into its value, denoted by* $(o.\phi)(\kappa'.f{:}\mathsf{t}')$*. Therefore, the domain of* $o.\phi$*,* $\mathsf{dom}(o.\phi)$ *is* $\mathsf{F}(o.\mathtt{type})$*, while its range* $\mathsf{rng}(o.\phi)$ *is the set composed of the values of all the fields available in* o*.*

Arrays are particular instances of array types. The way we represent them in this thesis is explained by the following definition.

Definition 3.12 (Array representation). *Given an array a, its type is determined by invoking a.*`type`*, i.e., we can state that a is an* instance *of a.*`type`*. The length of a can be obtained by invoking a.*`length`*. Each array a contains an* internal environment *$a.\phi$ that maps each index $0 \leq i < a.\texttt{length}$ into the value of the element corresponding to that index, $(a.\phi)(i)$. Therefore, the domain of $a.\phi$, $\mathsf{dom}(a.\phi)$ is $\{0, \ldots, a.\texttt{length}-1\}$, while its range $\mathsf{rng}(a.\phi)$ is the set composed of the values of all the elements of a.*

We are interested in capturing the current configuration of the program under analysis at each point of its execution. These program configurations are specified by the notion of *state*.

Definition 3.13 (State). *A* state *σ over a type environment $\tau \in \mathcal{T}$ is a pair $\langle\langle \mathsf{l} \parallel \mathsf{s}\rangle, \mu\rangle$ where l is an array of values memorized in the local variables L of $\mathsf{dom}(\tau)$, s is a stack of values of the variables from the operand stack S 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 of states: $\langle \rho, \mu\rangle$, where an* environment *ρ maps each $l_k \in L$ to its value $\mathsf{l}[k]$ and each $s_k \in S$ to its value $\mathsf{s}[k]$. The set of states is Ξ. We write Ξ_τ when we want to fix the type environment τ.*

We assume that variables hold values consistent with their static types, i.e., states are well-typed.

Definition 3.14 (Consistent state). *We say that a value v is* consistent *with a type t in $\langle \rho, \mu\rangle$ and we denote it by $v \backsim_{\langle \rho,\mu\rangle} \mathsf{t}$ if one of the following conditions holds:*

- *$v \in \mathbb{Z}$ and $\mathsf{t} =$* `int` *or*
- *$v =$* `null` *and $\mathsf{t} \in \mathbb{K} \cup \mathbb{A}$ or*
- *$v \in \mathbb{L}$, $\mathsf{t} \in \mathbb{K}$ and $\mu(v).\texttt{type} \leq \mathsf{t}$ or*
- *$v \in \mathbb{L}$, $\mathsf{t} \in \mathbb{A}$ and $\mu(v).\texttt{type} \leq \mathsf{t}$.*

We write $v \not\backsim_{\langle \rho,\mu\rangle} \mathsf{t}$ to denote that v is not consistent with t in $\langle \rho, \mu\rangle$. In a state $\langle \rho, \mu\rangle$ over τ, we require that $\rho(v)$ is consistent with the type $\tau(v)$ for any variable $v \in \mathsf{dom}(\tau)$ available at that point, that for every object $o \in \mathsf{rng}(\mu)$ available in the memory in that moment, and every field $\kappa'.f : \mathsf{t}' \in \mathsf{F}(o.\texttt{type})$ available in that object, the value memorized in that field, $(o.\phi)(\kappa'.f : \mathsf{t}')$, is consistent with its static type t' and that for every array $a \in \mathsf{rng}(\mu)$ available in memory in that moment, values in $\mathsf{rng}(a.\phi)$ are consistent with t', where $a.\texttt{type} = \mathsf{t}'[\,]$.

The Java virtual machine(JVM), as well as our formalization, supports exceptions. Therefore, 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. The operand stack of the states in $\underline{\Xi}$ always has exactly 1 variable holding 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 $\langle\langle \mathsf{l} \parallel \mathsf{s}\rangle, \mu\rangle$ for a normal state and $\underline{\langle\langle \mathsf{l} \parallel \mathsf{s}\rangle, \mu\rangle}$ for an exceptional state.

Definition 3.15 (Java virtual machine state). *The set of* Java virtual machine 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 the operand stack containing only one variable (s_0) whose static type is a subclass of* `Throwable`*, i.e., $\tau'(s_0) \leq$* `Throwable`*.*

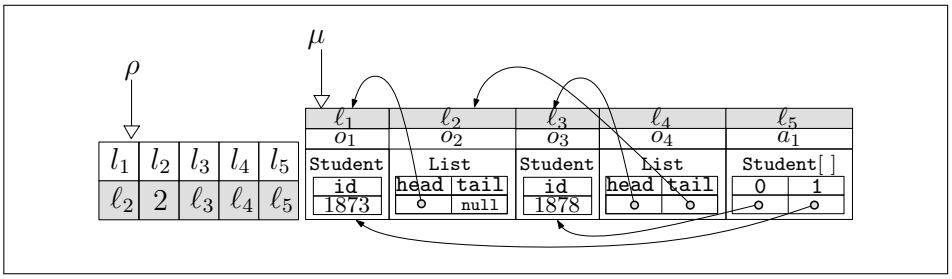


Fig. 3.5. A JVM state $\sigma = \langle \rho, \mu \rangle$

Example 3.16. Let `Student` be a class containing one instance field `id` of type `int`. Consider the following type environment:

$$\tau = [l_1 \mapsto \texttt{List}; l_2 \mapsto \texttt{int}; l_3 \mapsto \texttt{Student}; l_4 \mapsto \texttt{List}; l_5 \mapsto \texttt{Student[ ]}],$$

where `List` is the class defined in Fig. 3.2. In Fig. 3.5 we show a state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$. Environment ρ maps local variables l_1, l_2, l_3 and l_4 to values $\ell_2 \in \mathbb{L}$, $2 \in \mathbb{Z}$, $\ell_3 \in \mathbb{L}$, $\ell_4 \in \mathbb{L}$ and $\ell_5 \in \mathbb{L}$ respectively. Memory μ maps locations ℓ_2 and ℓ_4 to objects o_2 and o_4 of class `List`, location ℓ_3 to object o_3 of class `Student` and location ℓ_5 to array a_1 of array type `Student[ ]`. Objects are represented as boxes with a class tag and an internal environment mapping fields to values, while arrays are represented as boxes with an array type tag and an internal environment mapping indexes to values. For instance, fields `head` and `tail` of object o_4 contain locations ℓ_3 and ℓ_2 respectively, while indexes `0` and `1` of array a_1 contain locations ℓ_3 and ℓ_1 respectively, and its length is 2. □

3.4.2 Semantics of Bytecode Instructions

The semantics of a bytecode instruction ins is a partial map $ins : \Sigma_\tau \to \Sigma_{\tau'}$ 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 the type environment available at that point [52]. In the following we silently assume that bytecode instructions are run in a program point with type environment $\tau \in \mathcal{T}$ such that $\mathsf{dom}(\tau) = L \cup S$, where L and S are the array of local variables and the operand stack, and let $i = |L|$ and $j = |S|$ be the cardinalities of these sets. Moreover, we suppose that the semantics is undefined for input states of wrong sizes or types, as it is required in [52]. Fig. 3.6 defines the semantics of the bytecode instructions supported by our formalization and introduced in Section 3.3. We discuss it below.

const x is the semantics of the bytecode instruction const x. It pushes a value $x \in \mathbb{Z} \cup \{\texttt{null}\}$ onto the operand stack. Since $\langle \langle \mathsf{I} \parallel \mathsf{s} \rangle, \mu \rangle$ (where s might be ε) is not underlined, *const* x is undefined on exceptional states, i.e, const$x$ is run only when the Java virtual machine is in a normal state.

load k t is the semantics of the bytecode instruction load k t. It pushes onto the stack the value memorized into the local variable l_k, which must exist and have type t.

$$
\begin{array}{rl}
\textit{const } x = & \lambda\langle\langle \mathsf{l} \parallel \mathsf{s}\rangle,\mu\rangle \,.\, \langle\langle \mathsf{l} \parallel x :: \mathsf{s}\rangle,\mu\rangle \\
\textit{load } k\ \mathsf{t} = & \lambda\langle\langle \mathsf{l} \parallel \mathsf{s}\rangle,\mu\rangle \,.\, \langle\langle \mathsf{l} \parallel \mathsf{l}[k] :: \mathsf{s}\rangle,\mu\rangle \\
\textit{store } k\ \mathsf{t} = & \lambda\langle\langle \mathsf{l} \parallel t :: \mathsf{s}\rangle,\mu\rangle \,.\, \langle\langle \mathsf{l}[k \mapsto t] \parallel \mathsf{s}\rangle,\mu\rangle \\
\hline
\textit{add} = & \lambda\langle\langle \mathsf{l} \parallel t_1 :: t_2 :: \mathsf{s}\rangle,\mu\rangle \,.\, \langle\langle \mathsf{l} \parallel t_1 + t_2 :: \mathsf{s}\rangle,\mu\rangle \\
\textit{sub} = & \lambda\langle\langle \mathsf{l} \parallel t_1 :: t_2 :: \mathsf{s}\rangle,\mu\rangle \,.\, \langle\langle \mathsf{l} \parallel t_1 - t_2 :: \mathsf{s}\rangle,\mu\rangle \\
\textit{mul} = & \lambda\langle\langle \mathsf{l} \parallel t_1 :: t_2 :: \mathsf{s}\rangle,\mu\rangle \,.\, \langle\langle \mathsf{l} \parallel t_1 * t_2 :: \mathsf{s}\rangle,\mu\rangle \\
\textit{div} = & \lambda\langle\langle \mathsf{l} \parallel t_1 :: t_2 :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \parallel t_2/t_1\rangle,\mu\rangle & \text{if } t_1 \neq 0 \\ \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto ae]\rangle} & \text{otherwise} \end{cases} \\
\textit{rem} = & \lambda\langle\langle \mathsf{l} \parallel t_1 :: t_2 :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \parallel t_2 \% t_1\rangle,\mu\rangle & \text{if } t_1 \neq 0 \\ \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto ae]\rangle} & \text{otherwise} \end{cases} \\
\textit{inc } k\ x = & \lambda\langle\langle \mathsf{l} \parallel \mathsf{s}\rangle,\mu\rangle \,.\, \langle\langle \mathsf{l}[k \mapsto \mathsf{l}[k] + x] \parallel \mathsf{s}\rangle,\mu\rangle \\
\hline
\textit{new } \kappa = & \lambda\langle\langle \mathsf{l} \parallel \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \parallel \ell :: \mathsf{s}\rangle,\mu[\ell \mapsto o]\rangle & \text{if enough memory} \\ \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto oome]\rangle} & \text{otherwise} \end{cases} \\
\textit{getfield } \kappa.f{:}\mathsf{t} = & \lambda\langle\langle \mathsf{l} \parallel r :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \parallel (\mu(r).\phi)(f) :: \mathsf{s}\rangle,\mu\rangle & \text{if } r \neq \texttt{null} \\ \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto npe]\rangle} & \text{otherwise} \end{cases} \\
\textit{putfield } \kappa.f{:}\mathsf{t} = & \lambda\langle\langle \mathsf{l} \parallel t :: r :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \parallel \mathsf{s}\rangle,\mu[(\mu(r).\phi)(f) \mapsto t]\rangle & \text{if } r \neq \texttt{null} \\ \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto npe]\rangle} & \text{otherwise} \end{cases} \\
\hline
\textit{arraynew } \alpha = & \lambda\langle\langle \mathsf{l} \parallel n :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \parallel \ell :: \mathsf{s}\rangle,\mu[\ell \mapsto a]\rangle & \text{if } n \geq 0 \\ \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto nase]\rangle} & \text{otherwise} \end{cases} \\
\textit{arraylength } \alpha = & \lambda\langle\langle \mathsf{l} \parallel r :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \parallel \mu(r).\texttt{length} :: \mathsf{s}\rangle,\mu\rangle & \text{if } r \neq \texttt{null} \\ \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto npe]\rangle} & \text{otherwise} \end{cases} \\
\textit{arrayload } \alpha = & \lambda\langle\langle \mathsf{l} \parallel k :: r :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto obe]\rangle} & \text{if } k \geq \mu(r).\texttt{length} \text{ or } k < 0 \\ \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto npe]\rangle} & \text{if } r = \texttt{null} \\ \langle\langle \mathsf{l} \parallel (\mu(r).\phi)(k) :: \mathsf{s}\rangle,\mu\rangle & \text{otherwise} \end{cases} \\
\textit{arraystore } \alpha = & \lambda\langle\langle \mathsf{l} \parallel v :: k :: r :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto npe]\rangle} & \text{if } r = \texttt{null} \\ \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto obe]\rangle} & \text{if } k \geq \mu(r).\texttt{length} \text{ or } k < 0 \\ \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto ase]\rangle} & \text{if } v \in \mathbb{L} \text{ and} \\ & \quad \mu(v).\texttt{type}[] \not\leq \mu(r).\texttt{type} \\ \langle\langle \mathsf{l} \parallel \mathsf{s}\rangle,\mu[(\mu(r).\phi)(k) \mapsto v]\rangle & \text{otherwise} \end{cases} \\
\hline
\textit{dup } \mathsf{t} = & \lambda\langle\langle \mathsf{l} \parallel t :: \mathsf{s}\rangle,\mu\rangle \,.\, \langle\langle \mathsf{l} \parallel t :: t :: \mathsf{s}\rangle,\mu\rangle \\
\hline
\textit{ifeq } \mathsf{t} = & \lambda\langle\langle \mathsf{l} \parallel t :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \parallel \mathsf{s}\rangle,\mu\rangle & \text{if } t \in \{0, \texttt{null}\} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\textit{ifne } \mathsf{t} = & \lambda\langle\langle \mathsf{l} \parallel t :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \parallel \mathsf{s}\rangle,\mu\rangle & \text{if } t \notin \{0, \texttt{null}\} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\hline
\textit{return } \text{void} = & \lambda\langle\langle \mathsf{l} \parallel \mathsf{s}\rangle,\mu\rangle \,.\, \langle\langle \mathsf{l} \parallel \epsilon\rangle,\mu\rangle \\
\textit{return } \mathsf{t} = & \lambda\langle\langle \mathsf{l} \parallel t :: \mathsf{s}\rangle,\mu\rangle \,.\, \langle\langle \mathsf{l} \parallel t\rangle,\mu\rangle, \quad \text{where } \mathsf{t} \neq \text{void} \\
\hline
\textit{throw } \kappa = & \lambda\langle\langle \mathsf{l} \parallel t :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \underline{\langle\langle \mathsf{l} \parallel t\rangle,\mu\rangle} & \text{if } t \neq \texttt{null} \\ \underline{\langle\langle \mathsf{l} \parallel \ell\rangle,\mu[\ell \mapsto npe]\rangle} & \text{otherwise} \end{cases} \\
\textit{catch} = & \underline{\lambda\langle\langle \mathsf{l} \parallel t\rangle,\mu\rangle} \,.\, \langle\langle \mathsf{l} \parallel t\rangle,\mu\rangle \\
\textit{exception\_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).\texttt{type} \in K \\ \text{undefined} & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 3.6. Semantics of bytecode instructions: instructions are functions mapping states to states. $\ell \in \mathbb{L}$ is a fresh location. o is a new created object whose fields are initialized to their default values. a is a new created array whose elements are initialized to their default values. ae, $oome$, npe, $nase$, obe and ase are respectively new instances of the following exceptions: `ArithmeticException`, `OutOfMemoryError`, `NullPointerException`, `NegativeArraySizeException`, `ArrayIndexOutOfBoundsException` and `ArrayStoreException`

store k t is the semantics of the bytecode instruction store k t. It pops the topmost value from the operand stack and writes it into the local variable l_k. The topmost value must exist and have type t, and k must be a valid index of the array of local variables.

add is the semantics of the bytecode instruction add. It requires that two topmost values of the operand stack must be of type `int`. These values are popped from the operand stack, their sum is calculated and is pushed back onto the operand stack.

sub is the semantics of the bytecode instruction sub. It requires that two topmost values of the operand stack must be of type `int`. These values are popped from the operand stack, their difference is calculated and is pushed back onto the operand stack.

mul is the semantics of the bytecode instruction mul. It requires that two topmost values of the operand stack must be of type `int`. These values are popped from the operand stack, their product is calculated and is pushed back onto the operand stack.

div is the semantics of the bytecode instruction div. It requires that two topmost values of the operand stack must be of type `int`. These values are popped from the operand stack, their quotient is calculated and is pushed back onto the operand stack. It is worth noting that the first (second) topmost value from the operand stack is the divisor (dividend) of the operation. If the divisor is equal to 0, *div* throws an `ArithmeticException`.

rem is the semantics of the bytecode instruction rem. It requires that two topmost values of the operand stack must be of type `int`. These values are popped from the operand stack, the reminder of their division is calculated and is pushed back onto the operand stack. It is worth noting that the first (second) topmost value from the operand stack is the divisor (dividend) of the operation. If the divisor is equal to 0, *rem* throws an `ArithmeticException`.

inc k x is the semantics of the bytecode instruction inc k x. It requires that k is a valid index of the array of local variables and that the local variable l_k is of type `int`. Then, its value is incremented by $x \in \mathbb{Z}$ and written back in l_k.

new κ is the semantics of the bytecode instruction new κ. It pushes onto the operand 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 [52]. If the automatic storage manager was unable to reclaim enough memory to satisfy an object creation request, *new* κ throws an `OutOfMemoryException`.

getfield $\kappa.f$:t is the semantics of the bytecode instruction getfield f. It requires that the topmost value from the operand stack is a reference to an object whose class is a subclass of κ, and which contains a field f of type t. Under these hypotheses, the topmost value is popped from the operand stack, the value memorized in the field f of the corresponding object is read and written back onto the operand stack. If the topmost value of the operand stack is `null`, *getfield* $\kappa.f$:t throws a `NullPointerException`.

putfield $\kappa.f$:t is the semantics of the bytecode instruction putfield f. It requires that the topmost value from the operand stack is consistent with a subtype of t and that the second topmost value from the operand stack is a reference to an object whose class is a subclass of κ and which contains a field f of type t. Under these hypotheses, the two topmost values are popped from the operand stack and the first one is written in the field f of the object corresponding to the second one. If the second topmost value of the operand stack is `null`, *putfield* $\kappa.f$:t throws a `NullPointerException`.

arraynew α is the semantics of the bytecode instruction arraynew α. It pops the topmost element from the operand stack, which is an integer value, creates a new ar-

ray of array type α = t[ ] and pushes back onto the operand stack a reference to the new array. The length of the new created object is equal to the positive integer value popped from the stack. If that value is a negative number, *arraynew* α throws a `NegativeArraySizeException`. The elements of the new created array are initialized to the default value for t: `null` if t is a reference type, and 0 otherwise. Moreover, if the automatic storage manager was unable to reclaim enough memory to satisfy an array creation request, *arraynew* α throws an `OutOfMemoryException` (not shown in Fig. 3.6).

arraylength α is the semantics of the bytecode instruction arraylength α. It requires that the topmost value from the operand stack is a reference to an array whose array type is a subtyoe of α. Under these hypotheses, the topmost value (reference) from the operand stack is replaced with the length of and the array corresponding to that reference. If the topmost value of the operand stack is `null`, *arraylength* α throws a `NullPointerException`.

arrayload α is the semantics of the bytecode instruction arrayload α. It requires that the topmost value from the operand stack is an integer k, the second topmost value is a reference to an array whose array type is a subtype of α, and which length is greater than k. Under these hypotheses, the two topmost values are popped from the operand stack and the value of the k-th element of the array is pushed back onto the operand stack. If the topmost element of the operand stack is greater or equal to the array length, *arrayload* α throws a `ArrayIndexOutOfBoundsException`. If the second topmost value of the operand stack is `null`, *arrayload* α throws a `NullPointerException`.

arraystore α is the semantics of the bytecode instruction arraystore α. It requires that the topmost value from the operand stack is consistent with a subtype of t, that the second topmost value from the operand stack is an integr k and that the third topmost value from the operand stack is a reference to an array whose array type is a subtype of α = t[ ] and whose length is greater than k. Under these hypotheses, the two topmost values are popped from the operand stack and the first one is written in the k-th element of the array corresponding to the second one. If the topmost value of the operand stack is not consistent with t, *arraystore* α throws a `ArrayStoreException`. If the second tompost value from the operand stack is greater or equal to the array length, *arraystore* α throws a `ArrayIndexOutOfBoundsException`. If the third topmost value of the operand stack is `null`, *arraystore* α throws a `NullPointerException`.

dup t is the semantics of the bytecode instruction dup t. It duplicates the topmost value on the operand stack and push the duplicated value onto the operand stack.

ifeq t is the semantics of the bytecode instruction ifeq t. It checks whether the topmost value from the operand stack whose type is t, is 0 when t = `int` or is `null` when t ≠ `int`. In our formalization, conditional bytecode instructions are used in complementary pairs (ifne t and ifeq t), at the beginning of the two conditional branches. The semantics of a conditional bytecode instruction is undefined when its condition is false, i.e., in that case the Java virtual machine does not continue the execution of the code.

ifne t - complement of *ifeq* t.

return void is the semantics of the bytecode instruction return void. It terminates a void method and clears the operand stack of the frame corresponding to the method.

return t is the semantics of the bytecode instruction return t. It terminates a method returning a value of type t, clears the operand stack of the frame corresponding to the method and pushes onto the operand stack the returned value when $t \neq \mathsf{void}$.

throw κ is the semantics of the bytecoe instruction throw κ. It requires that the topmost value from the operand stack is a reference to an object whose class is a subclass of $\kappa \leq$ `Throwable`. Then it throws the object pointed by the topmost value from the stack and pops that value from the stack. If the former is null, *throw* κ throws a `NullPointerException`.

catch is the semantics of the bytecode instruction catch. It starts an exception handler. It takes an exceptional state and transforms it into a normal one, subsequently used by the handler.

exception_is K is the semantics of the bytecode instruction exception_is K. The latter is used after a catch, and it selects an appropriate handler depending on the run-time class of the topmost value from the operand stack: it filters those states whose topmost value of the operand stack is an instance of a class in $K \subseteq \mathbb{K}$.

3.4.3 Method calls

When a caller transfers control to a callee $\kappa.m(\vec{\mathsf{t}}) : \mathsf{t}$, the Java virtual machine runs an operation *makescope* $\kappa.m(\vec{\mathsf{t}})\colon \mathsf{t}$ that copies the topmost stack elements, holding the actual arguments of the call, to the local variables of the frame corresponding to the callee, which correspond to the formal parameters of the callee, and clears the stack. We only consider instance methods, where this is a special argument held in local variable l_0 of the callee.

Definition 3.17 (makescope). *Let* $\kappa.m(\vec{\mathsf{t}})\colon \mathsf{t}$ *be a method and* π *the number of stack elements holding its actual parameters, including the implicit parameter* this*. We define a function* (*makescope* $\kappa.m(\vec{\mathsf{t}})\colon \mathsf{t}) : \Sigma \to \Sigma$ *as*

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

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

More precisely, the i-th local variable of the callee's frame is a copy of the $(\pi-1)-i$-th topmost element from the operand stack of the caller's frame.

3.4.4 The Transition Rules

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

Definition 3.18 (Configuration). *A* configuration *is a pair* $\langle b \parallel \sigma\rangle$ *of a block* b *and a state* σ *representing the fact that the Java virtual machine is about to execute* b *in state* σ*. An* activation stack *is a stack* $c_1 :: c_2 :: \cdots :: c_n$ *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 instruction.

$$\frac{\mathsf{ins} \text{ is not a } \texttt{call},\ \mathit{ins}(\sigma) \text{ is defined}}{\langle \boxed{\begin{array}{l}\mathsf{ins}\\ \mathit{rest}\end{array}} \to \begin{array}{l} b_1\\ \vdots\\ b_m\end{array} \ \|\, \sigma\rangle :: a \Rightarrow \langle \boxed{\mathit{rest}} \to \begin{array}{l} b_1\\ \vdots\\ b_m\end{array} \ \|\, \mathit{ins}(\sigma)\rangle :: a} \quad (1)$$

$$\frac{\begin{array}{c}\pi \text{ is the number of parameters of the target method, including } \texttt{this}\\ \sigma = \langle \mathsf{l} \parallel v_{\pi-1} :: \cdots :: v_1 :: \mathit{rec} :: \mathsf{s}\rangle, \mu\rangle,\ \mathit{rec} \neq \texttt{null}\\ 1 \leq i \leq n,\ \sigma' = (\mathsf{makescope}\ m_i)(\sigma) \text{ is defined}\\ f = \mathit{first}(m_i), \text{ the block where the implementation starts}\end{array}}{\langle \boxed{\begin{array}{c}\texttt{call}\ m_1 \ldots m_n\\ \mathit{rest}\end{array}} \to \begin{array}{l} b_1\\ \vdots\\ b_m\end{array} \|\, \sigma\rangle :: a \Rightarrow \langle f \,\|\, \sigma'\rangle :: \langle \boxed{\mathit{rest}} \to \begin{array}{l} b_1\\ \vdots\\ b_m\end{array} \|\langle\langle \mathsf{l} \parallel \mathsf{s}\rangle, \mu\rangle\rangle :: a} \quad (2)$$

$$\frac{\begin{array}{c}\pi \text{ is the number of parameters of the target method, including } \texttt{this}\\ \sigma = \langle \mathsf{l} \parallel v_{\pi-1} :: \cdots :: v_1 :: \texttt{null} :: \mathsf{s}\rangle, \mu\rangle\\ \ell \in \mathbb{L} \text{ is fresh and } \mathit{npe} \text{ is a new instance of } \texttt{NullPointerException}\end{array}}{\langle \boxed{\begin{array}{c}\texttt{call}\ m_1 \ldots m_n\\ \mathit{rest}\end{array}} \to \begin{array}{l} b_1\\ \vdots\\ b_m\end{array} \|\, \sigma\rangle :: a \Rightarrow \langle \boxed{\mathit{rest}} \to \begin{array}{l} b_1\\ \vdots\\ b_m\end{array} \|\, \underline{\langle\langle \mathsf{l} \parallel \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle}\rangle :: a} \quad (3)$$

$$\frac{|\mathsf{s}| \leq 1}{\langle \boxed{\ } \|\langle\langle \mathsf{l} \parallel \mathsf{s}\rangle, \mu\rangle\rangle :: \langle b \|\langle\langle \mathsf{l}' \parallel \mathsf{s}'\rangle, \mu'\rangle\rangle :: a \Rightarrow \langle\, b \|\langle\langle \mathsf{l}' \parallel \mathsf{s} :: \mathsf{s}'\rangle, \mu\rangle\rangle :: a} \quad (4)$$

$$\frac{}{\langle \boxed{\ } \|\, \underline{\langle\langle \mathsf{l} \parallel e\rangle, \mu\rangle}\rangle :: \langle b \|\langle\langle \mathsf{l}' \parallel \mathsf{s}'\rangle, \mu'\rangle\rangle :: a \Rightarrow \langle\, b \,\|\, \underline{\langle\langle \mathsf{l}' \parallel e\rangle, \mu\rangle}\rangle :: a} \quad (5)$$

$$\frac{1 \leq i \leq m}{\langle \boxed{\ } \to \begin{array}{l} b_1\\ \vdots\\ b_m\end{array} \ \|\, \sigma\rangle :: a \Rightarrow \langle b_i \,\|\, \sigma\rangle :: a} \quad (6)$$

Fig. 3.7. The transition rules of our semantics

Definition 3.19 (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 by the rules in Fig. 3.7.*

Rule (1) runs the first instruction ins of a block, different from call, by using its semantics *ins*. Then it moves forward to run the remaining instructions.

Rules (2) and (3) are for method calls. If a call occurs on a `null` receiver, no actual call happens in this case and rule (3) creates a new state whose operand stack contains only a reference to a `NullPointerException`. On the other hand, 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 [67]. 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 might lead to many implementations, depending on the run-time class of

the receiver. Although this rule seems non-deterministic, only one thread of execution continues, since we assume that the look-up rules are deterministic, as in Java bytecode.

Control returns to the caller by rules (4) and (5). If the callee ends in a normal state, rule (4) rehabilitates the caller configuration but keeps the memory at the end of the execution of the callee and, if $\mathsf{s} \neq \epsilon$, it also pushes the return value on the operand stack of the caller. 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.

In the notation $\Rightarrow$, we often specify the rule in Fig. 3.7 used; for instance, we write $\overset{(1)}{\Rightarrow}$ for a derivation step through rule (1).

4

General Parameterized Framework for Constraint-based Static Analyses of Java Bytecode Programs

This chapter introduces a *generic parameterized static analysis framework for Java bytecode*, based on constraint generation and solving. This framework is able to deal with the exceptional flows inside the program and the side-effects induced by calls to non-pure methods. It is generic in the sense that different instantiations of its parameters give rise to different static analyses which might capture complex properties of the heap at each program point. This framework can be used to generate *possible* or *may* approximations of the property of interest, as well as *definite* or *must* approximation of that property. In order to exemplify these two types of analyses, the following chapters provide two instantiations of this framework: *Possible Reachability Analysis Between Program Variables* (Chapter 5) and *Definite Expression Aliasing Analysis* (Chapter 6). This chapter is based on [59].

4.1 Contribution and Organization of the Chapter

This chapter introduces a *general parameterized framework for constraint-based static analyses of Java bytecode programs*, which is abstract interpretation-based and can be used to formalize static analyses dealing with both numerical and memory-related properties. Moreover, framework's structure allows one to define static analyses which deal with both side-effects of the methods, and with their exceptional executions. Finally, it simplifies proofs of correctness of the static analyses formalized in the framework.

The crucial notion for the operational semantics of the target language introduced in Chapter 3 is the notion of *state*, representing a system configuration. The set of all possible states that might be related to a given program point is called the *concrete domain*, and it is denoted by C. Let P be a program under analysis, composed of a set of `.class` files, and let L be the set of libraries that P uses. Suppose that P's classes as well as libraries in L are archived in a `.jar` file, representing one of the inputs the present framework requires. Moreover, let $\mathcal{P}$ be a generic property of interest. The actions performed by the framework are listed below.

1. The framework extracts, from the `.jar` archive, an extended control flow graph (eCFG), which contains a node for each bytecode instruction available in P and L, some special nodes which deal with the side-effects of non-pure methods, as well as

with exceptional and non-exceptional method ends, and different types of arcs which connect those nodes. There are some simple arcs connecting one source node with one sink node, but there are also some special arcs, composed of two source nodes and one sink node: their main purpose is handling of the method's side effects in both exceptional and non-exceptional executions of those methods and they represent one of the actual contributions of the present thesis. Section 4.2 explains different parts of these graphs. It is worth noting that this step does not depend on any particular property of interest, and is always done automatically by the framework.

2. This framework is abstract interpretation-based, i.e., the property of interest $\mathcal{P}$ is mathematically encoded in terms of an abstract domain. This chapter considers a generic abstract domain A and introduces some requirements dealing with C and A in Section 4.3. This step is property-dependent and the designer of a new static analysis is supposed to give a concrete instantiation of the generic abstract domain A.
3. For each arc present in the eCFG, the framework requires a propagation rule Π representing the behavior of the bytecode instruction corresponding to the source node of the arc with respect to the abstract domain A, and therefore with respect to the property $\mathcal{P}$. This chapter considers generic propagation rules, while their concrete instantiation is property-dependent and is provided by the designer of a new static analysis. The framework annotates each eCFG's arc with the corresponding propagation rule, obtaining the *abstract constraints graph* (ACG). Requirements concerning the propagation rules are introduced in Section 4.4.
4. From the annotated graph the framework extracts a *system of constraints* that represents the actual definition of a new constraint-based static analysis; its solution is the approximation provided by that static analysis. This step is both property-independent and property-dependent, namely, the construction of the system of constraints does not depend on the property of interest, while the propagation rules are specific to the property of interest. The extraction of constraints from the ACG is explained in Section 4.5.

In order to define sound static analyses, the framework introduces, in Section 4.4, a set of *requirements that the framework's parameters, i.e., the abstract domain and the propagation rules, must satisfy*. The results provided in Section 4.6 guarantee that when an instantiation of the framework's parameters satisfies the requirements mentioned above, the static analysis determined by this instantiation is sound. This is a very important result, since it allows one to show that a static analysis of a huge, real-life program, written in Java bytecode, is sound. In order to show that, it is not necessary to consider the program's structure or the libraries it uses. Another important result of this chapter is the fact that the satisfiability of the requirements implies that there always exists a solution of the system of constraints obtained from the ACG.

When the designer of a new static analysis wants to formalize a sound static analysis using the present framework, he or she has to complete the following tasks:

1. *formally define the property of interest* $\mathcal{P}$ and express it as an *abstract domain* A in terms of the abstract interpretation framework [32]. Examples of this instantiation are given in Sections 5.3.1 and 6.3.1.
2. *define a propagation rule* $\Pi : \mathsf{A} \rightarrow \mathsf{A}$, for each bytecode instruction of the target language, and each possible behavior of that instruction (exceptional and non-exceptional). Examples of these instantiations are given in Sections 5.3.2 and 6.3.2.

3. show that A and the propagation rules satisfy all of the requirements specified by the framework, which is not a simple task, but it is drastically easier to show that, for example, each propagation rule soundly mimics the behavior of the corresponding bytecode instruction, than to show that the abstract semantics of whole program soundly approximates the operational semantics of that program. Sections 5.4 and 6.4 show that the static analyses introduced respectively in Chapters 5 and 6 actually satisfy the requirements introduced in this chapter.

Fig. 1.3 lists the sequence of actions automatically done by the framework (on the right) and required by the user (on the left). The actions are performed by following the order specified in the figure, i.e., first of all the framework extracts the eCFG, then the designer instantiate the abstract domain and the propagation rules, then the framework constructs the ACG, etc.

There is also another important benefit of using this framework. From an implementational point of view, Julia contains an abstract class (in terms of Java) implementing the generic engine for eCFG and ACG creation, constraint generation and solving. This implementation represents an implementation of the framework and is not a contribution of the present thesis. Each new specific static analysis is a concrete subclass of the abstract class mentioned above, providing an implementation for a few methods, where the specific static analyses deviates from the general framework. This largely simplifies the implementation of new static analyses. For instance, the developer needs not bother about the implementation of the constraints and the strategy for their solution.

4.2 Construction of the Extended Control Flow Graph

This section introduces the notion of *extended control flow graph* (eCFG), i.e., a control flow graph extracted from a `.jar` archive composed of all the classes belonging to the program under analysis, as well as of all the classes from the auxiliary libraries that program uses. Similarly to the traditional control flow graph (Chapter 3), eCFG is composed of a set of nodes corresponding to different bytecode instructions belonging to the program, and a set of arcs which connect those nodes. Differently from the traditional control flow graph, eCFG contains some special nodes and special arcs which are not present in the former. This section formally introduces both traditional and special nodes and arcs.

Definition 4.1 (eCFG). *Let P be the program under analysis enriched with all of the methods from the libraries it uses, already in the form of a CFG of basic blocks for each method or constructor (Chapter 3). The* extended control flow graph *(eCFG) for P is a directed graph $\langle V, E \rangle$ (nodes, arcs) where:*

1. *V contains a node* $\boxed{\text{ins}}$ *for each bytecode instruction* ins *in P;*
2. *for each method or constructor m in P, V contains nodes* $\boxed{\text{exit@}m}$ *and* $\boxed{\text{exception@}m}$*, representing the normal and the exceptional ends of m;*
3. *each node contains an abstract element representing an approximation of the information related to the property of interest at that point;*
4. *E contains directed arcs with one ($1-1$) or two ($2-1$) sources and always one sink. Each arc has a* propagation rule *i.e., a function over the abstract domain, from the approximation(s) contained in its source(s) to the one contained in its sink. We distinguish the following types of arcs:*

- ***Sequential arcs:*** *if* ins *is a bytecode instruction in* P*, distinct from* call*, immediately followed by a bytecode instruction* ins′*, distinct from* catch*, then a* $1-1$ *sequential arc is built from* $\boxed{\text{ins}}$ *to* $\boxed{\text{ins}'}$*;*
- ***Final arcs:*** *for each* return t *and* throw κ *instructions occurring in a method or a constructor* m *of* P*, there are* $1-1$ *final arcs from* $\boxed{\text{return t}}$ *to* $\boxed{\text{exit@}m}$ *and from* $\boxed{\text{throw }\kappa}$ *to* $\boxed{\text{exception@}m}$*, respectively;*
- ***Exceptional arcs:*** *for each bytecode instruction* ins *throwing an exception, immediately followed by a* catch*, an* $1-1$ *exceptional arc is built from* $\boxed{\text{ins}}$ *to* $\boxed{\text{catch}}$*;*
- ***Parameter passing arcs:*** *for each* call $m_1 \dots m_q$ *occurring in* P *with* π *parameters (including the implicit parameter* `this`*) we build, for each* $1 \leq w \leq q$*, a* $1-1$ *parameter passing arc from* $\boxed{\text{call } m_1 \dots m_q}$ *to the node corresponding to the first bytecode instruction of the method* m_w*;*
- ***Return value arcs:*** *for each call* ins_C = call $m_1 \dots m_q$ *to a method with* π *parameters (including the implicit parameter* `this`*) returning a value of type* t ≠ void*, and each subsequent bytecode instruction* ins_N *distinct from* catch*, we build, for each* $1 \leq w \leq q$*, a* $2-1$ *return value arc from* $\boxed{\text{ins}_C}$ *and* $\boxed{\text{exit@}m_w}$ *(2 sources, in that order) to* $\boxed{\text{ins}_N}$*;*
- ***Side-effects arcs:*** *for each call* ins_C = call $m_1 \dots m_q$ *to a method with* π *parameters (including the implicit parameter* `this`*), and each subsequent bytecode instruction* ins_N*, we build, for each* $1 \leq w \leq q$*, a* $2-1$ *side-effects arc from* $\boxed{\text{ins}_C}$ *and* $\boxed{\text{exit@}m_w}$ *(2 sources, in that order) to* $\boxed{\text{ins}_N}$*, if* ins_N *is not a* catch *and a* $2-1$ *side-effect arc from* $\boxed{\text{ins}_C}$ *and* $\boxed{\text{exception@}m_w}$ *(2 sources, in that order) to* $\boxed{\text{catch}}$.

The sequential arcs correspond to the non-exceptional executions of all the bytecode instructions except call, return and throw. The final arcs connect the nodes corresponding to the last bytecode instruction of each method or constructor m (i.e., return or throw) to the special nodes $\boxed{\text{exit@}m}$, in the case of return, and $\boxed{\text{exception@}m}$, in the case of throw. The exceptional arcs represent the exceptional executions of the bytecode instructions that might launch an exception, i.e., div, rem, new κ, getfield f, putfield f, arraynew α, arraylength α, arrayload α, arraystore α, throw and call, and they connect the nodes

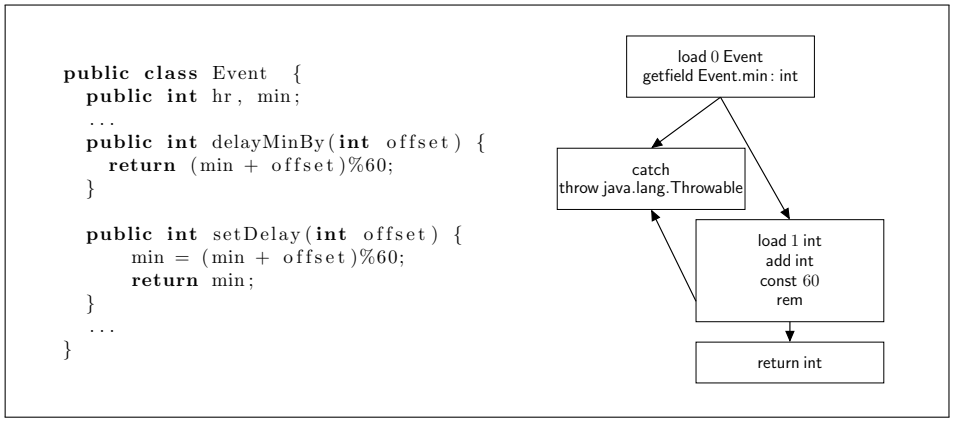


Fig. 4.1. Part of a Java class `Event` implementing a method `delayMinBy` on the left, and the traditional CFG of that method on the right

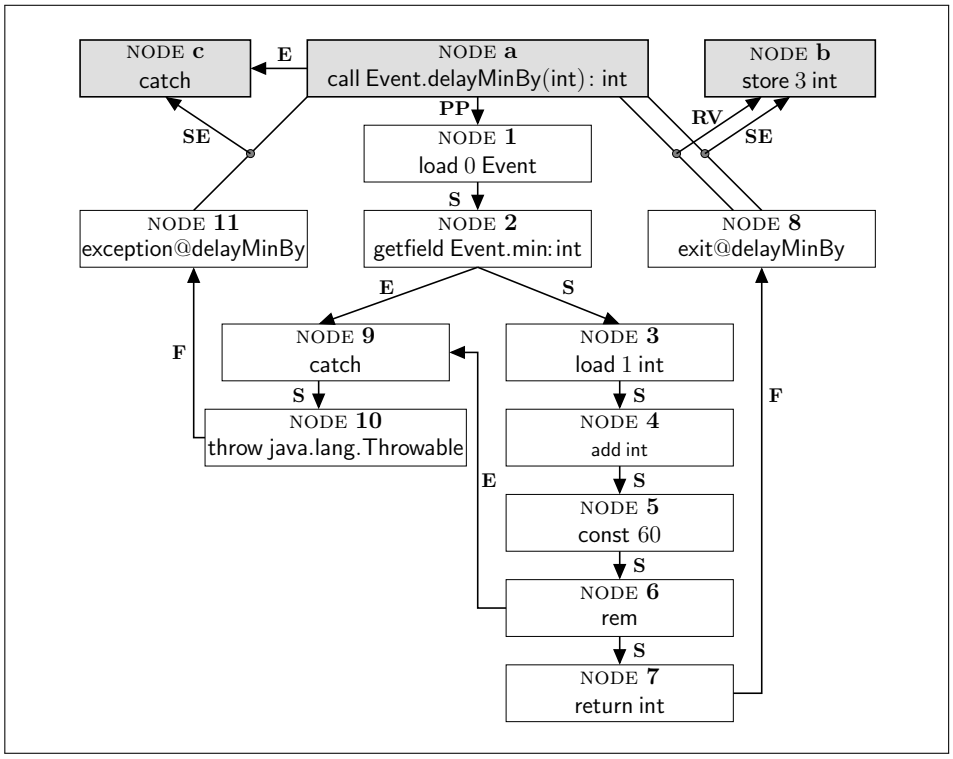


Fig. 4.2. The eCFG for the method `delayMinBy` in Fig. 4.1

corresponding to these instructions with the node related to the catch instruction at the beginning of their exceptional handlers. The parameter passing arcs link every node corresponding to a method call to the node corresponding to the first bytecode instruction of the method(s) that might be called there. There exists a return value arc for each dynamic target m of a call ins_C returning a value. These arcs have two sources, $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit@}m_w}$, and they propagate the approximations present at these nodes to the node corresponding to the bytecode instruction following ins_C. Moreover, these arcs might enrich the resulting approximation with some additional abstract elements due to the m's returned value. The execution of method m might modify the memory where m is executed and this might affect the approximation at node $\boxed{\mathsf{ins}_C}$ corresponding to the method call ins_C. The side-effects arcs deal with these phenomena, i.e., they are $\mathsf{2-1}$ arcs connecting $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit@}m}$ (respectively $\boxed{\mathsf{exception@}m}$) with the node corresponding to the bytecode instruction (respectively catch) which follows ins_C, for each dynamic target m of the call, and propagate the approximation at $\boxed{\mathsf{ins}_C}$ modified by the side-effects of m's execution.

Example 4.2. We introduce a class `Event` containing two instance fields `hr` and `min` of type `int` and two instance methods `delayMinBy` and `setDelay` taking one `int` argument and returning an `int` value. Its Java code, as well as the rapresentation of the method `delayMinBy` in our formalization are given in Fig. 4.1. In Fig. 4.2 we give the eCFG of the method `delayMinBy`. Nodes **a**, **b** and **c** belong to the caller of this method and exemplify the arcs related to the call and return bytecodes. Arcs are decorated with an

abbreviation denoting their types: **S**, **F**, **E**, **PP**, **RV** and **SE** state for sequential, final, exceptional, parameter passing, return value and side-effects arcs, respectively. □

4.3 Concrete and Abstract Domains

This framework is based on a general theory of approximation, called abstract interpretation [32, 34], whose idea is the following: suppose there exists a set of elements, called *concrete domain*, and a function f operating on them. Abstract interpretation defines an *abstract domain*, i.e., a set of elements containing less information than the concrete ones, two maps relating the domains and a function operating on the abstract domain that simulates f.

All the analyses formalized in the present framework use the same concrete domain which is composed of all possible states that can be verified at a program point. On the other hand, the abstract domain depends on the property of interest, and its elements are obtained by abstracting away from the concrete states all those pieces of information that are irrelevant for the analysis of interest. For every analysis, it is necessary to formally define the form of abstract elements and an ordering among them. The former corresponds to the property that the analysis deals with, while the latter is relevant for the existence of the least solution of that analysis. Moreover, concrete and abstract elements have to be connected by a pair of functions that specify which concrete elements correspond to which abstract elements and vice versa, i.e., they must form a Galois connection. This section provides some conditions that the abstract domain has to satisfy in order to guarantee the correctness of the static analysis and the existence of its solutions.

Definition 4.3 (Concrete and Abstract Domain). *The concrete and abstract domains over $\tau \in \mathcal{T}$ are $\mathsf{C}_\tau = \langle \wp(\Sigma_\tau), \subseteq, \cup, \cap, \Sigma_\tau, \varnothing \rangle$ and $\mathsf{A}_\tau = \langle \mathcal{A}_\tau, \sqsubseteq, \sqcup, \sqcap, \top_\tau, \bot_\tau \rangle$, where $\mathcal{A}_\tau$ is the set of elements that represent the property of interest over τ, $\sqcup$ and $\sqcap$ are the join and meet operators, and $\top_\tau$ and $\bot_\tau$ are the top and the bottom elements of $\mathcal{A}_\tau$.*

Sections 5.3.1 and 6.3.1 show different instantiations of $\mathcal{A}_\tau$ and A_τ.

Recall some well-known notions from lattice theory. A sequence $\{\vec{\mathsf{A}_i}\}_{i \in \mathbb{N}}$ of elements in $\mathcal{A}_\tau$ is an *ascending chain* if $n \leq m \Rightarrow \mathsf{A}_n \sqsubseteq \mathsf{A}_m$. That sequence *eventually stabilizes* iff $\exists n_0 \in \mathbb{N}. \forall n \in \mathbb{N}. n \geq n_0 \Rightarrow \mathsf{A}_n = \mathsf{A}_{n_0}$. The first requirement related to abstract domain is one of the conditions which guarantee the existence and the uniqueness of the least solution of the analyses, like Theorem 4.11 shows.

Requirement 4.1 (ACC) *Every ascending chain of elements in $\mathcal{A}_\tau$ eventually stabilizes.*

Abstract and concrete states are connected by a pair of maps $\alpha_\tau : \mathsf{C}_\tau \rightarrow \mathsf{A}_\tau$ and $\gamma_\tau : \mathsf{A}_\tau \rightarrow \mathsf{C}_\tau$ called *abstraction* and *concretization* maps. The former abstracts away some irrelevant pieces of information contained in the concrete elements, while the latter explains the actual meaning of the abstract states. Namely, it associates every abstract state A with the concrete states preserving the approximation represented by A. In order to have an abstract interpretation-based static analysis, it is necessary to provide these two maps and to show that they are related by a Galois connection (Chapter 2). This is specified by Requirement 4.2.

Requirement 4.2 (GC) *It is necessary that $\langle \mathsf{C}_\tau, \alpha_\tau, \gamma_\tau, \mathsf{A}_\tau \rangle$ is a Galois connection, i.e.,*

$$\forall A \in \mathsf{A}_\tau. \forall C \in \mathsf{C}_\tau. \alpha_\tau(C) \sqsubseteq A \Leftrightarrow C \subseteq \gamma_\tau(A).$$

4.4 Propagation Rules and the Abstract Constraint Graph

The goal of the constraint-based static analyses defined in this thesis is to associate each node of the eCFG constructed in Section 4.2 with an element of the abstract domain defined in Section 4.3. The latter represents an approximation of the property of interest at that point. In order to do that, the framework annotate each node of the eCFG with a variable, and each arc of the eCFG with a function specifying how the approximations at its source(s) is(are) transformed in the approximation of its sink. These functions are called *propagation rules*, their formal definition depends on a concrete property of interest and it represents the core of the actual static analysis. The eCFG whose arcs are annotated with these propagation rules is called *abstract constraint graph* (ACG).

The goal of this chapter is to define a general framework for constraint-based static analyses of Java bytecode programs. Since the former deals with generic properties, this section does not introduce one particular instantiation of the propagation rules. It rather specifies a set of *requirements* that the propagation rules must satisfy in order to have a sound and computable analysis. On the other hand, Sections 5.3.2 and 6.3.2 introduce the instantiations of the propagation rules used for the formalization of two novel constraint-based static analyses introduced respectively in Chapters 5 and 6.

The first condition concerning the propagation rules that the present framework requires is that all the propagation rules, representing an abstract semantics of bytecode instructions, are monotonic, like it is the case with the concrete semantics of those instructions (Requirement 4.3). This condition allows to show that there exists the least solution of our constraint-based analysis (see Theorem 4.11).

Requirement 4.3 (Monotonicity) *Propagation rules of the* ACG *related to the program under analysis are monotonic w.r.t.* $\sqsubseteq$.

It is also required that the propagation rules correctly approximate the bytecode instructions they simulate, i.e., in abstract interpretation terms, an abstract element correctly approximates the property of interest at a program point, if the concretization of the former contains all the concrete states that the program might be in at that point during any execution. In particular, for each sequential arcs, it is required that its propagation rule propagates only non-exceptional concrete states from the concretization of a correct approximation of the property of interest related to the source node of that sequential arc (Requirement 4.4). This is required because sequential arcs link a bytecode instruction with their successors when no exception is thrown, hence their abstract semantics must be consistent with that situation. On the other hand, in the case of the propagation rules of the exceptional arcs, the framework requires the correct propagation of the exceptional concrete states only, since these propagation rules simulate the exceptional behaviors of different bytecode instructions (Requirement 4.6). In the case of the final and the parameter passing arcs, both exceptional and non-exceptional concrete states have to be correctly propagated (Requirements 4.5 and 4.8). Finally, Requirement 4.7 deals with one particular case of the exceptional arcs: when a method is invoked on a `null` receiver. In that case it is required that the exceptional states launched by the method are included in the approximation of the property of interest after the call to that method.

Requirement 4.4 (Sequential arcs) *Consider a sequential arc from the node corresponding to a bytecode instruction* ins *and its propagation rule* Π. *Assume that* ins *has*

static type information τ at its beginning and τ' immediately after its non-exceptional execution. Then, for every $A \in \mathsf{A}_\tau$, we require

$$ins(\gamma_\tau(A)) \cap \Xi_{\tau'} \subseteq \gamma_{\tau'}(\Pi(A)). \tag{4.1}$$

Requirement 4.5 (Final arcs) *Consider a final arc from the node corresponding to a bytecode instructions* ins *and its propagation rule Π. 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*, its exceptional execution if* ins *is a* throw κ*). Then, for every $A \in \mathsf{A}_\tau$, we require*

$$ins(\gamma_\tau(A)) \subseteq \gamma_{\tau'}(\Pi(A)). \tag{4.2}$$

Requirement 4.6 (Exceptional arcs) *Consider an exceptional arc from the node corresponding to a bytecode instruction* ins *distinct from* call *and its propagation rule Π. Assume that* ins *has static type information τ at its beginning and τ' after its exceptional execution. Then, for every $A \in \mathsf{A}_\tau$, we require*

$$ins(\gamma_\tau(A)) \cap \underline{\Xi}_{\tau'} \subseteq \gamma_{\tau'}(\Pi(A)). \tag{4.3}$$

Requirement 4.7 *Consider an exceptional arc from the node corresponding to* $\mathsf{ins}_C =$ call $m_1 \dots m_q$ *and its propagation rule Π. Assume that it has π actual arguments (*this *included), and that τ and τ' are respectively the static type information before and immediately after* ins_C*. Then, for each $1 \leq w \leq q$, every $A \in \mathsf{A}_\tau$ and every $\sigma = \langle\langle \mathsf{l} \parallel v_{\pi-1} :: \ldots :: v_1 :: \mathtt{null} : \mathsf{s}\rangle, \mu\rangle \in \gamma_\tau(A)$, i.e., the ones assigning* `null` *to the receiver of* ins_C *right before its execution, we require*

$$\underline{\langle\langle \mathsf{l} \parallel \ell\rangle, \mu[\ell \mapsto npe]\rangle} \subseteq \gamma_{\tau'}(\Pi(A)), \tag{4.4}$$

where ℓ is a fresh location, and npe is a new instance of `NullPointerException`*.*

Requirement 4.8 (Parameter passing arcs) *Let us consider a parameter passing arc from the node corresponding to* call $m_1 \dots m_q$ *to the first bytecode of m_w, for some $1 \leq w \leq q$, and its propagation rule Π. Assume that* call $m_1 \dots m_q$ *has static type information τ at its beginning and that τ' is the static type information at the beginning of m_w. Then, for each $1 \leq w \leq q$ and every $A \in \mathsf{A}_\tau$, we require*

$$(makescope\ m_w)(\gamma_\tau(A)) \subseteq \gamma_{\tau'}(\Pi(A)). \tag{4.5}$$

The following requirement deals with the return values and side-effects of the non-exceptional executions of methods. Namely, it is required that, in the case a method returns a value, the propagation rule of the return value arc enriches the resulting approximation of the property of interest immediately after the call to that method by adding all those abstract elements that the return value might correspond to. On the other hand, that method might modify the initial memory from which the method has been executed. These modifications must be captured by the propagation rules of the side-effects arcs. The approximation of the property of interest after the call to a method is, therefore, determined as the join ($\sqcup$) of the approximations obtained from the propagation rules of the return value and the side-effects arcs, and we require it to be correct. Requirement 4.10 handles the case of a void method, and therefore only the corresponding side-effects arc is considered there.

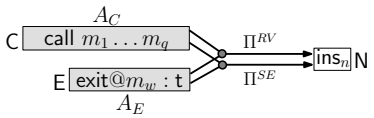


Fig. 4.3. Arcs going into the node corresponding to ins_N.

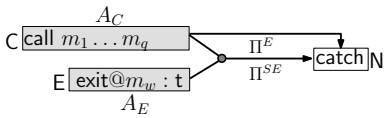


Fig. 4.4. Arcs going into the node corresponding to catch.

Requirement 4.9 *Let* $C = \boxed{\mathsf{ins}_C}$ *be the node corresponding to the call to a* non-void *method* $\mathsf{ins}_C = \mathsf{call}\ m_1 \dots m_q$, $E = \boxed{\mathsf{exit@}m_w}$, *be the exit node from a method* m_w, *where* $1 \leq w \leq q$, *and* $N = \boxed{\mathsf{ins}_N}$ *be the node corresponding to the only bytecode instruction different from* catch *which follows* ins_C. *Consider a return value and a side-effect arc from nodes* C *and* E *to* N, *and let* Π^{RV} *and* Π^{SE} *be the propagation rules of these arcs. We depict this situation in Fig. 4.3. Let* τ_C, τ_E *and* τ_N *be the static type information at* C, E *and* N *respectively, and let* d *be 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 each* $1 \leq w \leq q$, *every* $A_C \in \mathsf{A}_{\tau_C}$ *and every* $A_E \in \mathsf{A}_{\tau_E}$, *we require*

$$d((\mathit{makescope}\ m_w)(\gamma_{\tau_C}(A_C))) \cap \Xi_{\tau_N} \subseteq \gamma_{\tau_N}(\Pi^{RV}(A_C, A_E) \sqcup \Pi^{SE}(A_C, A_E)). \quad (4.6)$$

Requirement 4.10 *Let* $C = \boxed{\mathsf{ins}_C}$ *be the node corresponding to the call to a* void *method* $\mathsf{ins}_C = \mathsf{call}\ m_1 \dots m_q$, $E = \boxed{\mathsf{exit@}m_w}$, *be the exit node from a method* m_w, *where* $1 \leq w \leq q$, *and* $N = \boxed{\mathsf{ins}_N}$ *be the node corresponding to the only bytecode instruction different from* catch *which follows* ins_C. *Consider a side-effect arc from nodes* C *and* E *to* N, *and let* Π^{SE} *be the propagation rule of this arc. This situation is depicted in Fig. 4.3 where the arc annotated with* Π^{RV} *should not be considered. Let* τ_C, τ_E *and* τ_N *be the static type information at* C, E *and* N *respectively, and let* d *be 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 each* $1 \leq w \leq q$, *every* $A_C \in \mathsf{A}_{\tau_C}$ *and every* $A_E \in \mathsf{A}_{\tau_E}$, *we require*

$$d((\mathit{makescope}\ m_w)(\gamma_{\tau_C}(A_C))) \cap \Xi_{\tau_N} \subseteq \gamma_{\tau_N}(\Pi^{SE}(A_C, A_E)). \quad (4.7)$$

The last requirement deals with the exceptional executions of the methods. Namely, the approximation of the property of interest at the catch which captures the exceptional states of the method we are interested in, has to be enriched by all possible modifications of the initial memory due to the side-effects of the method. This is the task of the propagation rules of the side-effects arcs. On the other hand, the final approximation of the property of interest at the point of interest (catch) has to be enriched with the exceptions launched by the method when it is invoked on a `null` object, like Requirement 4.7 already specified. Like in the previous case, the approximation of the property of interest is determined as the join ($\sqcup$) of the two approximations mentioned above, and we require it to be correct.

Requirement 4.11 *Given nodes* $N = \boxed{\mathsf{catch}}$, $C = \boxed{\mathsf{ins}_C}$ *for* $\mathsf{ins}_C = \mathsf{call}\ m_1 \dots m_q$ *and* $E = \boxed{\mathsf{exception@}m_w}$ *for some* $1 \leq w \leq q$, *consider an exceptional arc from* C *to* N *and a side-effect arc from* C *and* E *to* N, *with their propagation rules* Π^E *and* Π^{SE} *respectively. We depict this situation in Fig. 4.4. Let* τ_C, τ_N *and* τ_E *be the static type information at* C, N *and* E *respectively and let* d *be 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 each* $1 \leq w \leq q$, *every* $A_C \in \mathsf{A}_{\tau_C}$ *and every* $A_E \in \mathsf{A}_{\tau_E}$, *we have*

$$d((\textit{makescope}\ m_w)(\gamma_{\tau_C}(A_C))) \cap \underline{\Xi}_{\tau_N} \subseteq \gamma_{\tau_N}(\Pi^E(A_C) \sqcup \Pi^{SE}(A_C, A_E)). \tag{4.8}$$

4.5 Extraction and Solution of Constraints

The ACG of the program under analysis introduces, for each of its nodes a set of constraints: one constraint for each arc reaching the node. Every correct solution of these constraints is a possible, not necessarily minimal, result of the static analysis determined by that ACG. The following definition shows how the constraints are extracted from an ACG.

Definition 4.4 (Constraints). *Let N be a node of an* ACG *and A_N the approximation of the property of interest information contained in that node. Suppose that there are k arcs whose sink is N and for each $1 \leq i \leq k$, let Π^i and* $\mathsf{approx}(i)$ *respectively denote the propagation rule and the approximation of the property of interest at the source(s) of the $i-$th arc. These arcs give rise to the following constraints:*

$$\Pi^1(\mathsf{approx}(1)) \sqsubseteq A_N, \ldots, \Pi^k(\mathsf{approx}(k)) \sqsubseteq A_N.$$

In order to reduce the number of constraints, there exists the equivalent form:

$$\bigsqcup_{i=1}^{k} \Pi^i(\mathsf{approx}(i)) \sqsubseteq A_N. \tag{4.9}$$

Example 4.5. Fig. 4.5 shows the constraints extracted from the ACG obtained by annotating the eCFG introduced in Example 4.2. These constraints concern the method `delayMinBy` only, and not the whole program under analysis. □

$\Pi^{PP}(A_{\mathbf{a}}) \sqsubseteq A_{\mathbf{1}}$	$\Pi^S(A_{\mathbf{9}}) \sqsubseteq A_{\mathbf{10}}$
$\Pi^S(A_{\mathbf{1}}) \sqsubseteq A_{\mathbf{2}}$	$\Pi^F(A_{\mathbf{7}}) \sqsubseteq A_{\mathbf{8}}$
$\Pi^S(A_{\mathbf{2}}) \sqsubseteq A_{\mathbf{3}}$	$\Pi^F(A_{\mathbf{10}}) \sqsubseteq A_{\mathbf{11}}$
$\Pi^S(A_{\mathbf{3}}) \sqsubseteq A_{\mathbf{4}}$	$\Pi^E(A_{\mathbf{2}}) \sqsubseteq A_{\mathbf{9}}$
$\Pi^S(A_{\mathbf{4}}) \sqsubseteq A_{\mathbf{5}}$	$\Pi^E(A_{\mathbf{6}}) \sqsubseteq A_{\mathbf{9}}$
$\Pi^S(A_{\mathbf{5}}) \sqsubseteq A_{\mathbf{6}}$	$\Pi^{RV}(A_{\mathbf{a}}, A_{\mathbf{8}}) \sqcup \Pi^{SE}(A_{\mathbf{a}}, A_{\mathbf{8}}) \sqsubseteq A_{\mathbf{b}}$
$\Pi^S(A_{\mathbf{6}}) \sqsubseteq A_{\mathbf{7}}$	$\Pi^E(A_{\mathbf{a}}) \sqcup \Pi^{RV}(A_{\mathbf{a}}, A_{\mathbf{11}}) \sqsubseteq A_{\mathbf{c}}$

Fig. 4.5. The constraints extracted from the ACG given in Fig. 4.2

Once the abstract constraint graph for the program under analysis has been built, the framework extracts the constraints contained in the graph's arcs and finds the least solution w.r.t. $\sqsubseteq$ satisfying these constraints. This section, which is inspired by [58, Chapter 1.3], briefly discusses the existence and uniqueness of this solution.

Definition 4.6. *Suppose that there are x nodes in the* ACG *under analysis, and for each $1 \leq n \leq x$, let τ_n and A_n be the static type information and the approximation concerned with the $n-$th node. Let $\mathsf{EA} = (\mathsf{A}_{\tau_1} \times \ldots \times \mathsf{A}_{\tau_x})$ denote a set of tuples whose $n-$th element*

represents the approximation contained in the $n-$th node. Let $\vec{EA} = \langle A_1, \ldots, A_x \rangle$ and $\vec{EA}' = \langle A'_1, \ldots, A'_x \rangle$ be two arbitrary elements of EA*. Consider the following ordering:*

$$\vec{EA} \vec{\sqsubseteq} \vec{EA}' \quad \textit{iff} \quad \forall 1 \leq n \leq x. A_n \sqsubseteq A'_n.$$

This ordering gives rise to the following bottom ($\vec{\bot}$) and top ($\vec{\top}$) elements:

$$\vec{\bot} = \langle \bot_{\tau_1}, \ldots, \bot_{\tau_x} \rangle$$
$$\vec{\top} = \langle \top_{\tau_1}, \ldots, \top_{\tau_x} \rangle$$

Moreover, the join $\vec{\sqcup}$ and the meet $\vec{\sqcap}$ operators over EA *are defined as:*

$$\vec{EA} \vec{\sqcup} \vec{EA}' = \langle A_1 \sqcup A'_1, \ldots, A_x \sqcup A'_x \rangle$$
$$\vec{EA} \vec{\sqcap} \vec{EA}' = \langle A'_1 \sqcap A'_1, \ldots, A_x \sqcap A'_x \rangle.$$

In the following some simple algebraic properties of EA are shown.

Lemma 4.7. *$\langle \mathsf{EA}, \vec{\sqsubseteq}, \vec{\sqcup}, \vec{\sqcap}, \vec{\top}, \vec{\bot} \rangle$ is a complete lattice.*

Proof. Follows directly from the fact that, for each n, A_{τ_n} is a complete lattice. ■

Another important property of EA is the fact that it satisfies the Ascending Chain Condition.

Lemma 4.8. *If Requirement 4.1 is satisfied, every ascending chain of elements in* EA *eventually stabilizes.*

Proof. If Requirement 4.1 is satisfied, every ascending chain of elements in EA represents a Cartesian product of x ascending chains of elements in $\mathsf{A}_{\tau_1}, \ldots, \mathsf{A}_{\tau_x}$ which eventually stabilize, and x is a finite number. ■

Definition 4.9. *Consider a function F operating over* EA*:*

$$F : \mathsf{EA} \to \mathsf{EA}$$
$$F(\vec{EA}) = \langle F_1(\vec{EA}), \ldots, F_x(\vec{EA}) \rangle,$$

where $F_n : \mathsf{EA} \to \mathsf{A}_{\tau_n}$ represents the constraint associated to the $n-$th node (Equation 4.9).

Lemma 4.10. *F is a monotonic function w.r.t. $\vec{\sqsubseteq}$.*

Proof. By Requirement 4.3, the propagation rules are monotonic w.r.t. $\sqsubseteq$, and this entails the monotonicity of the constraints defined by Equation 4.9. Consequently, F is a monotonic function w.r.t. $\vec{\sqsubseteq}$. ■

Theorem 4.11. *The least solution of the equation system $F(\vec{EA}) = \vec{EA}$ exists and can be characterized as*

$$\mathsf{lfp}(F) = \vec{\bigsqcup}_n F^n(\vec{\bot}),$$

where given $\vec{EA} \in \mathsf{EA}$, the $i-$th power of F in $\vec{EA}$ is inductively defined as follows:

$$\begin{cases} F^0(\vec{EA}) & = \vec{EA} \\ F^{i+1}(\vec{EA}) & = F(F^i(\vec{EA})). \end{cases}$$

Proof. It is well-known that any monotonic function f over a partially ordered set satisfying the Ascending Chain Condition is also continuous [36]. By Lemma 4.10, F is monotone, and by Lemma 4.8, EA satisfies the Ascending Chain Condition, hence F is continuous.

On the other hand, by Knaster-Tarski's fixpoint theorem [88], in a complete lattice $\langle L, \leqslant, \vee, \wedge, \tilde{\top}, \tilde{\bot}\rangle$, for any continuous function $f : L \to L$, the least fixpoint of f, *lfp*(f), is equal to $\bigvee_n f^n(\tilde{\bot})$. Since $\langle \mathsf{EA}, \vec{\sqsubseteq}, \vec{\sqcup}, \vec{\sqcap}, \vec{\top}, \vec{\bot}\rangle$ is a complete lattice (Lemma 4.7) and F is continuous, its least fixpont can be computer as $\mathsf{lfp}(F) = \vec{\bigsqcup}_n F^n(\vec{\bot})$. ■

Corollary 4.12. *The equation system $F(\vec{EA}) \vec{\sqsubseteq} \vec{EA}$ and the constraint system $\vec{EA} = F(\vec{EA})$ have the same least solution.*

Proof. By Theorem 4.11, the least solution of $F(\vec{EA}) = \vec{EA}$ is constructed as $F^n(\vec{\bot})$ for a value $n \in \mathbb{N}$ such that $F^n(\vec{\bot}) = F^{n+1}(\vec{\bot})$. Suppose that $\vec{EA}$ is a solution of the constraint system, i.e., $F(\vec{EA}) = \vec{EA}$. Then, starting from $\vec{\bot} \vec{\sqsubseteq} \vec{EA}$, by the monotonicity of F and mathematical induction, it can be shown that $F^n(\vec{\bot}) \vec{\sqsubseteq} \vec{EA}$. Since $F^n(\vec{\bot})$ is a solution of the constraint system, this shows that it is also the least solution of the constraint system. ■

Finally, the solutions of the abstract constraint graph, i.e., of its corresponding static analysis can be characterized.

Definition 4.13 (Constraint-based Static Analysis). *The* solution *of an* ACG *is the least assignment of an abstract element $A_n \in \mathsf{A}_{\tau_n}$ to each node n of the* ACG*, $\vec{EA} = \langle A_1, \ldots, A_x\rangle \in \mathsf{EA}$, which satisfy the constraints extracted from the* ACG*, i.e., such that $F(\vec{EA}) \vec{\sqsubseteq} \vec{EA}$ holds.*

4.6 Soundness

This section shows that when the requirements provided in Section 4.4 are satisfied, the approximations computed by our static analyses are sound.

Theorem 4.14 (Soundness). *Let $\langle b_{first(\texttt{main})} \parallel \xi\rangle \Rightarrow^* \langle \boxed{\begin{array}{c}\mathsf{ins}_1\\ rest\end{array}}\!\!\begin{array}{l}\to b_1\\ \ldots\\ \to b_m\end{array} \parallel \sigma\rangle :: a$ be the execution of our operational semantics, from the block $b_{first(\texttt{main})}$ starting with the first bytecode instruction of method* `main`*, ins_0, and an initial state $\xi \in \Sigma_{\tau_0}$, to a bytecode instruction ins_1 and assume that this execution leads to a state $\sigma \in \Sigma_{\tau_1}$, where τ_0 and τ_1 are the static type information at ins_0 and ins_1 respectively. Moreover, let $A_0 \in \mathsf{A}_{\tau_0}$ be a correct approximation of the property of interest at ins_0, i.e., such that $\xi \in \gamma_{\tau_0}(A_0)$, and $A_1 \in \mathsf{A}_{\tau_1}$ be the approximation of the property of interest at ins_1 computed by our static analysis starting from A_0. Then, if Requirements 4.4-4.11 are satisfied, $\sigma \in \gamma_{\tau_1}(A)$ holds.*

Proof. The blocks in the configurations of an activation stack, but the topmost, cannot be empty and with no successor. This is because the configurations are only stacked by rule (2) of Fig. 3.7 and if *rest* is empty there, then $m \geq 1$ or otherwise, the code ends with a `call` bytecode with no `return`, which is illegal in Java bytecode [52].

We proceed by induction on the length n of the execution

$$\langle b_{first(\mathsf{main})} \parallel \xi\rangle \Rightarrow^* \langle \boxed{\begin{array}{c}\mathsf{ins}\\ rest\end{array}}\!\!\begin{array}{l}\to b_1\\ \ldots\\ \to b_m\end{array} \parallel \sigma\rangle :: a.$$

Base case: If $n = 0$, the execution is just $\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi \rangle$. In this case, $\tau_0 = \tau_1$ and $A_0 = A_1$, hence $\sigma = \xi \in \gamma_{\tau_0}(A_0) = \gamma_{\tau_1}(A_1)$.

Inductive step: Assume now that the thesis holds for any such execution of length $k \leq n$. Consider an execution $\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi \rangle \Rightarrow^{n+1} \langle \underbrace{\boxed{\begin{array}{l}\mathsf{ins}_q \\ \mathit{rest}_q\end{array}} \begin{array}{l}\rightarrow b_1 \\ \cdots \\ \rightarrow b_m\end{array}}_{b_q} \parallel \sigma_q \rangle :: a_q$, with $\mathit{ins}_q(\sigma_q)$ defined. This execution must have the form

$$\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \overbrace{\boxed{\begin{array}{l}\mathsf{ins}_p \\ \mathit{rest}_p\end{array}} \begin{array}{l}\rightarrow b'_1 \\ \cdots \\ \rightarrow b'_{m'}\end{array}}^{b_p} \parallel \sigma_p \rangle :: a_p \Rightarrow^{n+1-n_p} \langle b_q \parallel \sigma_q \rangle :: a_q \tag{4.10}$$

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 A_a be the static type information and the approximation of the property of interest at the ACG node $\boxed{\mathsf{ins}_a}$ respectively. By inductive hypothesis we know that $\sigma_p \in \gamma_{\tau_p}(A_p)$ and we show that also $\sigma_q \in \gamma_{\tau_q}(A_q)$ 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 Equation 4.10.

Rule (1)**.** Then $\mathit{ins}_p(\sigma_p)$ is defined and ins_p is not a call.

case a: ins_p is not a return nor a throw

If rest_p is non-empty then, by the maximality of n_p, (4.10) must be

$$\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\begin{array}{l}\mathsf{ins}_p \\ \mathsf{ins}_q \\ \mathit{rest}_q\end{array}} \begin{array}{l}\rightarrow b_1 \\ \cdots \\ \rightarrow b_m\end{array}}_{b_p} \parallel \sigma_p \rangle :: a_p \overset{(1)}{\Rightarrow} \langle \underbrace{\boxed{\begin{array}{l}\mathsf{ins}_q \\ \mathit{rest}_q\end{array}} \begin{array}{l}\rightarrow b_1 \\ \cdots \\ \rightarrow b_m\end{array}}_{b_q} \parallel \underbrace{\mathit{ins}_p(\sigma_p)}_{\sigma_q} \rangle :: \underbrace{a_p}_{a_q}.$$

Otherwise $m' \geq 1$ must hold (legal Java bytecode can only end with a return or a $\mathsf{throw}\,\kappa$) and, by the maximality of n_p, it must be the case that $b_q = b'_h$ for a suitable $1 \leq h \leq m'$, so that (4.10) must have the form

$$\begin{aligned} \langle b_{\mathit{first}(\mathsf{main})} \parallel \xi \rangle \Rightarrow^{n_p} & \langle \underbrace{\boxed{\mathsf{ins}_p} \begin{array}{l}\rightarrow b'_1 \\ \cdots \\ \rightarrow b'_{m'}\end{array}}_{b_p} \parallel \sigma_p \rangle :: a_p \\ \overset{(1)}{\Rightarrow} & \langle \boxed{\phantom{\mathsf{ins}_p}} \begin{array}{l}\rightarrow b'_1 \\ \cdots \\ \rightarrow b'_{m'}\end{array} \parallel \overbrace{\mathit{ins}_p(\sigma_p)}^{\sigma_q} \rangle :: \overbrace{a_p}^{a_q} \\ \overset{(6)}{\Rightarrow} & \langle b_q \parallel \sigma_q \rangle :: a_q. \end{aligned}$$

In both cases, the ACG contains either a sequential or an exceptional arc from $\boxed{\mathsf{ins}_p}$ to $\boxed{\mathsf{ins}_q}$ and $A_q = \Pi(A_p)$, where Π is the propagation rule of the arc. We have:

$$\begin{aligned} \Xi_{\tau_q} \ni \sigma_q = & \quad \mathit{ins}_p(\sigma_p) & \\ \in & \quad \mathit{ins}_p(\gamma_{\tau_p}(A_p)) \cap \Xi_{\tau_p} & \text{[By hypothesis and by monotonicity of } \mathit{ins}_p\text{]} \\ \subseteq & \quad \gamma_{\tau_q}(\Pi(A_p)) = \gamma_{\tau_q}(A_q) & \text{[By Requirements 4.4 and 4.4].} \end{aligned}$$

case b: ins_p is a $\mathsf{return}\,\mathsf{t}$

We show the case when $\mathsf{t} \neq \mathsf{void}$, since the other case is simpler (there is no return value to consider). Then $rest_p$ is empty and $m' = 0$ (no code is executed after a return in legal Java bytecode, but the method terminates) and since $ins_p(\sigma_p) \in \Xi$ (definition of returnt), (4.10) must be in one of these two forms, depending on the emptiness of block b in Rule (4):

$$\begin{aligned}
\langle b_{first(\mathsf{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\mathsf{return\ t}}}_{b_p} \parallel \underbrace{\langle\langle \mathsf{l}_p \parallel t :: \mathsf{s}_p \rangle, \mu_p \rangle\rangle}_{\sigma_p} :: \overbrace{\underbrace{\langle b_q \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c \rangle, \mu_c \rangle\rangle}_{a_p}}^{\texttt{call-time}} :: a_q \\
\overset{(1)}{\Rightarrow} \quad \langle \boxed{} \parallel \langle\langle \mathsf{l}_p \parallel t \rangle, \mu_p \rangle\rangle :: \langle b_q \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c \rangle, \mu_c \rangle\rangle :: a_q \\
\overset{(4)}{\Rightarrow} \quad \langle b_q \parallel \overbrace{\langle\langle \mathsf{l}_c \parallel t :: \mathsf{s}_c \rangle, \mu_p \rangle\rangle}^{\sigma_q} :: a_q
\end{aligned} \tag{4.11}$$

or

$$\begin{aligned}
\langle b_{first(\mathsf{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\mathsf{return\ t}}}_{b_p} \parallel \underbrace{\langle\langle \mathsf{l}_p \parallel t :: \mathsf{s}_p \rangle, \mu_p \rangle\rangle}_{\sigma_p} :: \overbrace{\underbrace{\langle \boxed{} \begin{matrix} \to b'_1 \\ \cdots \\ \to b'_{m'} \end{matrix} \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c \rangle, \mu_c \rangle\rangle}_{a_p}}^{\texttt{call-time}} :: a_q \\
\overset{(1)}{\Rightarrow} \quad \langle \boxed{} \parallel \langle\langle \mathsf{l}_p \parallel t \rangle, \mu_p \rangle\rangle :: \langle \boxed{} \begin{matrix} \to b'_1 \\ \cdots \\ \to b'_{m'} \end{matrix} \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c \rangle, \mu_c \rangle\rangle :: a_q \\
\overset{(4)}{\Rightarrow} \quad \langle \boxed{} \begin{matrix} \to b'_1 \\ \cdots \\ \to b'_{m'} \end{matrix} \parallel \langle\langle \mathsf{l}_c \parallel t :: \mathsf{s}_c \rangle, \mu_p \rangle\rangle :: a_q \\
\overset{(6)}{\Rightarrow} \quad \langle b_q \parallel \langle\langle \mathsf{l}_c \parallel t :: \mathsf{s}_c \rangle, \mu_p \rangle\rangle :: a_q
\end{aligned}$$

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 (4.11), the other being similar. Consider the configuration `call-time`. Since only Rule (2) can stack configurations, `call-time` was the topmost one when a call was executed and, for a suitable $1 \leq w \leq n$, (4.11) must have the form

$$\begin{aligned}
&\langle b_{first(\mathsf{main})} \parallel \xi \rangle \\
&\Rightarrow^{n_c} \langle \boxed{\begin{matrix} \mathsf{call}\ m_1 \dots m_n \\ \mathsf{ins}_q \\ rest_q \end{matrix}} \begin{matrix} \to b'_1 \\ \cdots \\ \to b'_{m'} \end{matrix} \parallel \underbrace{\langle\langle \mathsf{l}_c \parallel v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0 \rangle, \mu_c \rangle\rangle}_{\sigma_c} :: a_q \\
&\overset{(2)}{\Rightarrow} \quad \langle b_{first(m_w)} \parallel \langle\langle [v_{j-\pi} :: \ldots :: v_{j-1}] \parallel \epsilon \rangle, \mu_c \rangle\rangle :: a_p \\
&\Rightarrow^{n_p - n_c - 1} \langle b_p \parallel \sigma_p \rangle :: a_p \\
&\overset{(1)}{\Rightarrow} \quad \langle \boxed{} \parallel \langle\langle \mathsf{l}_p \parallel t \rangle, \mu_p \rangle\rangle :: a_p \\
&\overset{(4)}{\Rightarrow} \quad \langle b_q \parallel \sigma_q \rangle :: a_q,
\end{aligned}$$

where j is the number of stack elements before $\mathsf{ins}_c = \mathsf{call}\ m_1 \dots m_q$ is executed, π is the number of parameters of method m, $b_{first(m_w)}$ is the block where the implementation of m_w starts and the rules in the portion $\Rightarrow^{n_p - n_c - 1}$ never make the stack lower than at the beginning of that portion. Moreover, only in this proof we slightly abuse notation and use $v_0, \dots, v_{j-1}$ to denote the values of variables $v_0, \dots, v_{j-1}$ in σ_c.

Consider $\sigma_c = \langle\langle \mathsf{l}_c \parallel v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0 \rangle, \mu_c \rangle$ and $\sigma_p = \langle\langle \mathsf{l}_p \parallel t :: \mathsf{s}_p \rangle, \mu_p \rangle$. By inductive hypothesis for n_c and n_p we know that $\sigma_c \in \gamma_{\tau_c}(A_c)$ and $\sigma_p \in \gamma_{\tau_p}(A_p)$. Let $\sigma_e = return\,\mathsf{t}(\sigma_p) = \langle\langle \mathsf{l}_p \parallel t \rangle, \mu_p \rangle$. Then, the ACG contains a final arc from $\boxed{\mathsf{return\ t}}$ to $\boxed{\mathsf{exit@}\, m_w{:}\mathsf{t}}$, for a suitable $1 \leq w \leq n$, and $A_e = \Pi(A_p)$, where Π is the propagation rule of the arc. The following relations hold

$$\begin{aligned}
\sigma_e &= \mathit{return}\ \mathsf{t}(\sigma_p) \\
&\in \mathit{return}\ \mathsf{t}(\gamma_{\tau_p}(A_p)) && \text{[By hypothesis and monotonicity of } \mathit{return}\text{]} \\
&\subseteq \gamma_{\tau_e}(\Pi(A_p)) = \gamma_{\tau_e}(A_e) && \text{[By Requirement 4.5].}
\end{aligned}$$

In this case there are two $2{-}1$ arcs (a return value and a side-effect arc) going into $\boxed{\mathsf{ins}_q}$ (see Fig. 4.3), and A_c and A_e represent the correct approximations of the property of interest at the sources of these arcs. Let $A_q = \Pi^{RV}(A_c, A_e) \sqcup \Pi^{SE}(A_c, A_e)$, where Π^{RV} and Π^{SE} are the propagation rules of the return value and side-effect arcs respectively. We have

$$\begin{aligned}
\Xi_{\tau_q} \ni \sigma_q &= d((\mathit{makescope}\ m_w)(\sigma_c)) \\
&\in d((\mathit{makescope}\ m_w)(\gamma_{\tau_c}(A_c))) \cap \Xi_{\tau_q} && \text{[By hypothesis and monotonicity of } d\text{]} \\
&\subseteq \gamma_{\tau_q}(\Pi^{RV}(A_c, A_e) \sqcup \Pi^{SE}(A_c, A_e)) && \text{[By Requirement 4.9]} \\
&= \gamma_{\tau_q}(A_q).
\end{aligned}$$

case c: ins_p is a throw

If rest_p is empty and $m' > 0$, the execution (4.10) must have the form

$$\begin{aligned}
\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi \rangle &\Rightarrow^{n_p} \langle \underbrace{\boxed{\mathsf{throw}\ \kappa}}_{b_p} \overset{b'_1}{\underset{b'_{m'}}{\rightarrow \cdots \rightarrow}} \underbrace{\langle \langle \mathsf{l}_p \parallel e :: \mathsf{s}_p \rangle, \mu_p \rangle}_{\sigma_p} \rangle :: a_p \\
&\overset{(1)}{\Rightarrow} \langle \square \overset{b'_1}{\underset{b'_{m'}}{\rightarrow \cdots \rightarrow}} \overbrace{\langle \langle \mathsf{l}_p \parallel e \rangle, \mu_p \rangle}^{\sigma_q} \rangle :: a_p \\
&\overset{(6)}{\Rightarrow} \langle b_q \parallel \sigma_q \rangle :: \overbrace{a_p}^{a_q},
\end{aligned}$$

where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \le h \le m'$. If rest_p is non-empty, the execution (4.10) must have the form

$$\begin{aligned}
\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi \rangle &\Rightarrow^{n_p} \langle \underbrace{\boxed{\begin{array}{c}\mathsf{throw}\ \kappa \\ \mathsf{catch} \\ \mathit{rest}_q\end{array}}}_{b_p} \overset{b_1}{\underset{b_m}{\rightarrow \cdots \rightarrow}} \underbrace{\langle \langle \mathsf{l}_p \parallel e :: \mathsf{s}_p \rangle, \mu_p \rangle}_{\sigma_p} \rangle :: a_p \\
&\overset{(1)}{\Rightarrow} \langle \overbrace{\boxed{\begin{array}{c}\mathsf{catch} \\ \mathit{rest}_q\end{array}}}^{b_q} \overset{b'_1}{\underset{b'_{m'}}{\rightarrow \cdots \rightarrow}} \overbrace{\langle \langle \mathsf{l}_p \parallel e \rangle, \mu_p \rangle}^{\sigma_q} \rangle :: \overbrace{a_p}^{a_q}
\end{aligned}$$

since catch is the only bytecode whose semantics can be defined on the exceptional state $\sigma_q \in \underline{\Xi}_{\tau_q}$. In both these cases, by inductive hypothesis we have $\sigma_p \in \gamma_{\tau_p}(A_p)$, the ACG contains an exceptional arc from $\boxed{\mathsf{throw}\ \kappa}$ to $\boxed{\mathsf{catch}}$, and $A_q = \Pi(A_p)$, where Π is the propagation rule of the arc. We have

$$\begin{aligned}
\underline{\Xi}_{\tau_q} \ni \sigma_q &= \mathit{throw}\ \kappa(\sigma_p) \\
&\in \mathit{throw}\ \kappa(\gamma_{\tau_p}(A_p)) \cap \underline{\Xi}_{\tau_q} && \text{[By hypothesis and monotonicity of } \mathit{throw}\text{]} \\
&\subseteq \gamma_{\tau_q}(\Pi(A_p)) = \gamma_{\tau_q}(A_q) && \text{[By Requirement 4.6].}
\end{aligned}$$

If rest_p is empty and $m' = 0$, the execution (4.10) must have one of these two forms, depending on the emptiness of block b in Rule (5):

$$
\begin{aligned}
\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi\rangle &\Rightarrow^{n_p} \langle \underbrace{\boxed{\mathsf{throw}\ \kappa}}_{b_p} \parallel \underbrace{\langle\langle \mathsf{l}_p \parallel e :: \mathsf{s}_p\rangle, \mu_p\rangle\rangle}_{\sigma_p} :: \overbrace{\underbrace{\langle b_q \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c\rangle, \mu_c\rangle\rangle :: a_q}_{a_p}}^{\texttt{call-time}} \\
&\overset{(1)}{\Rightarrow} \langle \boxed{} \parallel \underbrace{\langle\langle \mathsf{l}_p \parallel e\rangle, \mu_p\rangle\rangle}_{\sigma_q} :: \langle b_q \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c\rangle, \mu_c\rangle\rangle :: a_q \\
&\overset{(5)}{\Rightarrow} \langle b_q \parallel \overbrace{\underline{\langle\langle \mathsf{l}_c \parallel e\rangle, \mu_p\rangle\rangle}}^{\sigma_q} :: a_q,
\end{aligned}
\tag{4.12}
$$

or

$$
\begin{aligned}
\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi\rangle &\Rightarrow^{n_p} \langle \underbrace{\boxed{\mathsf{throw}\ \kappa}}_{b_p} \parallel \underbrace{\langle\langle \mathsf{l}_p \parallel e :: \mathsf{s}_p\rangle, \mu_p\rangle\rangle}_{\sigma_p} :: \overbrace{\underbrace{\langle \boxed{} \overset{b'_1}{\underset{b'_{m'}}{\rightarrow\!\vdots\!\rightarrow}} \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c\rangle, \mu_c\rangle\rangle :: a_q}_{a_p}}^{\texttt{call-time}} \\
&\overset{(1)}{\Rightarrow} \langle \boxed{} \parallel \underbrace{\langle\langle \mathsf{l}_p \parallel e\rangle, \mu_p\rangle\rangle}_{\sigma_q} :: \langle \boxed{} \overset{b'_1}{\underset{b'_{m'}}{\rightarrow\!\vdots\!\rightarrow}} \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c\rangle, \mu_c\rangle\rangle :: a_q \\
&\overset{(5)}{\Rightarrow} \langle \boxed{} \overset{b'_1}{\underset{b'_{m'}}{\rightarrow\!\vdots\!\rightarrow}} \parallel \overbrace{\underline{\langle\langle \mathsf{l}_c \parallel e\rangle, \mu_p\rangle\rangle}}^{\sigma_q} :: a_q \\
&\overset{(6)}{\Rightarrow} \langle b_q \parallel \sigma_q\rangle :: a_q
\end{aligned}
$$

where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \le h \le m'$. We only prove (4.12), the other being similar. Consider configuration `call-time`. Since only Rule (2) can stack configurations, it was the tompost one when the call was executed and (4.12) must have the form

$$
\begin{aligned}
\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi\rangle &\Rightarrow^{n_c} \langle \boxed{\begin{array}{c}\mathsf{call}\ m_1 \dots m_n \\ \mathsf{ins}_q \\ \mathit{rest}_q\end{array}} \overset{b'_1}{\underset{b'_{m'}}{\rightarrow\!\vdots\!\rightarrow}} \parallel \overbrace{\langle\langle \mathsf{l}_c \parallel v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0\rangle, \mu_c\rangle\rangle}^{\sigma_c} :: a_q \\
&\overset{(2)}{\Rightarrow} \langle b_{\mathit{first}(m_w)} \parallel \langle\langle [v_{j-\pi} :: \ldots :: v_{j-1}] \parallel \epsilon\rangle, \mu_q\rangle\rangle :: \langle b_q \parallel \langle\langle \mathsf{l}_q \parallel \mathsf{s}_q\rangle, \mu_q\rangle\rangle :: a_q \\
&\Rightarrow^{n_p - n_c - 1} \langle b_p \parallel \sigma_p\rangle :: a_p \\
&\overset{(1)}{\Rightarrow} \langle \boxed{} \parallel \underline{\langle\langle \mathsf{l}_p \parallel e\rangle, \mu_p\rangle\rangle} :: a_p \\
&\overset{(5)}{\Rightarrow} \langle b_q \parallel \sigma_q\rangle :: a_q,
\end{aligned}
$$

where j is the number of stack elements before $\mathsf{ins}_c = \mathsf{call}\ m_1 \dots m_q$ is executed, π is the number of parameters of method m, $b_{\mathit{first}(m_w)}$ is the block where the implementation of m_w starts 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 recall that, only in this proof, we slightly abuse notation and use $v_0, \dots, v_{j-1}$ to denote the values of variables $v_0, \dots, v_{j-1}$ in σ_c. By the semantics of Java bytecode, since $\sigma_q \in \underline{\Xi}$, the only possibility for ins_q is to be a catch.

Consider $\sigma_c = \langle\langle \mathsf{l}_c \parallel v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0\rangle, \mu_c\rangle$ and $\sigma_p = \langle\langle \mathsf{l}_p \parallel e :: \mathsf{s}_p\rangle, \mu_p\rangle$. By inductive hypothesis for n_c and n_p we know that $\sigma_c \in \gamma_{\tau_c}(A_c)$ and $\sigma_p \in \gamma_{\tau_p}(A_p)$. Let $\sigma_e = \mathit{throw}\ \kappa(\sigma_p) = \underline{\langle\langle \mathsf{l}_p \parallel e\rangle, \mu_p\rangle}$. Then, the ACG contains a final arc from $\boxed{\mathsf{throw}\ \kappa}$ to $\boxed{\mathsf{exit@}\, m_w{:}\mathsf{t}}$, for a suitable $1 \le w \le n$, $A_e = \Pi(A_p)$, where Π is the propagation rule #13 (Definition 4.1), and the following relations hold

$$
\begin{aligned}
\sigma_e &= \mathit{throw}\ \mathsf{t}(\sigma_p) \\
&\in \mathit{throw}\ \mathsf{t}(\gamma_{\tau_p}(A_p)) && \text{[By hypothesis and monotonicity of } \mathit{throw}\text{]} \\
&\subseteq \gamma_{\tau_e}(\Pi(A_p)) = \gamma_{\tau_e}(A_e) && \text{[By Requirement 4.5].}
\end{aligned}
$$

In this case there are two arcs (a side-effect and an exceptional arc) going into $\boxed{\mathsf{catch}}$ (see Fig. 4.4), and A_c and A_e represent the correct approximations of the property of interest at the sources of these arcs. Let $A_q = \Pi^E(A_c) \sqcup \Pi^{SE}(A_c, A_e)$, where Π^E and Π^{SE} are the propagation rules of the exceptional and the side-effects arcs respectively. We have

$$\begin{aligned}
\underline{\Xi}_{\tau_q} \ni \sigma_q &= d((\mathit{makescope}\ m_w)(\sigma_c)) \\
&\in d((\mathit{makescope}\ m_w)(\gamma_{\tau_c}(A_c))) \cap \underline{\Xi}_{\tau_q} && \text{[By hypothesis and monotonicity of } d\text{]} \\
&\subseteq \gamma_{\tau_q}(\Pi^E(A_c) \sqcup \Pi^{SE}(A_c, A_e)) = \gamma_{\tau_q}(A_q) && \text{[By Requirement 4.11].}
\end{aligned}$$

Rule (2)**.** By definition of *makescope*, (4.10) must have the form

$$\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi\rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\mathsf{call}\ m_1 \dots m_n} \to \begin{matrix} b'_1 \\ \dots \\ b'_{m'} \end{matrix}}_{b_p} \parallel \underbrace{\langle\langle \mathsf{l}_p \parallel v_{j-1} :: \dots :: v_{j-\pi} :: \dots :: v_0\rangle, \mu_p\rangle\rangle}_{\sigma_p} :: a_p$$
$$\overset{(2)}{\Rightarrow} \langle \overbrace{b_{\mathit{first}(m_w)}}^{b_q} \parallel \overbrace{\langle\langle [v_{j-\pi} :: \dots :: v_{j-1}] \parallel \epsilon\rangle, \mu_p\rangle\rangle}^{\sigma_q} :: a_q,$$

where j is the number of stack elements before $\mathsf{call}\ m_1 \dots m_q$ is executed, π is the number of parameters of method m and $b_{\mathit{first}(m_w)}$ is the block where the implementation of m_w starts. In this case, the ACG contains a parameter passing arc from $\boxed{\mathsf{call}\ m_1 \dots m_q}$ to $\boxed{\mathit{first}(m_w)}$, where $\mathit{first}(m_w)$ is the first instruction of m_w for a suitable $w \in [1..n]$ and $A_q = \Pi(A_p)$, where Π is the propagation rule of the arc. We have

$$\begin{aligned}
\sigma_q &= \mathit{makescope}(\sigma_p) \\
&\in \mathit{makescope}(\gamma_{\tau_p}(A_p)) && \text{[By hypothesis and monotonicity of } \mathit{makescope}\text{]} \\
&\subseteq \gamma_{\tau_q}(\Pi(A_p)) = \gamma_{\tau_q}(A_q) && \text{[By Requirement 4.8].}
\end{aligned}$$

Rule (3)**.** Let i and j be the number of local variables and stack elements before $\mathsf{ins}_p = \mathsf{call}\ m_1 \dots m_q$ is executed and π be the number of parameters of methods m_w. In this case, (4.10) must have the form

$$\begin{aligned}
&\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi\rangle \\
&\Rightarrow^{n_p} \langle \underbrace{\begin{matrix}\mathsf{call}\ m_1 \dots m_n \\ \mathit{rest}_p\end{matrix} \to \begin{matrix} b'_1 \\ \dots \\ b'_{m'} \end{matrix}}_{b_p} \parallel \underbrace{\langle\langle \mathsf{l}_p \parallel v_{j-1} :: \dots :: v_{j-\pi+1} :: \mathtt{null} :: \dots :: v_0\rangle, \mu_p\rangle\rangle}_{\sigma_p} :: a_p \\
&\overset{(3)}{\Rightarrow} \langle \overbrace{\boxed{\mathit{rest}_p}}^{b_q} \to \begin{matrix} b'_1 \\ \dots \\ b'_{m'} \end{matrix} \parallel \overbrace{\underline{\langle\langle \mathsf{l}_p \parallel \ell\rangle, \mu_p[\ell \mapsto \mathit{npe}]\rangle\rangle}}^{\sigma_q} :: a_q
\end{aligned}$$

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

$$\begin{aligned}
&\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi\rangle \\
&\Rightarrow^{n_p} \langle \underbrace{\mathsf{call}\ m_1 \dots m_n \to \begin{matrix} b'_1 \\ \dots \\ b'_{m'} \end{matrix}}_{b_p} \parallel \underbrace{\langle\langle \mathsf{l}_p \parallel v_{j-1} :: \dots :: v_{j-\pi+1} :: \mathtt{null} :: \dots :: v_0\rangle, \mu_p\rangle\rangle}_{\sigma_p} :: a_p \\
&\overset{(3)}{\Rightarrow} \langle \square \to \begin{matrix} b'_1 \\ \dots \\ b'_{m'} \end{matrix} \parallel \overbrace{\underline{\langle\langle \mathsf{l}_p \parallel \ell\rangle, \mu_p[\ell \mapsto \mathit{npe}]\rangle\rangle}}^{\sigma_q} :: a_q \\
&\overset{(6)}{\Rightarrow} \langle b_q \parallel \sigma_q\rangle :: a_q
\end{aligned}$$

where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \leq h \leq m'$. In both cases, the ACG contains an exceptional arc from $\boxed{\mathsf{ins}_p}$ to $\boxed{\mathsf{ins}_q}$, and $A_q = \Pi(A_p)$, where Π is the propagation rule of the arc. We have

$$\begin{aligned} \Xi_{\tau_q} \ni \sigma_q &= d((\mathit{makescope}\ m_w)(\sigma_p)) \cap \Xi_{\tau_q} \\ &\subseteq \gamma_{\tau_q}(\Pi(A_p)) = \gamma_{\tau_q}(A_q) \qquad \text{[By Requirement 4.7].} \end{aligned}$$

■

5

Possible Reachability Analysis of Program Variables

In this chapter we present the first instantiation of the general parameterized framework for constraint-based static analyses defined in Chapter 4. Namely, we define a new abstract domain REACH for the static analysis of the *reachability between program variables*, through dynamically allocated memory locations. Reachability from a program variable v to a program variable w states that starting from v it is possible to follow a path of memory locations that leads to the object bound to w. This useful piece of information is important for improving the precision of other static analyses, such as side-effects, field initialization, cyclicity and path-length analysis, as well as more complex analyses built upon them, such as nullness and termination analysis. We define the notion of reachability, instantiate the parameters of the framework and prove that the requirements imposed by the framework are satisfied. This result implies the soundness of our reachability analysis.

Our reachability analysis is an example of a *possible analysis*. Namely, for each program point p, we determine a set of ordered pairs of variables of reference type $\langle v, w \rangle$ such that v *might reach* w at p when the program is executed on an arbitrary input. On the other hand, if a pair $\langle v, w \rangle$ is not present in our over-approximation at p, it means that v *definitely does not reach* w at p. We have implemented the analysis inside the Julia static analyzer. Our experiments of analysis of non-trivial Java and Android programs show the improvement of precision due to the presence of reachability information. As a side-effect, reachability analysis actually reduces the overall costs of nullness and termination analysis.

This chapter is based on the work published in [64] and its extended version [61].

5.1 Introduction

In this chapter we present the first instantiation of the general parameterized framework for constraint-based static analyses defined in Chapter 4. Namely, we define a new abstract domain REACH for the static analysis of the *reachability between program variables*, through dynamically allocated memory locations. We say that a variable v *reaches* a variable w if w is bound to an object reachable from v, by following the fields of the object bound to v, recursively. This notion is distinct from sharing: if v reaches w then v and w share, but the converse is in general false. In this sense reachability is more *precise*, that is, it induces a finer, more concrete abstraction of the computational states than sharing

analysis. Reachability can of course be abstracted from very precise abstractions of the memory, such as the result of a shape analysis. However, we want an analysis that uses the most abstract domain for reachability analysis, which coincides with the reachability property itself. Hence, in this chapter, our abstract domain REACH will just be made of ordered pairs of variables $\langle v, w \rangle$, stating that v reaches w. Those pairs are propagated along all possible execution paths by using a constraint-based technique, proved correct by abstract interpretation. The implementation has been performed inside the Julia analyzer [4], which allows us to discuss the actual benefits of our reachability analysis. It is worth noting that our reachability analysis provides, for *each program point*, an *over-approximation* of the actual reachability information available at that point. The over-approximation corresponding to a given program point contains the complete actual reachability information concerning that point, but it might also contain some *spurious* pairs of variables (i.e., *false negatives*) due to the simple abstract domain that we use. On the other hand, we are sure that the pairs of variables which are *not* included in the over-approximation are *definitely not reachable* one from another at that program point.

Both reachability (i.e., calculated over-approximation) and non-reachability (i.e., pairs not belonging to the calculated over-approximation) are useful in many situations. By just considering our work related to the Julia static analyzer, we find the following uses:

for side-effects analysis Side-effects analysis tracks (among other things) which parameters `p` of a method might be affected by its execution in the sense that the method might update a field of an object reachable from `p`. Namely, if the method performs an assignment `a.f=b`, this affects `p` only if `p` reaches `a`. Therefore, if we know that `p` *definitely does not reach* a right before the assignment is performed, the latter does not affect `p`. If we used non-sharing rather than non-reachability information, that choice would lead to a loss of precision, since it might be the case that `p` and `a` share but the assignment modifies an object unreachable from `p`.

for field initialization analysis It is often the case that a field is initialized by all of the constructors of its defining class *before that field is read* by any of these constructors. Spotting this frequent situation is important for many analyses, including nullness [68, 86]. Hence, we want to know whether a field read operation `a=expression.f` inside a constructor can actually read the field `f` of the `this` object, which is being initialized by its constructor. This might happen only if `this` reaches `expression`. Hence, if we know that `this` *definitely does not reach* `expression` right before the assignment is executed, that assigment will not read the field `f` of `this`. Again, sharing would be less precise here.

for cyclicity analysis Having an acyclic pointer `b`, i.e., a pointer to a cyclical data structure, an assignment `a.f=b` might make `a` *cyclical* (i.e., point to a cyclical data structure), only if `b` reaches `a`. Originally, this analysis was built upon sharing information [74], but analysis of reachable variables gives better precision.

for path-length analysis Path-length is a data structure measure used in termination analysis [87]. It is the maximum number of pointer dereferences that can be followed from a program variable. An assignment `a.f=b` can only modify the path-length of the program variables that share with `a`, according to the original definition of path-length [87]. Reachability analysis improves this approximation, since the path-length of a program variable `v` is actually modified only if `v` *reaches* `a`.

Julia already includes the four static analyses mentioned above (among tens of others). These analyses are used as building blocks of larger *tools*, such as nullness checker and termination checker tools. The former spots the points in which a program might throw a null-pointer exception at run-time, while the latter spots which method calls might diverge at run-time. A tool performs its supporting static analyses (building blocks) in distinct threads and hence runs in parallel on multicore hardware. When a supporting static analysis needs the results of another analysis, it suspends itself until those results become available. The analyzer does not deadlock since a partial ordering is imposed on the analyses: if an analysis x needs the results of an analysis y, then y never asks for the results of x, not even indirectly.

At the end of this chapter, we provide an experimental evaluation of our reachability analysis. Namely, we show that reachability analysis is more precise than a sharing analysis, when reachability information must be computed. We also report the effects of the reachability analysis on the precision of side-effects, field initialization and cyclicity analyses. The effects on path-length analysis can only be measured indirectly, by checking if the termination analysis, built over the path-length analysis, increases its precision. We show that reachability increases the overall precision of the nullness tool of Julia, for the analysis of non-trivial Java and Android programs. On the other hand, the performance of the termination tool is not improved for the programs mentioned above. Instead, it is well improved for the analysis of a set of programs from the international termination competition [2], where 6 more examples are shown to terminate thanks to the addition of reachability analysis. We explain this with the observation that, in most real cases such as the large programs that we have analyzed, termination is related to loops over integer counters rather than to recursion over recursive data structures. The samples from the termination competition are small (a few hundreds lines of source code), which means that the shape of the memory can be more easily inferred; they ban complications such as static fields and calls to the Java library; they are often devised with the goal of showing specific features of the competing analyzers and are consequently often unrealistic. An unexpected and surprising effect of reachability is, however, an increase in speed for both tools: adding an extra static analysis (reachability) reduces the total run-time the tools (reachability run-time included). This can be actually explained: reachability increases the precision of other analyses (side-effects, field initialization, cyclicity...) and hence helps their convergence and makes them use smaller abstractions (i.e., they track less spurious information). Moreover, reachability is run in parallel to other analyses, so that it does not actually add to the total cost of the tools (as long as enough processing cores are available).

The rest of the chapter is organized as follows. Section 5.2 introduces different notions of reachability needed by our formalization. In Section 5.3 we show how different parameters of the constraint-based framework can be instantiated in order to obtain an over-approximation of the actual reachability information available at every program point statically. Section 5.4 shows that this instantiation actually satisfies the requirements imposed by the framework, which implies that our reachability analysis has the least solution and that it is sound. In Section 5.5 we show the experimental evaluation of the implementation of our reachability analysis inside the Julia static analyzer: we show how the precision of Julia improved after the reachability analysis was implemented.

5.2 Property of Reachability Between Variables

In this section we formalize the notion of *reachability* between two program variables. In order to do that we first determine the locations reachable from an arbitrary location ℓ. Intuitively, we collect all the locations held in the fields of the object bound to ℓ, or in the elements of the array bound to ℓ. We then consider the contents of the fields of the objects or elements of the arrays held at these locations and so on until a fixpoint is reached. Let us formalize this intuition.

Definition 5.1 (Locations reachable from a location). *Given $\tau \in \mathcal{T}$, we define the set of* locations reachable from a location $\ell \in \mathbb{L}$ in a memory μ *as* $\mathsf{L}_\mu(\ell) = \bigcup_{i \geq 0} \mathsf{L}^i_\mu(\ell)$*, where* $\mathsf{L}^i_\mu(\ell)$ *are the locations reachable from ℓ in at most i steps, defined as:*

$$\mathsf{L}^i_\mu(\ell) = \begin{cases} \ell & \text{if } i = 0 \\ \mathsf{L}^{i-1}_\mu(\ell) \cup \bigcup_{\ell_1 \in \mathsf{L}^{i-1}_\mu(\ell)} (\mathsf{rng}(\mu(\ell_1).\phi) \cap \mathbb{L}) & \text{if } i > 0 \end{cases}$$

Hence, if an object (an array) $\mu(\ell_1)$ is bound to a location reachable from ℓ, then also $\mathsf{rng}(\mu(\ell_1).\phi) \cap \mathbb{L}$, the locations held in $\mu(\ell_1)$'s fields (elements) are reachable from ℓ.

Lemma 5.2 is a technical result stating that if we write a location ℓ'' into a field (element) of an object (array), then the set of locations reachable from a given location ℓ might be enlarged with at most the locations reachable from ℓ''.

Lemma 5.2. *Let μ be a memory, $\ell', \ell'' \in \mathsf{dom}(\mu)$ and $d \in \mathsf{dom}(\mu(\ell').\phi)$ (d is a field of $\mu(\ell').\phi$ if $\mu(\ell).\mathtt{type} \in \mathbb{K}$ or an index of $\mu(\ell).\mathtt{type} \in \mathbb{A}$). Let $\mu' = \mu[(\mu(\ell').\phi)(d) \mapsto \ell'']$. Then $\mathsf{L}_{\mu'}(\ell) \subseteq \mathsf{L}_\mu(\ell) \cup \mathsf{L}_\mu(\ell'')$ for all $\ell \in \mathsf{dom}(\mu)$.*

Proof. Let $\ell \in \mathsf{dom}(\mu)$. We prove, by induction on i, that $\mathsf{L}^i_{\mu'}(\ell) \subseteq \mathsf{L}^i_\mu(\ell) \cup \mathsf{L}^i_\mu(\ell'')$, which entails the thesis. If $i = 0$, we have $\mathsf{L}^0_{\mu'}(\ell) = \{\ell\} \subseteq \{\ell\} \cup \mathsf{L}^0_\mu(\ell'') = \mathsf{L}^0_\mu(\ell) \cup \mathsf{L}^0_\mu(\ell'')$. Assume now that $\mathsf{L}^{i-1}_{\mu'}(\ell) \subseteq \mathsf{L}^{i-1}_\mu(\ell) \cup \mathsf{L}^{i-1}_\mu(\ell'')$. We have

$$\begin{aligned}
&\mathsf{L}^i_{\mu'}(\ell) = \mathsf{L}^{i-1}_{\mu'}(\ell) \cup \bigcup_{\ell_1 \in \mathsf{L}^{i-1}_{\mu'}(\ell)} (\mathsf{rng}(\mu'(\ell_1).\phi) \cap \mathbb{L}) \\
&\subseteq \mathsf{L}^{i-1}_\mu(\ell) \cup \mathsf{L}^{i-1}_\mu(\ell'') \cup \bigcup_{\ell_1 \in \mathsf{L}^{i-1}_\mu(\ell) \cup \mathsf{L}^{i-1}_\mu(\ell'')} (\mathsf{rng}(\mu'(\ell_1).\phi) \cap \mathbb{L}) \\
&= \mathsf{L}^{i-1}_\mu(\ell) \cup \mathsf{L}^{i-1}_\mu(\ell'') \cup \bigcup_{\ell_1 \in \mathsf{L}^{i-1}_\mu(\ell)} (\mathsf{rng}(\mu'(\ell_1).\phi) \cap \mathbb{L}) \cup \bigcup_{\ell_1 \in \mathsf{L}^{i-1}_\mu(\ell'')} (\mathsf{rng}(\mu'(\ell_1).\phi) \cap \mathbb{L}) \\
&= \mathsf{L}^{i-1}_\mu(\ell) \cup \mathsf{L}^{i-1}_\mu(\ell'') \cup \bigcup_{\ell_1 \in \mathsf{L}^{i-1}_\mu(\ell) \setminus \{\ell'\}} (\mathsf{rng}(\mu'(\ell_1).\phi) \cap \mathbb{L}) \cup (\mathsf{rng}(\mu'(\ell').\phi) \cap \mathbb{L}) \\
&\qquad \cup \bigcup_{\ell_1 \in \mathsf{L}^{i-1}_\mu(\ell'') \setminus \{\ell'\}} (\mathsf{rng}(\mu'(\ell_1).\phi) \cap \mathbb{L}) \cup (\mathsf{rng}(\mu'(\ell').\phi) \cap \mathbb{L}) \\
&\subseteq \mathsf{L}^{i-1}_\mu(\ell) \cup \mathsf{L}^{i-1}_\mu(\ell'') \cup \bigcup_{\ell_1 \in \mathsf{L}^{i-1}_\mu(\ell) \setminus \{\ell'\}} (\mathsf{rng}(\mu(\ell_1).\phi) \cap \mathbb{L}) \cup (\mathsf{rng}(\mu(\ell').\phi) \cap \mathbb{L}) \cup \{\ell''\} \\
&\qquad \cup \bigcup_{\ell_1 \in \mathsf{L}^{i-1}_\mu(\ell'') \setminus \{\ell'\}} (\mathsf{rng}(\mu(\ell_1).\phi) \cap \mathbb{L}) \cup (\mathsf{rng}(\mu(\ell').\phi) \cap \mathbb{L}) \cup \{\ell''\} \\
&= \mathsf{L}^{i-1}_\mu(\ell) \cup \mathsf{L}^{i-1}_\mu(\ell'') \cup \bigcup_{\ell_1 \in \mathsf{L}^{i-1}_\mu(\ell)} (\mathsf{rng}(\mu(\ell_1).\phi) \cap \mathbb{L}) \cup \bigcup_{\ell_1 \in \mathsf{L}^{i-1}_\mu(\ell'')} (\mathsf{rng}(\mu(\ell_1).\phi) \cap \mathbb{L}) \cup \{\ell''\} \\
&= \mathsf{L}^i_\mu(\ell) \cup \mathsf{L}^i_\mu(\ell'') \cup \{\ell''\} \\
&= \mathsf{L}^i_\mu(\ell) \cup \mathsf{L}^i_\mu(\ell'')
\end{aligned}$$

since $\ell'' \in \mathsf{L}^i_\mu(\ell'')$. ∎

$\mathsf{L}^0_\sigma(l_1) = \{\ell_2\}$ $\mathsf{L}^1_\sigma(l_1) = \mathsf{L}_\sigma(l_1) = \{\ell_1, \ell_2\}$
$\mathsf{L}^0_\sigma(l_2) = \mathsf{L}_\sigma(l_2) = \varnothing$ $\mathsf{L}^0_\sigma(l_3) = \mathsf{L}_\sigma(l_3) = \{\ell_3\}$
$\mathsf{L}^0_\sigma(l_4) = \{\ell_4\}$ $\mathsf{L}^1_\sigma(l_4) = \{\ell_2, \ell_3, \ell_4\}$ $\mathsf{L}^2_\sigma(l_4) = \mathsf{L}_\sigma(l_4) = \{\ell_1, \ell_2, \ell_3, \ell_4\}$
$\mathsf{L}^0_\sigma(l_5) = \{\ell_5\}$ $\mathsf{L}^1_\sigma(l_5) = \mathsf{L}_\sigma(l_5) = \{\ell_1, \ell_3, \ell_5\}$

Fig. 5.1. Example of computation of reachable locations

We say that a variable a reaches a location ℓ if the former is bound to a location that reaches ℓ.

Definition 5.3 (Locations reachable from a variable). *Given $\tau \in \mathcal{T}$, we define the set of* locations reachable from a variable $a \in \mathsf{dom}(\tau)$ in a state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ *as* $\mathsf{L}_\sigma(a) = \mathsf{L}_\mu(\rho(a))$ *if* $\rho(a) \in \mathbb{L}$ *and* $\mathsf{L}_\sigma(a) = \emptyset$ *otherwise.*

We say that a variable is reachable from another one if the former is bound to a location reachable from the latter.

Definition 5.4 (Reachability between variables). *Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ and variables $a, b \in \mathsf{dom}(\tau)$. We say that b is* reachable *from a in σ or, equivalently, that a reaches b in σ, denoted as $a \rightsquigarrow^\sigma b$, iff $\rho(b) \in \mathsf{L}_\sigma(a)$.*

Remark 5.5. It is worth noting that two variables a and b share in a state σ if and only if $\mathsf{L}_\sigma(a) \cap \mathsf{L}_\sigma(b) \neq \varnothing$.

Example 5.6. Consider the state $\sigma \in \Sigma_\tau$ from Example 3.16. In Fig. 5.1 we give, for each variable $l_i \in \mathsf{dom}(\tau)$, and for every $j \geq 0$, the set of reachable locations from l_i in σ in at most j steps until the fixpoint is reached. Therefore, we conclude that: $l_1 \rightsquigarrow^\sigma l_1$, $l_1 \rightsquigarrow^\sigma l_2$, $l_3 \rightsquigarrow^\sigma l_3$, $l_4 \rightsquigarrow^\sigma l_1$, $l_4 \rightsquigarrow^\sigma l_2$, $l_4 \rightsquigarrow^\sigma l_3$, $l_4 \rightsquigarrow^\sigma l_4$, $l_5 \rightsquigarrow^\sigma l_1$, $l_5 \rightsquigarrow^\sigma l_3$, $l_5 \rightsquigarrow^\sigma l_5$. □

Let us show a very important and useful result: given two variables a and b and two states which assign the same values to both a and b, and each location reachable from the former to the same object, then we can state that a reaches b in the first state if and only if a reaches b in the second one.

Lemma 5.7. *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 $\ell \in \mathsf{L}_\sigma(\rho(a))$ we have $\mu(\ell) = \mu'(\ell)$.*

Then $a \rightsquigarrow^\sigma b$ if and only if $a' \rightsquigarrow^{\sigma'} b'$.

Proof. Since $\rho(b) = \rho'(b')$, it is enough to prove that $\mathsf{L}_\sigma(a) = \mathsf{L}_{\sigma'}(a')$. In fact, if $\mathsf{L}_\sigma(a) = \mathsf{L}_{\sigma'}(a')$ then $\rho(b) \in \mathsf{L}_\sigma(a)$ if and only if $\rho'(b') \in \mathsf{L}_{\sigma'}(a')$, i.e., $a \rightsquigarrow^\sigma b$ if and only if $a' \rightsquigarrow^{\sigma'} b'$. If $\rho(a) = \rho'(a') \notin \mathbb{L}$ then $\mathsf{L}_\sigma(a) = \mathsf{L}_{\sigma'}(a') = \emptyset$ (Definition 5.3) and the thesis trivially holds. Assume that $\rho(a) = \rho'(a') \in \mathbb{L}$. We prove that $\mathsf{L}^i_\mu(\rho(a)) = \mathsf{L}^i_{\mu'}(\rho'(a'))$ for every $i \geq 0$, by induction on i.

Base case: $i = 0$**.** We have $\mathsf{L}^0_\mu(\rho(a)) = \{\rho(a)\} = \{\rho'(a')\} = \mathsf{L}^0_{\mu'}(\rho'(a'))$.

Induction step: assume that $i > 0$ and $\mathsf{L}^{i-1}_\mu(\rho(a)) = \mathsf{L}^{i-1}_{\mu'}(\rho'(a'))$. We have

$$
\begin{aligned}
\mathsf{L}^i_\mu(\rho(a)) &= \mathsf{L}^{i-1}_\mu(\rho(a)) \cup \bigcup_{\ell \in \mathsf{L}^{i-1}_\mu(\rho(a))} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) && \text{[By Definition 5.1]} \\
&= \mathsf{L}^{i-1}_{\mu'}(\rho'(a')) \cup \bigcup_{\ell \in \mathsf{L}^{i-1}_{\mu'}(\rho'(a'))} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) && \text{[By hypothesis]} \\
&= \mathsf{L}^{i-1}_{\mu'}(\rho'(a)) \cup \bigcup_{\ell \in \mathsf{L}^{i-1}_\mu(\rho'(a))} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) && \text{[By points (3) and (4)]} \\
&= \mathsf{L}^i_{\mu'}(\rho'(a')). && \text{[By Definition 5.1]}
\end{aligned}
$$

■

Note that points 3 and 4 of Lemma 6.15 hold, in particular, when $\mu = \mu'$, but the more general form of the lemma will be useful later.

We observe that, in a programming language such as Java bytecode, an activation of a method can only access locations reachable from its actual parameters or allocated during its execution[1]. Hence we can safely state that the activation does not read nor write into the locations $\mathcal{L}$, already allocated at the time of call, but not reachable from the actual parameters of the method. Those locations keep their value unchanged during the execution of the activation of the method. For the same reason, no location in $\mathcal{L}$ is reachable from the return value of the activation of the method, if any. Moreover, the locations in $\mathcal{L}$ are not written inside the fields (respectively, array elements) of the objects (respectively, arrays) reachable by the activation of the method. The following proposition formalizes these intuitions. Although technical, it is important since it bounds the side-effects of a method to the locations not in $\mathcal{L}$. As a consequence, we can be sure that the execution of a method will never affect the locations reachable from variables that do not share with the actual parameters of the call. We will exploit this observation for the definition of the abstract semantics, later, to provide a sound approximation of the side-effects of the execution of a method and of the reachability for its return value.

Proposition 5.8. *Let* $\sigma = \langle\langle \mathsf{l} \parallel \mathsf{s}\rangle, \mu\rangle = \langle\rho, \mu\rangle$ *and* $\sigma' = \langle\langle \mathsf{l}' \parallel \mathsf{s}'\rangle, \mu'\rangle = \langle\rho', \mu'\rangle$ *be the states right before two adjacent bytecode instructions* $\mathsf{ins} = \mathsf{call}\ \kappa_1.m \ldots \kappa_n.m$ *and* $\mathsf{ins}' \neq \mathsf{catch}$ *are executed. Namely,* σ' *is a non-exceptional state obtained at the end of execution of a callee* $\kappa_w.m(\vec{\mathsf{t}})\!:\!\mathsf{t}$ *in* σ*, for a* $w \in [1..n]$*, the topmost* π *stack elements of* σ *(*$s_{|\mathsf{s}|-1}, \ldots, s_{|\mathsf{s}|-\pi}$*) and the topmost operand stack element of* σ' *(*$s_{|\mathsf{s}'|-1}$*) contain the parameters of the callee and its return value, respectively. We define* $\mathcal{L}_\sigma$*, the set of locations not reachable from the actual parameters of the callee in* σ*:*

$$\mathcal{L}_\sigma = \mathsf{dom}(\mu) \setminus \bigcup_{i \in [|\mathsf{s}|-\pi..|\mathsf{s}|)} \mathsf{L}_\sigma(s_i).$$

Then, the following conditions hold:

[1] If we considered full Java bytecode, we would also include the locations reachable from the static fields.

1. $\forall \ell \in \mathcal{L}_\sigma.\mu(\ell) = \mu'(\ell)$,
2. $\mathsf{s}'[|\mathsf{s}'| - 1] \notin \mathcal{L}_\sigma$ *and*
3. $\forall \ell \in \mathsf{dom}(\mu') \setminus \mathcal{L}_\sigma.\mathsf{rng}(\mu'(\ell).\phi) \cap \mathcal{L}_\sigma = \emptyset$.

Proof. It is enough to prove that, during the execution of the callee(s) $m_1, \ldots, m_n$ and of the methods that they might call, the locations held in the stack elements or local variables are not in $\mathcal{L}_\sigma$ and do not reach any location in $\mathcal{L}_\sigma$. This entails the thesis since

1. only $\mathsf{putfield}\, f$ and $\mathsf{arraystore}\ \alpha$ modify objects or arrays in memory. They do it inside an object or array pointed by a location ℓ' on the stack. Since, by hypothesis, $\ell' \notin \mathcal{L}_\sigma$, we have $\mu(\ell) = \mu'(\ell)$ for all $\ell \in \mathcal{L}_\sigma$;
2. the returned value is left on top of the stack of the callee(s). By hypothesis, it does not belong to $\mathcal{L}_\sigma$;
3. at the beginning of the execution of the callee(s), this condition holds since ℓ would be a location reachable from the parameters of the call. Hence $\mu'(\ell).\phi$ can only contain, then, locations not in $\mathcal{L}_\sigma$, since those in $\mathcal{L}_\sigma$ are, by definition, unreachable from the parameters. Later, during the execution of the callee(s), only $\mathsf{putfield}\ f$ and $\mathsf{arraystore}\ \alpha$ modify objects or arrays in memory. They do it by writing, inside a field or an array, a value held on the stack. By hypothesis, that value is not in $\mathcal{L}_\sigma$. Hence also this condition holds.

It remains to prove, then, the invariant that, during the execution of the callee(s) and of the methods that they might call, the locations held in the stack elements or local variables are not in $\mathcal{L}_\sigma$ and do not reach any location in $\mathcal{L}_\sigma$. This holds at the beginning of the execution of the callee(s) since, at the beginning of the execution of a method or constructor, the stack is empty and the local variables hold the actual parameters of the call that, by definition of $\mathcal{L}_\sigma$, are not in $\mathcal{L}_\sigma$ and do not reach any location in $\mathcal{L}_\sigma$. During the subsequent execution of the callee(s) and of the methods or constructors that it might call, most bytecode instructions simply move or duplicate values on the stack or to and from the stack and the local variables, hence keeping the invariant true. Only $\mathsf{putfield}\ f$ and $\mathsf{arraystore}\ \alpha$ modify objects or arrays in memory. They do it by writing, inside a field or an array, a value held on the stack. By hypothesis, that value is not in $\mathcal{L}_\sigma$ and does not reach any location in $\mathcal{L}_\sigma$. Also in this case, the invariant is hence maintained. Finally, instructions $\mathsf{getfield}\ f$ and $\mathsf{arrayload}\ \alpha$ push on the stack a value v reachable from a location ℓ on the operand stack. The invariant entails that ℓ is not in $\mathcal{L}_\sigma$ and does not reach any location in $\mathcal{L}_\sigma$. Hence v, which is reachable from ℓ, is not in $\mathcal{L}_\sigma$ and cannot reach a any location in $\mathcal{L}_\sigma$, or otherwise ℓ would reach the same location, which is impossible. Also in this last case the invariant is hence maintained. ■

Let us explain the meaning of these three points.

1. For each location ℓ, *not reachable* from the actual arguments of the method at ins (i.e., $\ell \in \mathcal{L}_\sigma$), the object or array bound to ℓ at ins ($\mu(\ell)$) is the same one bound to ℓ at ins' ($\mu'(\ell)$), i.e., the execution of the method does not modify the values bound to locations in $\mathcal{L}_\sigma$.
2. The method's return value $\mathsf{s}'[|\mathsf{s}'| - 1]$, at ins' is not a location in $\mathcal{L}_\sigma$. Anyway, that value might be a location ℓ that did not exist at call time (i.e., at ins), when $\ell \notin \mathsf{dom}(\mu)$. In this case, the location ℓ might have been allocated during the execution of the activation of the method and would consequently be bound to a new object or array.

3. Every location ℓ available at ins' (i.e., at the end of the execution of the activation of the method), that moreover does not belong to $\mathcal{L}_\sigma$ (i.e., ℓ is reachable from the actual arguments of the method at ins) can reach only the location which are reachable from the actual arguments at ins or the locations that have been allocated during the execution pf the method.

Lemmas 5.9, 5.10 and 5.11 highlight some important properties of the set of the locations which are not reachable from the actual arguments of a method at the point where that method is invoked ($\mathcal{L}_\sigma$). This set has been introduced in Proposition 5.8. We have used these results in the proofs of Lemmas 4.6-4.8.

Suppose that after a method is executed, there exists a variable bound to a location that was reachable from an actual parameter of the method before its execution (state σ). Lemma 5.9 shows that this variable does not share with any location belonging to $\mathcal{L}_\sigma$.

Lemma 5.9. *Under the hypotheses of Proposition 5.8, consider a variable x such that $\rho'(x) \in \mathbb{L} \setminus \mathcal{L}_\sigma$. Then $\mathsf{L}_{\sigma'}(x) \cap \mathcal{L}_\sigma = \emptyset$.*

Proof. We prove that $\forall i \in \mathbb{N}.\mathsf{L}^i_{\sigma'}(x) \cap \mathcal{L}_\sigma = \emptyset$, and we do it by induction on i.
Base case: Since $\mathsf{L}^0_{\sigma'}(x) = \rho'(x) \notin \mathcal{L}_\sigma$, we have $\mathsf{L}^0_{\sigma'}(x) \cap \mathcal{L}_\sigma = \emptyset$.
Inductive step: Suppose that $\mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma = \emptyset$, and let us prove that $\mathsf{L}^{n+1}_{\sigma'}(x) \cap \mathcal{L}_\sigma = \emptyset$. It is worth noting that $\mathsf{L}^n_{\sigma'}(x) \subseteq \mathsf{dom}(\mu')$. By the third condition of Proposition 5.8, we have $\forall \ell \in \mathsf{dom}(\mu') \setminus \mathcal{L}_\sigma.\mathsf{rng}(\mu'(\ell).\phi) \cap \mathcal{L}_\sigma = \emptyset$. Therefore, $\mathsf{L}^n_{\sigma'}(x) \subseteq \mathsf{dom}(\mu') \setminus \mathcal{L}_\sigma$, since $\mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma = \emptyset$ and $\mathsf{L}^n_{\sigma'}(x) \subseteq \mathsf{dom}(\mu')$. This entails that $\forall \ell \in \mathsf{L}^n_{\sigma'}(x).\mathsf{rng}(\mu'(\ell).\phi) \cap \mathcal{L}_\sigma = \emptyset$, which implies

$$\bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma = \emptyset. \tag{5.1}$$

We have:

$$\begin{aligned}
\mathsf{L}^{n+1}_{\sigma'}(x) \cap \mathcal{L}_\sigma &= (\mathsf{L}^n_{\sigma'}(x) \cup \bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L})) \cap \mathcal{L}_\sigma && \text{[By Definition 5.3]} \\
&= (\mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma) \cup (\textstyle\bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma) && \text{[By Distributivity]} \\
&= \emptyset \cup \emptyset = \emptyset && \text{[By hyp. and (5.1)]}
\end{aligned}$$

■

The following lemma states that the set of locations reachable from the variables not sharing with any actual parameter of a method before its execution, cannot be affected by that execution.

Lemma 5.10. *Under the hypotheses of Proposition 5.8, let $x \in \mathsf{dom}(\tau) \cap \mathsf{dom}(\tau')$ be a variable such that $\rho(x) = \rho'(x) \in \mathcal{L}_\sigma$ and $\mathsf{L}_\sigma(x) \subseteq \mathcal{L}_\sigma$. Then $\mathsf{L}_\sigma(x) = \mathsf{L}_{\sigma'}(x)$.*

Proof. We prove that, $\forall i \in \mathbb{N}.\mathsf{L}^i_\sigma(x) = \mathsf{L}^i_{\sigma'}(x)$, and we do it by induction on i.
Base case: $\mathsf{L}^0_\sigma(x) = \rho(x) = \rho'(x) = \mathsf{L}^0_{\sigma'}(x)$.
Inductive step: Suppose that $\mathsf{L}^n_\sigma(x) = \mathsf{L}^n_{\sigma'}(x)$ and let us prove that $\mathsf{L}^{n+1}_\sigma(x) = \mathsf{L}^{n+1}_{\sigma'}(x)$. By the first condition of Proposition 5.8, we have that $\forall \ell \in \mathcal{L}_\sigma.\mu(\ell) = \mu'(\ell)$, and therefore $\forall \ell \in \mathsf{L}^n_{\sigma'}(x) = \mathsf{L}^n_\sigma(x) \subseteq \mathsf{L}_\sigma(x) \subseteq \mathcal{L}_\sigma.\mu(\ell) = \mu'(\ell)$, which entails

$$\bigcup_{\ell \in \mathsf{L}^n_\sigma(x)} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) = \bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}). \tag{5.2}$$

We have:

$$\begin{aligned}
\mathsf{L}^{n+1}_{\sigma'}(x) &= \mathsf{L}^{n}_{\sigma'}(x) \cup \bigcup_{\ell \in \mathsf{L}^{n}_{\sigma'}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) && \text{[By Definition 5.3]}\\
&= \mathsf{L}^{n}_{\sigma}(x) \cup \bigcup_{\ell \in \mathsf{L}^{n}_{\sigma}(x)} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) && \text{[By hypothesis and (5.2)]}\\
&= \mathsf{L}^{n+1}_{\sigma}(x) && \text{[By Definition 5.3]}
\end{aligned}$$

■

The following lemma shows that if a variable is bound to the same location before and after a method is executed, then that location reaches a location in $\mathcal{L}_\sigma$ before the method is executed if and only if it reaches a location in $\mathcal{L}_\sigma$ after the mehod is executed.

Lemma 5.11. *Under the hypotheses of Proposition 5.8, for any variable* $x \in \mathsf{dom}(\tau) \cap \mathsf{dom}(\tau')$ *such that* $\rho(x) = \rho'(x)$*, it holds that* $\mathsf{L}_\sigma(x) \cap \mathcal{L}_\sigma = \mathsf{L}_{\sigma'}(x) \cap \mathcal{L}_\sigma$.

Proof. We prove that for any $i \in \mathbb{N}$, $\mathsf{L}^i_\sigma(x) \cap \mathcal{L}_\sigma = \mathsf{L}^i_{\sigma'}(x) \cap \mathcal{L}_\sigma$ and we do it by induction on i.
Base case: $\mathsf{L}^0_\sigma(x) \cap \mathcal{L}_\sigma = \{\rho(x)\} \cap \mathcal{L}_\sigma = \{\rho'(x)\} \cap \mathcal{L}_\sigma = \mathsf{L}^0_{\sigma'}(x) \cap \mathcal{L}_\sigma$.
Inductive step: Suppose that $\mathsf{L}^n_\sigma(x) \cap \mathcal{L}_\sigma = \mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma$ and let us prove that $\mathsf{L}^{n+1}_\sigma(x) \cap \mathcal{L}_\sigma = \mathsf{L}^{n+1}_{\sigma'}(x) \cap \mathcal{L}_\sigma$. Consider a location $\ell \in \mathsf{L}^n_{\sigma_1}(x)$, where $\sigma_1 \in \{\sigma, \sigma'\}$. If $\ell \notin \mathcal{L}_\sigma$, then there exists an actual parameter p of method m (Proposition 5.8) such that ℓ is reachable from p in σ_1. In that case, all the locations reachable from ℓ are reachable from p in σ_1 as well, i.e.

$$\forall \ell \in \mathsf{L}^n_{\sigma_1}(x) \setminus \mathcal{L}_\sigma . \mathsf{rng}(\mu_1(\ell).\phi) \cap \mathcal{L}_\sigma = \varnothing. \tag{5.3}$$

Consider now a location $\ell \in \mathsf{L}^n_\sigma(x) \cap \mathcal{L}_\sigma = \mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma$. In this case, since $\ell \in \mathcal{L}_\sigma$, by the first condition of Proposition 5.8, $\mu(\ell) = \mu'(\ell)$, which entails

$$\bigcup_{\ell \in \mathsf{L}^n_\sigma(x) \cap \mathcal{L}_\sigma} ((\mathsf{rng}(\mu(\ell)) \cap \mathbb{L}) \cap \mathcal{L}_\sigma) = \bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma} ((\mathsf{rng}(\mu'(\ell)) \cap \mathbb{L}) \cap \mathcal{L}_\sigma). \tag{5.4}$$

We have

$$\begin{aligned}
\mathsf{L}^{n+1}_{\sigma'}(x) \cap \mathcal{L}_\sigma &= (\mathsf{L}^n_{\sigma'}(x) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L})) \cap \mathcal{L}_\sigma && \text{[By Definition 5.3]}\\
&= (\mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x)} ((\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma) && \text{[By Distributivity]}\\
&= (\mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma} ((\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma) && \text{[By (5.3)]}\\
&= (\mathsf{L}^n_{\sigma}(x) \cap \mathcal{L}_\sigma) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^n_{\sigma}(x) \cap \mathcal{L}_\sigma} ((\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma) && \text{[By hyp. and (5.4)]}\\
&= (\mathsf{L}^n_{\sigma}(x) \cap \mathcal{L}_\sigma) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^n_{\sigma}(x)} ((\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma) && \text{[By (5.3)]}\\
&= (\mathsf{L}^n_{\sigma}(x) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^n_{\sigma}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L})) \cap \mathcal{L}_\sigma && \text{[By Distributivity]}\\
&= \mathsf{L}^{n+1}_{\sigma}(x) \cap \mathcal{L}_\sigma && \text{[By Definition 5.3]}
\end{aligned}$$

■

We also introduce a static notion of reachability between types. The intuition is that a type t reaches a type t' whenever a variable of static (declared) type t might reach, in some state, another variable of static type t'. In this sense, as we will prove later (Lemma 5.17), this is a weaker, conservative approximation of the dynamic notion of reachability of Definition 5.4: if there exists a state σ where variable a reaches variable b, then the static type of a must reach the static type of b, but the converse does not hold in general.

Definition 5.12 (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:*

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

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

Let us now show some important results regarding type reachability. Namely, we show that if a type reaches another type, then the former also reaches all possible supertypes of the latter (Lemma 5.13) and that the set of reachable types of a type t is included in the set of reachable types of all t's supertypes (Lemmas 5.14 and 5.15). These lemmas are used for the proofs of Lemmas 5.35 and 5.40.

Lemma 5.13. *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 $i \geq 0$, if $\mathsf{t}' \in \mathsf{T}^i(\mathsf{t})$ 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 5.12 we have $\mathsf{t}' \in \mathsf{compatible}(\mathsf{t})$ and by Lemma 3.5 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 $\mathsf{t}'' \in \mathsf{T}^{i-1}(\mathsf{t})$. Since $\mathsf{t}' \in \mathsf{T}^i(\mathsf{t})$, by Definition 5.12 we have two cases:

- if $\mathsf{t}' \in \mathsf{T}^{i-1}(\mathsf{t})$ then, by inductive hypothesis, also $\mathsf{t}'' \in \mathsf{T}^{i-1}(\mathsf{t})$ which entails $\mathsf{t}'' \in \mathsf{T}^i(\mathsf{t})$;
- if $\mathsf{t}' \notin \mathsf{T}^{i-1}(\mathsf{t})$ then, by Definition 5.12, $\mathsf{t}' \in \mathsf{compatible}(\mathsf{t}_1)$, where
  - there exists $\kappa \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{K}$ and $\kappa'.f:\mathsf{t}_1 \in \mathsf{F}(\kappa)$ or
  - there exists $\mathsf{t}_1[\,] \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{A}$.

  In both cases, by Definition 5.12, $\mathsf{compatible}(\mathsf{t}_1) \subseteq \mathsf{T}^i(\mathsf{t})$. Since $\mathsf{t}' \leq \mathsf{t}''$, by Lemma 3.5 we have $\mathsf{t}'' \in \mathsf{compatible}(\mathsf{t}_1)$ and hence $\mathsf{t}'' \in \mathsf{T}^i(\mathsf{t})$.

■

Lemma 5.14. *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 follows by induction on i and Definition 5.12. ■

Lemma 5.15. *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 $i \geq 0$, $\mathsf{T}^i(\mathsf{t}) \subseteq \mathsf{T}^i(\mathsf{t}')$, by induction over i. If $i = 0$ the thesis follows by Lemma 3.6. Assume hence that $\mathsf{T}^{i-1}(\mathsf{t}) \subseteq \mathsf{T}^{i-1}(\mathsf{t}')$, for $i > 0$. Then,

$$\begin{aligned} \mathsf{T}^i(\mathsf{t}) &= \mathsf{T}^{i-1}(\mathsf{t}) \cup \bigcup_{\substack{\kappa \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{K} \\ \kappa'.f:\mathsf{t}'' \in \mathsf{F}(\kappa)}} \mathsf{compatible}(\mathsf{t}'') \cup \bigcup_{\mathsf{t}''[\,] \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{A}} \mathsf{compatible}(\mathsf{t}'') && \text{[By Def. 5.12]} \\ &\subseteq \mathsf{T}^{i-1}(\mathsf{t}') \cup \bigcup_{\substack{\kappa \in \mathsf{T}^{i-1}(\mathsf{t}') \cap \mathbb{K} \\ \kappa'.f:\mathsf{t}'' \in \mathsf{F}(\kappa)}} \mathsf{compatible}(\mathsf{t}'') \cup \bigcup_{\mathsf{t}''[\,] \in \mathsf{T}^{i-1}(\mathsf{t}') \cap \mathbb{A}} \mathsf{compatible}(\mathsf{t}'') && \text{[By hypothesis]} \\ &= \mathsf{T}^i(\mathsf{t}') && \text{[By Def. 5.12]} \end{aligned}$$

■

$\mathsf{T}^0(\texttt{Object})$	$= \mathsf{T}(\texttt{Object})$
	$= \{\texttt{Object}, \texttt{Student}, \texttt{List}\}$
$\mathsf{T}^0(\texttt{Student})$	$= \{\texttt{Object}, \texttt{Student}\}$
$\mathsf{T}^1(\texttt{Student})$	$= \mathsf{T}(\texttt{Student})$
	$= \{\texttt{int}, \texttt{Object}, \texttt{Student}\}$
$\mathsf{T}^0(\texttt{List})$	$= \{\texttt{List}, \texttt{Object}\}$
$\mathsf{T}^1(\texttt{List})$	$= \{\texttt{List}, \texttt{Object}, \texttt{Student}\}$
$\mathsf{T}^2(\texttt{List})$	$= \mathsf{T}(\texttt{List})$
	$= \{\texttt{int}, \texttt{List}, \texttt{Object}, \texttt{Student}\}$
$\mathsf{T}^0(\texttt{Student[ ]})$	$= \{\texttt{Object[ ]}, \texttt{Student[ ]}\}$
$\mathsf{T}^1(\texttt{Student[ ]})$	$= \{\texttt{Object[ ]}, \texttt{Student[ ]}, \texttt{Object}, \texttt{Student}\}$
$\mathsf{T}^2(\texttt{Student[ ]})$	$= \mathsf{T}(\texttt{Student[ ]})$
	$= \{\texttt{int}, \texttt{Object[ ]}, \texttt{Student[ ]}, \texttt{Object}, \texttt{Student}\}$

Fig. 5.2. Example of computation of reachable locations and types

Example 5.16. Consider class `List` from Fig. 3.2 and suppose that the class `Student` contains only one field, of type `int`, like stated in Example 3.16. Both `List` and `Student` are subclasses of `Object`. In Fig. 5.2 we show the types reachable from each of these three classes, as well as the types reachable from the array type `Student[ ]`. For instance, $\texttt{List} \rightsquigarrow \texttt{Student}$, $\texttt{Object} \rightsquigarrow \texttt{Student}$, $\texttt{Student} \rightsquigarrow \texttt{Object}$, $\texttt{Object} \rightsquigarrow \texttt{Student}$, etc. □

The following lemma shows a very important result. Namely, it illustrates the relationship between variable and type reachability: 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 5.17. *Let $\tau \in \mathcal{T}$, $\sigma \in \Sigma_\tau$ and $a, b \in \mathsf{dom}(\tau)$. If $a \rightsquigarrow^\sigma b$, then $\tau(a) \rightsquigarrow \tau(b)$.*

Proof. By letting $\sigma = \langle \rho, \mu \rangle$, from $a \rightsquigarrow^\sigma b$ and Definition 5.4 we have $\rho(a), \rho(b) \in \mathbb{L}$. We prove that for every $i \geq 0$, 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 $\mu(\ell).\texttt{type} \in \mathsf{T}^j(\tau(a))$. This entails our thesis. Namely, since $a \rightsquigarrow^\sigma b$, there exists $i \geq 0$ such that $\rho(b) \in \mathsf{L}^i_\mu(\rho(a)) \subseteq \mathsf{L}_\sigma(a)$, and $P(i)$ ensures that there also exists $0 \leq j \leq i$ such that $\mu\rho(b).\texttt{type} \in \mathsf{T}^j(\tau(a)) \subseteq \mathsf{T}(\tau(a))$, i.e., $\tau(a) \rightsquigarrow \mu\rho(b).\texttt{type}$. Since (Definition 3.13) $\mu\rho(b).\texttt{type} \leq \tau(b)$, by Lemma 5.13 we conclude that $\tau(a) \rightsquigarrow \tau(b)$.

Let us now prove that, for every $i \geq 0$, $P(i)$ holds.

Base case: $i = 0$. Since $a \rightsquigarrow^\sigma b$, we have $\rho(a) \in \mathbb{L}$ and therefore $\mathsf{L}^0_\sigma(a) = \{\rho(a)\}$. By Definition 3.13, $\mu\rho(a).\texttt{type} \leq \tau(a)$ i.e., $\mu\rho(a).\texttt{type} \in \mathsf{compatible}(\tau(a)) = \mathsf{T}^0(\tau(a))$. Since $j = 0 \leq 0 = i$, $P(0)$ holds.

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 5.3 we have two cases:

- if $\ell \in \mathsf{L}^{i-1}_\mu(\rho(a))$ then, by inductive hypothesis, ($P(i-1)$ holds) we know that there exists $0 \leq j \leq i-1 < i$ such that $\mu(\ell).\texttt{type} \in \mathsf{T}^j(\tau(a))$. Therefore, $P(i)$ holds.
- if $\ell \notin \mathsf{L}^{i-1}_\mu(\rho(a))$ then $\ell \in \mathsf{rng}(\mu(\ell').\phi) \cap \mathbb{L}$ for some $\ell' \in \mathsf{L}^{i-1}_\mu(\rho(a))$. We distinguish the following cases:
  - if $\mu(\ell').\texttt{type} \in \mathbb{K}$, then there exists $\kappa'.f : \mathsf{t}' \in \mathsf{F}(\mu(\ell').\texttt{type})$ such that $\ell = (\mu(\ell').\phi)(\kappa'.f : \mathsf{t}')$ and $\mu(\ell).\texttt{type} \leq \mathsf{t}'$. Hence, $\mu(\ell).\texttt{type} \in \mathsf{compatible}(()\mathsf{t}') \subseteq \mathsf{T}^{j+1}(\tau(a))$;

– if $\mu(\ell').\texttt{type} \in \mathbb{A}$, then there exists $\kappa'.f : \mathsf{t}' \in \mathsf{F}(\mu(\ell').\texttt{type})$ such that $\ell = (\mu(\ell').\phi)(\kappa'.f : \mathsf{t}')$ and $\mu(\ell).\texttt{type} \leq \mathsf{t}'$. Hence, $\mu(\ell).\texttt{type} \in \mathsf{compatible}(()\mathsf{t}') \subseteq \mathsf{T}^{j+1}(\tau(a))$;

Hence, in both cases, $\mu(\ell).\texttt{type} \in \mathsf{compatible}(()\mathsf{t}') \subseteq \mathsf{T}^{j+1}(\tau(a))$, and since $0 \leq j+1 \leq i$, $P(i)$ holds as well.

■

Example 5.18. Since $l_4 \rightsquigarrow^{\sigma} l_3$ (Ex. 5.6), by Lemma 5.17, also $\tau(l_4) \rightsquigarrow \tau(l_3)$ holds. In fact, Ex. 5.16 shows that $\tau(l_4) = \texttt{List} \rightsquigarrow \texttt{Student} = \tau(l_3)$. □

5.3 Definition of the Possible Reachability Analysis

The goal of this section is to define a static analysis that computes, for each program point, an over-approximation of the reachability information available at that point, i.e., which variables might reach the other ones. We show how it is possible to instantiate the parameters of the general parameterized framework introduced in previous chapter in order to obtain the desired static analysis. There are two essential things a designer should do in order to define such a static analysis:

1. Mathematically encode the property of interest, define the abstract domain and show how it can be related to the concrete one (Section 4.3);
2. Define a propagation rule for every possible arc available in the ACG, i.e., define an abstract semantics of our target language which simulates the behavior of concrete bytecode instructions with respect to the abstract domain defined above.

Subsections 5.3.1 and 5.3.2 deal with the points 1. and 2. respectively.

5.3.1 Abstract Domain Reach

The first goal of this section is to mathematically encode the property of interest. In Section 5.2 we introduced the concrete property of interest which strictly depends on the current state of the program. We want to determine that property statically, and the most natural way for representing the fact that for an execution of a program a variable v might reach a variable w is by using the ordered pair $\langle v, w \rangle$. We followed this idea and formally defined the abstract domain Reach.

Definition 5.19 (Concrete and Abstract Domain). *The concrete and abstract domains over $\tau \in \mathcal{T}$ are $\mathsf{C}_\tau = \langle \wp(\Sigma_\tau), \subseteq, \cup, \cap, \Sigma_\tau, \varnothing \rangle$ and $\textsc{Reach}_\tau = \langle \mathcal{A}_\tau, \subseteq, \cup, \cap, \mathcal{A}_\tau, \emptyset \rangle$, where $\mathcal{A}_\tau = \wp(\mathsf{dom}(\tau) \times \mathsf{dom}(\tau))$ is the powerset of the set of ordered pairs of variables. For every $v, w \in \mathsf{dom}(\tau)$, we write $v \rightsquigarrow w$ to denote the ordered pair $\langle v, w \rangle$.*

An abstract domain element $R \in \textsc{Reach}_\tau$ represents those concrete states in Σ_τ whose reachability information is conservatively over-approximated by the pairs of variables in R. Namely, for each ordered pairs of variables $a \rightsquigarrow b \in R$, the concretization of R must contain *at least* those concrete states in which a reaches b. The following definition formalizes this intuition.

Definition 5.20 (Concretization map). *For every type environment $\tau \in \mathcal{T}$, we define the* concretization map $\gamma_\tau : \textsc{Reach}_\tau \rightarrow \mathsf{C}_\tau$ *as:*

$$\gamma_\tau = \lambda R.\{\sigma \in \Sigma_\tau \mid \forall a, b \in \mathsf{dom}(\tau). a \rightsquigarrow^\sigma b \Rightarrow a \rightsquigarrow b \in R\}.$$

Example 5.21. Consider the state σ introduced in Example 3.16, and the concrete reachability information that σ gives rise to: $l_1 \rightsquigarrow^\sigma l_1$, $l_1 \rightsquigarrow^\sigma l_2$, $l_3 \rightsquigarrow^\sigma l_3$, $l_4 \rightsquigarrow^\sigma l_1$, $l_4 \rightsquigarrow^\sigma l_2$, $l_4 \rightsquigarrow^\sigma l_3$, $l_4 \rightsquigarrow^\sigma l_4$, $l_5 \rightsquigarrow^\sigma l_1$, $l_5 \rightsquigarrow^\sigma l_3$, $l_5 \rightsquigarrow^\sigma l_5$ (determined in Example 5.6). Then σ can be soundly approximated only by the sets of pairs of variables that include at least the following ordered pairs of variables: $R_1 = \{l_1 \rightsquigarrow l_1, l_1 \rightsquigarrow l_2, l_3 \rightsquigarrow l_3, l_4 \rightsquigarrow l_1, l_4 \rightsquigarrow l_2, l_4 \rightsquigarrow l_3, l_4 \rightsquigarrow l_4, l_5 \rightsquigarrow l_1, l_5 \rightsquigarrow l_3, l_5 \rightsquigarrow l_5\}$, i.e., for every $R \supseteq R_1$, $\sigma \in \gamma_\tau(R)$. □

Requirements 4.1 and 4.2 deal with the abstract domain representing the property of interest. We will show in Section 5.4 that Reach actually satisfies these requirements.

5.3.2 Propagation Rules

In Chapter 4 we defined the notion of abstract constraint graph, ACG, and we showed how these graphs can be constructed from the text of the program under analysis. We recall that an ACG is composed of the set of nodes, corresponding to different program bytecode instructions, and of the set of arcs which connect those nodes. Each node of an ACG created for the reachability analysis of program variables is enriched with an element of the abstract domain Reach. That abstract element represents an over-approximation of the actual reachability information available at that point. On the other hand, each arc of that ACG is enriched with a propagation rule showing how the abstract elements (i.e., approximations) available at arc's sources are propagated to its sink. In Section 4.4 we specified the requirements that these propagation rules have to satisfy in order to guarantee the soundness of the overall analysis, but we did not give any concrete definition of any propagation rule, since they strictly depend on the property which is being analyzed, while we were dealing with a generic property in that chapter. On the contrary, in this chapter, we are interested in one particular property, i.e., reachability, we have shown how it can be mathematically represented in our framework, and in this subsection we show how we propagate the abstract elements approximating that property.

In the following we assume the presence of *possible sharing* and *definite aliasing* approximations. Namely, we suppose that at each program point, there exist a set of (non-ordered) pairs of variables representing an over-approximation of the actual sharing information at that program point, and a set of (non-ordered) pairs of variables representing an under-approximation of the actual aliasing information at that point. Pairs of variables not belonging to the former, definitely do not share at that point, i.e., there is no execution of the program under analysis in which these variables reach a common location. On the contrary, pairs of variables that belong to the latter, are definitely aliased at that point, i.e., for any execution of the program under analysis, these variables point to the same location, but there might also be other pairs of aliased variables. These pieces of information can be computed statically, and our tool Julia is able to provide them [63, 82]. Our analysis works correctly even when these two approximations are not available: we can always assume that at each program point every variable a can share with any other variable available at that point, and that there is no variable definitely aliased to a. In that case the reachability information we determine would be less precise, but still sound.

	ins	propagation rule
#1	load k t	$\lambda R.R \cup R[l_k/s_j] \cup \{l_k \rightsquigarrow s_j, s_j \rightsquigarrow l_k \mid l_k \rightsquigarrow l_k \in R\}$
#2	store k t	$\lambda R.\{(a \rightsquigarrow b)[s_{j-1}/l_k] \mid a \rightsquigarrow b \in R \wedge a, b \neq l_k\}$
#3	new κ	$\lambda R.R \cup \{s_j \rightsquigarrow s_j\}$
#4	getfield f	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \neq s_{j-1}\} \cup \{s_{j-1} \rightsquigarrow b \in R \mid \mathsf{t} \rightsquigarrow \tau(b)\} \cup$ $\{a \rightsquigarrow s_{j-1} \mid \tau(a) \rightsquigarrow \mathsf{t} \neq \texttt{int} \wedge [a \text{ and } s_{j-1} \text{ might share at ins}]\}$
#5	putfield f	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_{j-1}, s_{j-2}\}\} \cup$ $\{a \rightsquigarrow b \mid a, b \notin \{s_{j-1}, s_{j-2}\} \wedge a \rightsquigarrow s_{j-2} \in R \wedge s_{j-1} \rightsquigarrow b \in R\}$
#6	arraynew α	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \neq s_{j-1}\} \cup \{s_{j-1} \rightsquigarrow s_{j-1}\}$
#7	arraylength α	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \neq s_{j-1}\}$
#8	arrayload t[ ]	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_{j-1}, s_{j-2}\}\} \cup \{s_{j-2} \rightsquigarrow b \in R \mid \mathsf{t} \rightsquigarrow \tau(b)\} \cup$ $\{a \rightsquigarrow s_{j-2} \mid \tau(a) \rightsquigarrow \mathsf{t} \neq \texttt{int} \wedge [a \text{ and } s_{j-2} \text{ might share at ins}]\}$
#9	arraystore t[ ]	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_{j-1}, s_{j-2}, s_{j-3}\}\} \cup$ $\{a \rightsquigarrow b \mid a, b \notin \{s_{j-1}, s_{j-2}, s_{j-3}\} \wedge a \rightsquigarrow s_{j-3} \in R \wedge s_{j-1} \rightsquigarrow b \in R\}$
#10	dup t	$\lambda R.R \cup R[s_{j-1}/s_j] \cup \{s_{j-1} \rightsquigarrow s_j, s_j \rightsquigarrow s_{j-1} \mid s_{j-1} \rightsquigarrow s_{j-1} \in R\}$
#11	const x, ifne t, ifeq t, add, sub, mul, div, rem, inc k x, catch, exception_is K	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$
#12	return void	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_0, \ldots, s_{j-1}\}\}$
#13	return t	$\lambda R.\{(a \rightsquigarrow b)[s_{j-1}/s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\}$
#14	throw κ	$\lambda R.\{(a \rightsquigarrow b)[s_{j-1}/s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\}$ $\cup \{s_0 \rightsquigarrow s_0\}$
#15	throw κ	$\lambda R.\{(a \rightsquigarrow b)[s_{j-1}/s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\}$ $\cup \{s_0 \rightsquigarrow s_0\}$
#16	call $m_1 \ldots m_k$	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_0, \ldots, s_{j-1}\}\} \cup \{s_0 \rightsquigarrow s_0\}$ $\cup \{a \rightsquigarrow s_0 \mid a \in \{l_0, \ldots, l_{i-1}\} \wedge \tau(a) \rightsquigarrow \texttt{Throwable}\}$ $\cup \{s_0 \rightsquigarrow a \mid a \in \{l_0, \ldots, l_{i-1}\} \wedge \texttt{Throwable} \rightsquigarrow \tau(a)\}$
#17	div, rem, new κ, getfield f, putfield f, arraynew α, arraylength α, arrayload α, arraystore α	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_0, \ldots, s_{j-1}\}\} \cup \{s_0 \rightsquigarrow s_0\}$
#18	call $m_1 \ldots m_k$	$\lambda R.\left\{(a \rightsquigarrow b)\begin{bmatrix} s_{j-\pi}/l_0 \\ \cdots \\ s_{j-1}/l_{\pi-1} \end{bmatrix} \middle\| a \rightsquigarrow b \in R \wedge a, b \in \{s_{j-\pi}, \ldots, s_{j-1}\}\right\}$

Fig. 5.3. Propagation rules of simple arcs

Definition 5.22 (Propagation rules). *We distinguish between simple (*1−1*) arcs, having one source and one sink node, and multi (*2−1*) arcs, which have two source and one sink node. In Fig. 5.3 we report the propagation rules for the reachability analysis of simple* ACG *arcs: propagation rules* #1-#11 *deal with the sequential arcs, propagation rules* #12-#14 *deal with the final arcs, propagation rules* #15-#17 *deal with the exceptional arcs, while propagation rule* #18 *deals with the parameter passing arcs of the* ACG.

The propagation rules for multi arcs are shown in Fig. 5.4: propagation rules #19 *and* #20 *specify how the return value arcs and the side-effects arcs propagate the reachability approximations available at their sources to the approximation of their sink node.*

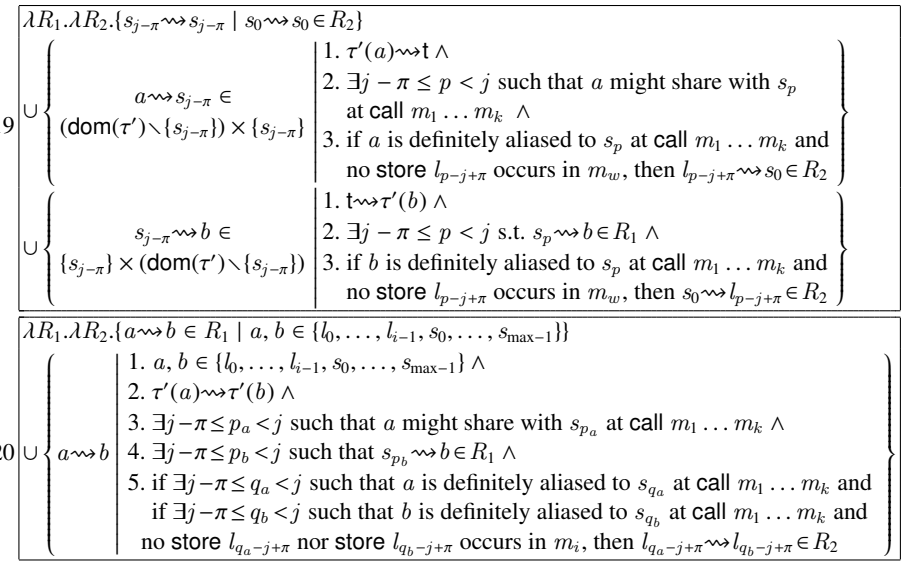


Fig. 5.4. Propagation rules of multi-arcs

Definitions of our propagation rules deserve some explanations. They specify how the ACG of the program under analysis propagates approximations available at each its node. We start with an example illustrating the construction of an ACG (Example 5.23), and then we explain in more detail the propagation rules defined above.

Example 5.23. Fig. 5.5 shows the ACG of the constructor from Fig. 3.3. It also shows, in grey, three nodes of a caller of this constructor (nodes A, B and C) and two nodes of the callee of call `java.lang.Object.⟨init⟩`(): void, to exemplify the arcs related to the method's invocation and its normal (`end`) and exceptional (`exception`) end. Arcs are decorated with the number of their associated propagation rule. Note that the graph for the whole program includes other nodes and arcs. Fig. 5.5 only shows the portion that is relevant for our example.

In the following examples, for each node n, we let i_n and j_n be the number of local and operand stack variables at n respectively and τ_n be the static type information available at that node. We suppose that $i_A = 4$ and $j_A = 4$, and that variables l_1, l_2 and l_3 correspond to variables $list$, i and st from Fig. 3.2, respectively. We assume that previous static analyses provided a correct possible sharing information at node A: $\mathsf{share}_A = \{\langle s_0, s_1\rangle, \langle l_3, s_2\rangle, \langle l_1, s_3\rangle\}$ (only these non-ordered pairs of variables *might possibly share*) and a correct definite aliasing information at node A: $\mathsf{alias}_A = \{\langle s_0, s_1\rangle, \langle l_3, s_2\rangle\}$ (these non-ordered pairs of variables *must be aliased*, but also other pairs of variables might be aliased). Moreover, we suppose that our reachability analysis performed until node A provided the following approximation of the actual reachability information at that point:

$$R_A = \left\{ \begin{array}{c} l_0 \rightsquigarrow l_0,\ l_1 \rightsquigarrow l_1,\ l_3 \rightsquigarrow l_3,\ l_1 \rightsquigarrow s_3,\ l_3 \rightsquigarrow s_2,\ s_2 \rightsquigarrow l_3, \\ s_0 \rightsquigarrow s_0,\ s_0 \rightsquigarrow s_1,\ s_1 \rightsquigarrow s_0,\ s_1 \rightsquigarrow s_1,\ s_2 \rightsquigarrow s_2,\ s_3 \rightsquigarrow s_3 \end{array} \right\}. \tag{5.5}$$

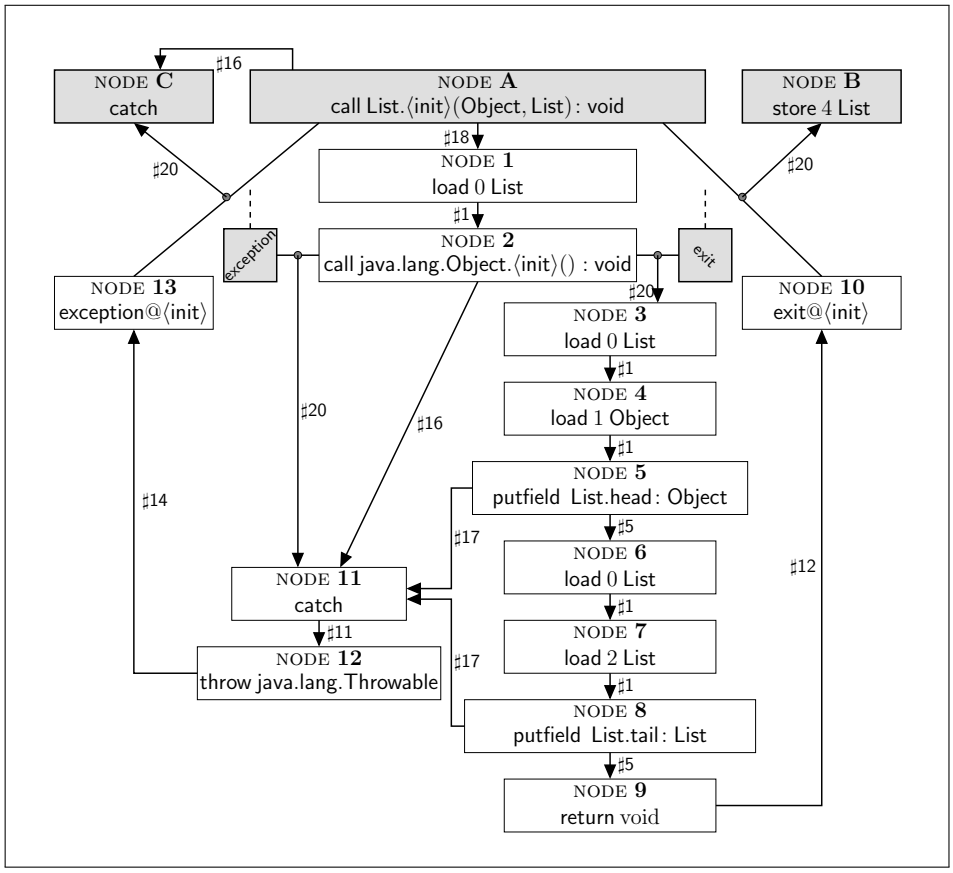


Fig. 5.5. The ACG for the method ⟨`init`⟩ in Fig. 3.3

Constructor *con* = `List`.⟨init⟩(`Object`, `List`): void is invoked at the program point corresponding to node A. The receiver of *con*, s_1, is definitely aliased to s_0, while its first actual argument, s_2, is definitely aliased to l_3 (since $\langle s_0, s_1\rangle, \langle l_3, s_2\rangle \in \mathsf{alias}_A$). Since *con* creates a new object of class `List` and instantiate its fields `head` and `tail` with the values contained in its arguments, it is clear that after that object is created, s_0 (aliased to the new created object) reaches l_3 (aliased to the variable whose value is written inside the field `head`), and we expect our reachability analysis to include the pair $s_0 \rightsquigarrow l_3$ in the approximation at node B. □

Let us now explain, in more detail, the propagation rules introduced in Figures 5.3 and 5.4. They *simulate* the behavior of the concrete semantics of bytecode instructions given in Fig. 3.6.

The **sequential arcs** link an instruction to its immediate successors. We suppose that the approximation available before a bytecode instruction is executed is R and we discuss how it is propagated by the sequential arcs' propagation rules:

load k t - In this case a new variable (s_j) is pushed onto the operand stack and its value is equal to that of l_k. Therefore, we propagate R by keeping all the reachability pairs

already present in R and by using the fact that everything that might reach (or might be reachable from) l_k in R, might also reach (or might be reachable from) s_j in the final approximation (i.e., $R[l_k/s_j]$). Moreover, l_k and s_j contain the same value, and if l_k might reach itself in R (i.e., if $\tau(l_k) \neq \texttt{int}$), then also s_j might reach itself in R's propagation.

store k t - In this case the topmost variable is popped from the operand stack (s_{j-1}) and its value is assigned to l_k. Therefore, all the reachability pairs involving l_k in the initial approximation R should be removed from the final one. On the other hand, everything except l_k that might reach (or might be reachable from) s_{j-1} in R, might also reach (or might be reachable from) l_k in the final approximation (i.e., $(a \rightsquigarrow b)[s_{j-1}/l_k]$, where $a \rightsquigarrow b \in R$ and $a, b \neq l_k$).

new κ - In this case a new object is created and the location it is bound to is pushed onto the operand stack, in s_j. Therefore, the initial approximation R is kept, and since objects are not of primitive type, i.e., $\tau(s_j) \neq \texttt{int}$, we should also add the pair $s_j \rightsquigarrow s_j$ to denote that the new created object can reaches itself. It does not reach nor is reachable from anything else, since its fields are initialized to their default values and the location it is bound to is fresh.

getfield f - In this case the location memorized in the topmost operand stack element s_{j-1} is replaced with the value of the field f of the object corresponding to the former. Hence each reachability pair that does not involve s_{j-1} in R should be present in the final approximation too. Additionally, we must consider all those variables b which might be reachable from the field f (i.e., such that $s_{j-1} \rightsquigarrow b$) and all those variables a which might reach the field f (i.e., such that $a \rightsquigarrow s_{j-1}$) in the final approximation. In the former case, we observe that if the field reaches b, then also its containing object (i.e., the old top of the operand stack) had to reach b in the initial approximation: $s_{j-1} \rightsquigarrow b \in R$. In order to improve the precision we consider only those pairs of variables that satisfy the type reachability requirement: $\mathsf{t} \rightsquigarrow \tau(b)$. In the latter case, we rely on a pessimistic (but conservative) assumption: every variable a might reach the field in the final approximation, as long as the field has a reference type reaching the static type of a: $\tau(a) \rightsquigarrow \mathsf{t} \neq \texttt{int}$. We can, though, improve the precision of this rule by considering only those variables a which, beside the previous condition, might also share with the receiver s_{j-1} in R.

putfield f - In this case the value memorized in the topmost operand stack element s_{j-1} is written in the field f of the object corresponding to the location memorized in the second topmost operand stack element s_{j-2}, and both s_{j-1} and s_{j-2} are popped from the operand stack. Hence, the corresponding propagation rule keeps a reachability pair available in R if it involves neither s_{j-1} nor s_{j-2}. Some additional pairs are added to the final approximation though: a variable a might reach a variable b there if a reaches the receiver s_{j-2} ($a \rightsquigarrow s_{j-2}$) in R and the value s_{j-1} reaches b ($s_{j-1} \rightsquigarrow b$) in R.

arraynew α - In this case the topmost operand stack element containing an integer value is replaced with the fresh location bound to the new created array. The propagation is similar to the case of new κ.

arraylength α - In this case the topmost operand stack element containing a reference to an array is replaced with the length of that array, which is of integer type, and therefore cannot be reachable from anything and cannot reach anything.

arrayload α - In this case the k-th element of the array corresponding to the location memorized in the second topmost operand stack element s_{j-2}, where k is the topmost

operand stack element, is written onto the top of the stack. Previously, both s_{j-1} and s_{j-2} are popped from the stack. The propagation rule and its explanation are analogous to the case of getfield f.

arraystore α - In this case the value memorized in the topmost operand stack element s_{j-1} is written in the k-th element of the array corresponding to the location memorized in the third topmost operand stack element s_{j-3}, where k is the integer value memorized in the second topmost operand stack element. All s_{j-1}, s_{j-2} and s_{j-3} are popped from the operand stack. The propagation rule and its explanation are analogous to the case of putfield f.

dup t - In this case a new variable s_j is pushed onto the operand stack, and it is associated the location memorized in s_{j-1}. Since s_j is aliased to s_{j-1}, it is clear that every variable that might reach (or might be reachable from) s_{j-1} in R, might also reach (or might be reachable from) s_j in the final approximation: (i.e., $R[s_{j-1}/s_j]$). Moreover, if s_{j-1} reaches itself in R then, in the final approximation, it should also reach s_j and vice versa.

otherwise - Bytecode instructions const x, add, sub, mul, div, rem, inc k x deal with the values of primitive type (or `null`), and therefore do not introduce or remove any reachability. On the other hand, catch and exception_is K do not modify the initial state, and therefore do not change the reachability information available at that point, while ifne t and ifeq t just pop the topmost operand stack element, and therefore do not modify the reachability information concerning all other variables different from the topmost stack element. In all these cases, we keep the reachability pairs available in R if they are composed of the variables actually available at that point.

Example 5.24. Consider, for instance, nodes 4, 5 and 6 in Fig. 5.5, and suppose that the reachability approximation at node 4 is:

$$R_4 = \{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, s_0 \rightsquigarrow l_0, s_0 \rightsquigarrow s_0\}.$$

From hypotheses assumed in Example 5.23, it can be easily determined that $i_4 = i_5 = 3$, $j_4 = 1$ and $j_5 = 2$. Nodes 4 and 5 are linked by a sequential arc with propagation rule #1, while nodes 5 and 6 are linked by a sequential arc with propagation rule #5. Definition of propagation rule #1 (Fig. 5.3) gives:

$$\Pi^{\#1}(R_4) = R_4 \cup R_4[l_1/s_1] \cup \{l_1 \rightsquigarrow s_1, s_1 \rightsquigarrow l_1\} = \left\{ \begin{array}{c} l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, l_1 \rightsquigarrow s_1, \\ s_0 \rightsquigarrow l_0, s_1 \rightsquigarrow l_1, s_0 \rightsquigarrow s_0, s_1 \rightsquigarrow s_1 \end{array} \right\}.$$

Suppose now that $R_5 = \Pi^{\#1}(R_4)$. Then, by definition of propagation rule #5 (Fig. 5.3) we have:

$$\begin{aligned} \Pi^{\#5}(R_5) = \{a \rightsquigarrow b \in R_5 \mid a, b \notin \{s_0, s_1\}\} \cup \{a \rightsquigarrow b \mid a, b \notin \{s_0, s_1\} \wedge a \rightsquigarrow s_0 \in R_5 \wedge s_1 \rightsquigarrow b \in R_5\} \\ = \{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}. \end{aligned}$$

□

The **final arcs** feed nodes $\boxed{\text{exit@}m}$ and $\boxed{\text{exception@}m}$ for each method or constructor m. The former (respectively latter) contains the reachability information present in all states at a non-exceptional (respectively exceptional) end of m. Hence, $\boxed{\text{exit@}m}$ is the sink of the arcs starting from the bytecode instructions return t present inside m. The

propagation rules state that the operand stack is emptied at the end of execution of a void method m (rule #12) or only one element survives, the returned value (rule #13). Similarly, $\boxed{\text{exception@}m}$ is the sink of the bytecode instructions throw κ with no exception handler in m (i.e., not followed by a catch inside m). Rule #14 states that all elements of the operand stack, except the topmost one, s_{j-1}, disappear. The latter is renamed into the exception object s_0, and is always non-`null` (thus, $s_0 \rightsquigarrow s_0$). 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 two putfield in Fig. 3.3).

Example 5.25. Consider, for instance, nodes 9 and 10 in Fig. 5.5, and suppose that the reachability approximation at node 9 is:

$$R_9 = \{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_0 \rightsquigarrow l_2, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}.$$

Nodes 9 and 10 are linked by a final arc with propagation rule #12. Definition of the latter imposes that $\Pi^{\#12}(R_9)$ contains all the pairs of R_9 containing no operand stack variable, and since $j_9 = 0$ (it can be easily shown using the hypotheses of Example 5.23), we conclude:

$$\Pi^{\#12}(R_9) = \{a \rightsquigarrow b \in R_9 \mid a, b \notin \emptyset\} = \{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_0 \rightsquigarrow l_2, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}.$$

□

The **exceptional arcs** link every instruction that might throw an exception to the catch at the beginning of their exception handler(s). Rules #14 and #15 are identical, but the latter is applied in the case of a throw κ with a successor. Rule #16 states a pessimistic assumption about the exceptional states after a method call: the reachability pairs before the call can survive as long as they do not deal with the operand stack elements. The thrown object s_0 is non-`null` (thus, $s_0 \rightsquigarrow s_0$) and conservatively assumed to reach and to be reached from every local variable a, as long as the static types allow it. We recall that, in Java, `Throwable` is the superclass of all exceptions. Rule #17 deals with all other bytecode instructions that might throw an exception div, rem, new, getfield, putfield, arraynew, arraylength, arrayload, arraystore): it states that, in that case, the operand stack disappears but the reachability among local variables remains unaffected.

Example 5.26. Consider, for instance, nodes 5 and 11 in Fig. 5.5. In Example 5.24 we assumed that:

$$R_5 = \{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, l_1 \rightsquigarrow s_1, s_0 \rightsquigarrow l_0, s_1 \rightsquigarrow l_1, s_0 \rightsquigarrow s_0, s_1 \rightsquigarrow s_1\}.$$

Nodes 5 and 11 are linked by an exceptional arc with propagation rule #17. Definition of the latter imposes that $\Pi^{\#17}(R_5)$ contains all the pairs from R_5 containing no operand stack element enriched with $s_0 \rightsquigarrow s_0$, where s_0 holds the thrown exception, and it is the only operand stack variable available at node 11, i.e.,

$$\Pi^{\#17}(R_5) = \{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, s_0 \rightsquigarrow s_0\}.$$

□

We come now to the arcs that deal with the method call and return mechanism. The **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 a dynamic target of that invocation. Propagation rule #18 simply states that the actual parameters of the call, held in the operand stack variables $s_{j-\pi}, \ldots, s_{j-1}$, are renamed into m_w's formal parameters, i.e., the local variables $l_0, \ldots, l_{\pi-1}$. No other variable exists at the beginning of m_w.

Example 5.27. Consider, for instance, nodes A and 1 in Fig. 5.5. Nodes A and 1 are linked by a parameter passing arc with propagation rule #18. We have $j_A = 4$ and $\pi = 3$ (stack elements s_1, s_2 and s_3 hold the actual parameters of a call to this constructor). Definition of propagation rule #18 gives:

$$\Pi^{\#18}(R_A) = \left\{ (a \rightsquigarrow b) \begin{bmatrix} s_1/l_0 \\ s_2/l_1 \\ s_3/l_2 \end{bmatrix} \middle\| \; a \rightsquigarrow b \in R_A \text{ and } a, b \in \{s_1, s_2, s_3\} \right\} = \{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\},$$

where R_A is given in Equation 5.5. □

There is a **return value multi-arc** for each target m_w of a call. Rule #19 concerns R_1 and R_2, approximations at the node corresponding to the call and at node $\boxed{\text{exit@}\, m_w}$, respectively. It creates the reachability pairs related to the returned value that, after the call, becomes the topmost operand stack element $s_{j-\pi}$. Namely, $s_{j-\pi}$ reaches itself after the call if s_0, its corresponding variable at the end of the callee m_w, reaches itself. But the complex part of this rule deals with the other variables of the caller, since it must be determined whether they reach the return value or can be reached from it. Here, we exploited the observation that a variable of the caller might reach or be reached from the return value only if it shares with an actual parameter of the call. We do need sharing here, since it is well possible that this variable does not reach and is not reachable from any of the actual parameters of the call but yet shares with one of them and is consequently made to reach (or be reachable from) the return value of the call. Moreover, in the frequent case when it is actually aliased to an actual parameter of the call, we exploited the possibility of checking the reachability of the corresponding formal parameter of the callee (to and from the returned value), provided that it is not reassigned inside the callee. Namely, an arbitrary variable a available after the call and different from $s_{j-\pi}$ ($a \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\}$) might reach $s_{j-\pi}$ at that point ($a \rightsquigarrow s_{j-1}$) if the following conditions hold:

1. the static types allow it ($\tau'(a) \rightsquigarrow \mathsf{t}$);
2. a might share with at least one actual parameter s_p of the call at call-time;
3. moreover, if a is definitely aliased to an actual parameter s_p whose corresponding formal parameter $l_{p-j+\pi}$ is never re-assigned inside the callee m_w (i.e., there is no store $l_{p-j+\pi}$ in m_w), then it must also be the case that $l_{p-j+\pi}$ reaches s_0 (holding the returned value) at the end of m_w ($l_{p-j+\pi} \rightsquigarrow s_0 \in R_2$).

Similarly, an arbitrary variable b available after the call and different from the returned value $s_{j-\pi}$ ($b \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\}$) might be reachable from $s_{j-\pi}$ at that point ($s_{j-1} \rightsquigarrow b$) if the following conditions hold:

1. the static types allow it ($\mathsf{t} \rightsquigarrow \tau'(b)$);
2. b might be reachable from at least one actual parameter s_p at call-time ($s_p \rightsquigarrow b \in R_1$);

3. moreover, if b is definitely aliased to an actual parameter s_p whose corresponding formal parameter $l_{p-j+\pi}$ is never re-assigned inside the callee m_w (i.e., there is no store $l_{p-j+\pi}$ in m_w), then it must also be the case that s_0 (holding the returned value) reaches $l_{p-j+\pi}$ at the end of m_w ($s_0 \rightsquigarrow l_{p-j+\pi} \in R_2$).

The **side-effects multi-arcs** enlarge the reachability information at call-time with additional pairs of variables whose reachability is introduced by the callee by side-effect. These arcs do not consider the returned value of the method. We suppose that the topmost relevant operand stack element is $s_{\max}$. The complexity of these rules follows from the fact that we wanted a relatively precise, yet sound, approximation and, for that reason, we exploited the property that only variables that share with an actual parameter might be affected by the callee. Again, we do need sharing here, since it is well possible that those variables do not reach and are not reachable from any of the actual parameters of the call but yet share with one of them and are consequently affected by side-effects during the execution of the call. Moreover, we exploited the fact that the variables of the caller are often aliased to some actual parameter, in which case we can exploit the reachability information for the corresponding formal parameter inside the callee, for better precision. However, we must be sure that that formal parameter is not reassigned inside the callee. Namely, rule #20 adds a new pair $a \rightsquigarrow b$ of arbitrary variables if the following conditions hold:

1. if a and b exist after the call ($a, b \in \{l_0, \ldots, l_{i-1}, s_0, \ldots, s_{\max-1}\}$);
2. the static types allow it ($\tau'(a) \rightsquigarrow \tau'(b)$);
3. a might share with at least one actual parameter s_{p_a} at call-time;
4. b might be reachable from at least one actual parameter s_{p_b} of at call-time ($s_{p_b} \rightsquigarrow b \in R_1$);
5. if a and b are definitely aliased to two actual parameters s_{q_a} and s_{q_b}, whose corresponding formal parameters $l_{q_a-j+\pi}$ and $l_{q_b-j+\pi}$ are not re-assigned inside m_w (i.e., there is no store $l_{q_a-j+\pi}$ and no store $l_{q_b-j+\pi}$ in m_w), then $l_{q_a-j+\pi}$ might reach $l_{q_b-j+\pi}$ at the end of m_w ($l_{q_a-j+\pi} \rightsquigarrow l_{q_b-j+\pi} \in R_2$).

Example 5.28. Consider, for instance, nodes A and 10 in Fig. 5.5. In Example 5.23 we assumed that a reachability approximation at node A is known (Equation 5.5). Suppose that $R_{10} = \Pi^{\#12}(R_9)$, where $\Pi^{\#12}(R_9) = \{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_0 \rightsquigarrow l_2, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$, like it is determined in Example 5.25. Consider the side-effect arc linking nodes A and 10 with node B. Let us illustrate the application of the propagation rule #20 on approximations R_A and R_{10} in the presence of the sharing and aliasing approximations share_A and alias_A provided in Example 5.23. First of all we note that *con* has $\pi = 3$ actual parameters: the implicit parameter this, and 2 parameters of type `Object` and `List` respectively. Since *con* is a void methods, by Definition 3.19, we obtain $i_B = !4$ and $j_B = 1$ and, for each variable $v \in \mathsf{dom}(\tau_B) = \{l_0, l_1, l_2, l_3, s_0\}$, $\tau_B(v) = \tau_A(v)$. Approximations R_A and R_{10} are propagated by rule #20 (Fig. 5.4) as follows:

$$\{a \rightsquigarrow b \in R_A \mid a, b \in \mathsf{dom}(\tau_B)\} \cup \left\{ a \rightsquigarrow b \;\middle|\; \begin{array}{l} 1.\ a, b \in \mathsf{dom}(\tau_B) \wedge \\ 2.\ \tau_B(a) \rightsquigarrow \tau_B(b) \wedge \\ 3.\ \exists 1 \le p_a \le 3 . \langle a, s_{p_a} \rangle \in \mathsf{share}_A \wedge \\ 4.\ \exists 1 \le p_b \le 3 . s_{p_b} \rightsquigarrow b \in R_A \wedge \\ 5.\ \text{if } \exists 1 \le q_a, q_b \le 3 . \langle a, s_{q_a} \rangle, \langle b, s_{q_b} \rangle \in \mathsf{alias}_A \\ \quad \text{and } \textit{con} \text{ cotains no store } l_{q_a-1} \text{ nor store } l_{q_b-1}, \\ \quad \text{then } l_{q_a-1} \rightsquigarrow l_{q_b-1} \in R_{10} \end{array} \right\}.$$

The left-hand set ($\{a \rightsquigarrow b \in R_A \mid a, b \in \mathsf{dom}(\tau_B)\}$) extracts from R_A all the pairs composed of only those variables available in $\mathsf{dom}(\tau_B)$, i.e., the following pairs are added to R_B: $l_0 \rightsquigarrow l_0$, $l_1 \rightsquigarrow l_1$, $l_3 \rightsquigarrow l_3$, and $s_0 \rightsquigarrow s_0$. The right-hand set enlarges R_B by adding some new reachability information characterized by the 5 conditions of the rule #20. Conditions 1 and 2 add all possible ordered pairs of variables available in $\mathsf{dom}(\tau_B)$ such that the static type of the first variable reaches the static type of the second one, i.e., they give rise to the following pairs: $\{l_0, l_1, s_0\} \times \{l_0, l_1, l_3, s_0\} \cup \{l_3 \rightsquigarrow l_3\}$. Conditions 3 and 4 improves the precision of this approximation. Namely, condition 3 considers as the first element of a pair only those variables that might share with an actual parameter of *con* at A, and only l_1, l_3 and s_0 satisfy this condition (share_A). On the other hand, condition 4 considers as the second element of a pair only those variables that might be reachable from an actual parameter of *con* at A, and only l_3 and s_0 satisfy this requirement ($s_2 \rightsquigarrow l_3, s_1 \rightsquigarrow s_0 \in R_A$). Therefore, these two conditions restrict the former approximation to $\{l_1, s_0\} \times \{l_3, s_0\} \cup \{l_3 \rightsquigarrow l_3\}$. Condition 5 makes no further improvement of the approximation. Therefore, rule #20 adds to R_B, the over-approximation of the actual reachability information at node B, the following pairs of variables:

$$R_B = \Pi^{\#20}(R_A, R_{10}) = \{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_1 \rightsquigarrow l_3, l_1 \rightsquigarrow s_0, l_3 \rightsquigarrow l_3, s_0 \rightsquigarrow l_3, s_0 \rightsquigarrow s_0\}. \quad (5.6)$$

It is worth noting that, like we required at the end of Example 5.23, our reachability analysis actually provides the pair $s_0 \rightsquigarrow l_3$ in the approximation at node B.

Note that propagation rules #4, #19 and #20 use possible sharing and definite aliasing information between program variables. As we mentioned above, if these approximations are missing, one can always soundly assume that every pair of variables might share ($\mathsf{share}'_A = \{\langle a, b\rangle \mid a, b \in \mathsf{dom}(\tau_B)\}$) and is definitely not aliased ($\mathsf{alias}'_A = \emptyset$), although this reduces the precision of the propagation. In fact, if we apply rule #20 in a context with share'_A and alias'_A, (i.e., in the absence of possible sharing and definite aliasing approximations), conditions 1-4 would give rise to the following pairs: $\{l_0, l_1, s_0\} \times \{l_3, s_0\} \cup \{l_3 \rightsquigarrow l_3\}$, and condition 5 would not eliminate any pair. This way, the propagated reachability information becomes:

$$\{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_3, l_1 \rightsquigarrow l_1, l_1 \rightsquigarrow l_3, l_1 \rightsquigarrow s_0, l_3 \rightsquigarrow l_3, s_0 \rightsquigarrow l_3, s_0 \rightsquigarrow s_0\},$$

which is a bit less precise than (5.6). □

In our experiments (Section 5.5) the reachability analysis is performed inside the nullness and termination tools of Julia, that already perform definite aliasing [63] and possible sharing [82] analyses.

At this point we can formally define our reachability analysis in the general framework of constraint-based analyses.

Definition 5.29 (Possible Reachability Analysis). Possible Reachability Analysis *is a* system of constraints *extracted from the* ACG *whose nodes are enriched with elements of the abstract domain* Reach*, and whose arcs are enriched with the propagation rules from Figures 5.3 and 5.4. The extraction of constraints and the generation of the system of constraints concerning an* ACG *is explained in Section 4.5.*

Example 5.30. In Fig. 5.6 we provide the constraints extracted from the ACG introduced in Fig. 5.5. □

$$
\begin{array}{llllll}
\Pi^{\#18}(R_A) & \subseteq R_1 & \Pi^{\#5}(R_5) & \subseteq R_6 & & \\
\Pi^{\#1}(R_1) & \subseteq R_2 & \Pi^{\#1}(R_6) & \subseteq R_7 & \Pi^{\#11}(R_{11}) & \subseteq R_{12} \\
\Pi^{\#20}(R_2, R_{\mathsf{exit}}) & \subseteq R_3 & \Pi^{\#1}(R_7) & \subseteq R_8 & \Pi^{\#14}(R_{12}) & \subseteq R_{13} \\
\Pi^{\#1}(R_3) & \subseteq R_4 & \Pi^{\#5}(R_8) & \subseteq R_9 & \Pi^{\#20}(R_A, R_{10}) & \subseteq R_B \\
\Pi^{\#1}(R_4) & \subseteq R_5 & \Pi^{\#12}(R_9) & \subseteq R_{10} & \Pi^{\#20}(R_A, R_{13}) & \subseteq R_C \\
\multicolumn{6}{l}{\Pi^{\#20}(R_2, R_{\mathsf{exception}}) \cup \Pi^{\#16}(R_2) \cup \Pi^{\#17}(R_5) \cup \Pi^{\#17}(R_8) \subseteq R_{11}}
\end{array}
$$

Fig. 5.6. Constraints generated from the ACG from Fig. 5.5

5.4 Soundness of the Reachability Analysis

The goal of this section is to prove that there exists a solution to the system of constraints extracted from the ACG which deals with the reachability analysis we defined in the previous section, and that this solution is sound. Since we follow the constraint-based approach defined in Chapter 4, if we prove that the requirements provided in the latter (Requirements 4.1- 4.11) hold, results obtained in Sections 4.5 and 4.6 guarantee the existence of the least solution of the system of constraints mentioned above, as well as its soundness. The following subsections show that the instantiation of the general parameterized framework for constraint-based statyc analyses of Java bytecode program that we present in this section, i.e., the abstract domain Reach and the propagation rules introduced in Figures 5.3 and 5.4, satisfy those requirements.

5.4.1 ACC Condition

This requirement is one of the conditions which guarantee the existence and the uniqueness of the least solution of our analyses, like Theorem 4.11 has shown. The following lemma shows that the abstract domain Reach, and therefore our reachability analysis satisfy it.

Lemma 5.31. *The abstract domain* Reach *satisfies Requirement 4.1. More precisely, given a type environment $\tau \in \mathcal{T}$, every ascending chain of elements in* Reach_τ *eventually stabilizes.*

Proof. We recall that

$$\text{Reach}_\tau = \langle \wp(\mathsf{dom}(\tau) \times \mathsf{dom}(\tau)), \subseteq, \cup, \cap, \wp(\mathsf{dom}(\tau) \times \mathsf{dom}(\tau)), \emptyset \rangle.$$

There is a finite number of variables belonging to $\mathsf{dom}(\tau)$, and therefore the number of all possible ordered pairs of those variables is finite too. Every abstract element is a set of pairs of variables, and the partial ordering is the set inclusion $\subseteq$. Therefore, the greatest element an ascending chain might have is the top element, $\wp(\mathsf{dom}(\tau) \times \mathsf{dom}(\tau))$, which is finite, which implies that this ascending chain eventually stabilizes, i.e., Reach_τ satisfies the ACC condition. Hence, Requirement 4.1 is satisfied. ∎

5.4.2 Galois Connection

The next step is to show that the concretization map γ that we introduced in Definition 5.20 gives rise to a Galois connection (Chapter 2). This result would imply the satisfiability of Requirement 4.2. We start this proof by showing that our γ map is co-additive.

Lemma 5.32. *For any type environment $\tau \in \mathcal{T}$, the function γ_τ is co-additive, i.e.,*

$$\gamma_\tau(\bigcap_{i\geq 0} R_i) = \bigcap_{i\geq 0} \gamma_\tau(R_i).$$

Proof. Let $R_i \in \textsc{Reach}_\tau$ for an $i \geq 0$. We have:

$$\begin{array}{ll}
\gamma_\tau(\bigcap_{i\geq 0} R_i) & \\
= \{\sigma \in \Sigma_\tau \mid \forall a, b \in \mathsf{dom}(\tau). a \rightsquigarrow^\sigma b \Rightarrow a \rightsquigarrow b \in \cap_{i\geq 0} R_i\} & \text{[Definition 5.20]} \\
= \{\sigma \in \Sigma_\tau \mid \forall a, b \in \mathsf{dom}(\tau). a \rightsquigarrow^\sigma b \Rightarrow \bigwedge_{i\geq 0} a \rightsquigarrow b \in R_i\} & [x \in \cap_i X_i \Leftrightarrow \wedge_i x \in X_i] \\
= \{\sigma \in \Sigma_\tau \mid \forall a, b \in \mathsf{dom}(\tau). \bigwedge_{i\geq 0} (a \rightsquigarrow^\sigma b \Rightarrow a \rightsquigarrow b \in R_i)\} & [y \Rightarrow \wedge_i x_i \Leftrightarrow \wedge_i y \Rightarrow x_i] \\
= \{\sigma \in \Sigma_\tau \mid \bigwedge_{i\geq 0} (\forall a, b \in \mathsf{dom}(\tau). (a \rightsquigarrow^\sigma b \Rightarrow a \rightsquigarrow b \in R_i))\} & \left[\begin{array}{c} \forall x \in X. \wedge_i f(x, y_i) \\ \Leftrightarrow \\ \wedge_i \forall x \in X. f(x, y_i) \end{array}\right] \\
= \bigcap_{i\geq 0} \{\sigma \in \Sigma_\tau \mid \forall a, b \in \mathsf{dom}(\tau). a \rightsquigarrow^\sigma b \Rightarrow a \rightsquigarrow b \in R_i\} & \\
= \bigcap_{i\geq 0} \gamma_\tau(R_i) &
\end{array}$$

■

We can now show that the map γ actually gives rise to a Galois connection.

Lemma 5.33. *The abstract domain* Reach *satisfies Requirement 4.2. More precisely, given a type environment $\tau \in \mathcal{T}$, and the function $\gamma_\tau : \textsc{Reach}_\tau \to \mathsf{C}_\tau$, there exists a function $\alpha_\tau : \mathsf{C}_\tau \to \textsc{Reach}_\tau$ such that $\langle \mathsf{C}_\tau, \alpha_\tau, \gamma_\tau, \textsc{Reach}_\tau \rangle$ is a Galois connection.*

Proof. Both C_τ and $\textsc{Reach}_\tau$ are complete lattices. Moreover, Lemma 5.32 shows that γ_τ is co-additive and therefore, by the results mentioned in Chapter 2 (subsection Galois Connection), there exists the unique map α_τ, determined as:

$$\forall C \in \mathsf{C}_\tau. \alpha(C) = \bigcap \{R \in \textsc{Reach}_\tau | C \subseteq \gamma_\tau(R)\},$$

such that $\langle \mathsf{C}_\tau, \alpha_\tau, \gamma_\tau, \textsc{Reach}_\tau \rangle$ is a Galois connection. Therefore, Requirement 4.2 is satisfied and $\textsc{Reach}_\tau$ is actually an abstract domain, in the sense of abstract interpretation. ■

5.4.3 Monotonicity of the Propagation Rules

Another important condition necessary for the proof of existence of the least solution of our analysis is the monotonicity of the propagation rules. These rules represent an abstract semantics of bytecode instructions. We enunciate the following lemma without a proof, since it is straightforward.

Lemma 5.34. *The propagation rules from Figures 5.3 and 5.4 satisfy Requirement 4.3, i.e., they are monotonic with respect to $\subseteq$.*

Proof. This proof is straightforward from the definition of the propagation rules. ■

5.4.4 Sequential Arcs

This subsection is dedicated to Requirement 4.4, which states that the sequential arcs should propagate only the non-exceptional concrete states in the concretization of a correct approximation of the property of interest before a bytecode instruction is executed. This holds because sequential arcs link a bytecode instruction with its normally subsequently executed bytecode, when no exception is thrown, and their semantics must hence be consistent with that situation.

Lemma 5.35. *The propagation rules* #1-#11 *from Fig. 5.3 satisfy Requirement 4.4. More precisely, let us consider a sequential arc from a bytecode* ins *and its propagation rule* Π. *Assume that* ins *has static type information* τ *at its beginning and* τ' *immediately after its non-exceptional execution. Then, for every* $R \in \textsc{Reach}_\tau$ *we have:*

$$ins(\gamma_\tau(R)) \cap \Xi_{\tau'} \subseteq \gamma_{\tau'}(\Pi(R))$$

(we recall that ins is the semantics of ins*, see Fig. 3.6).*

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contain i local variables $L = \{l_0, \ldots, l_{i-1}\}$ and j operand stack elements $S = \{s_0, \ldots, s_{j-1}\}$. Let $\mathsf{dom}(\tau') = L' \cup S'$, where L' and S' are the local and operand stack variables of $\mathsf{dom}(\tau')$. Consider an arbitrary abstract element $R \in \textsc{Reach}_\tau$ and a state $\omega' = \langle \rho', \mu' \rangle \in ins(\gamma_\tau(R)) \cap \Xi_{\tau'}$. We prove that $\omega' \in \gamma_{\tau'}(\Pi(R))$, i.e., (Definition 5.20) that

$$\text{for every } x, y \in \mathsf{dom}(\tau'),\ x \rightsquigarrow^{\omega'} y \text{ entails } x \rightsquigarrow y \in \Pi(R).$$

The latter can be proved by showing that either $x \not\rightsquigarrow^{\omega'} y$ or $x \rightsquigarrow y \in \Pi(R)$. Note that, by the choice of ω', there exists $\omega = \langle \rho, \mu \rangle \in \gamma_\tau(R)$ such that $\omega' = ins(\omega)$. Moreover, $\omega \in \gamma_\tau(R)$ implies that for every $x, y \in \mathsf{dom}(\tau)$, $x \rightsquigarrow^{\omega} y$ entails $x \rightsquigarrow y \in R$. We analyze different propagation rules corresponding to different types of sequential arcs.

If $\mathsf{ins} = \mathsf{load}\ k\ \mathsf{t}$. We have $L' = L$, $S' = S \cup \{s_j\}$, $\mu' = \mu$ and for every $a \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\rho'(a) = \rho(a)$, while $\rho'(s_j) = \rho'(l_k)$. By definition of the propagation rules, $\Pi(R) = R \cup R[l_k/s_j] \cup R_1$, where $R_1 = \{l_k \rightsquigarrow s_j, s_j \rightsquigarrow l_k \mid l_k \rightsquigarrow l_k \in R\}$. We distinguish the following cases:

- if $x, y \neq s_j$, then, $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 5.7,
$$x \rightsquigarrow^{\omega'} y \Leftrightarrow x \rightsquigarrow^{\omega} y \Rightarrow x \rightsquigarrow y \in R \subseteq \Pi(R).$$
- if $x = s_j$ and $y \neq s_j$, then $\rho'(x) = \rho'(s_j) = \rho(l_k)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 5.7,
$$s_j \rightsquigarrow^{\omega'} y \Leftrightarrow l_k \rightsquigarrow^{\omega} y \Rightarrow l_k \rightsquigarrow y \in R.$$
If $y = l_k$, then $l_k \rightsquigarrow l_k \in R$, hence $x \rightsquigarrow y = s_j \rightsquigarrow l_k \in\in R_1 \subseteq \Pi(R)$. If $y \neq l_k$, then $l_k \rightsquigarrow y \in R$ implies that $x \rightsquigarrow y = s_j \rightsquigarrow y \in R[l_k/s_j] \subseteq \Pi(R)$.
- if $x \neq s_j$ and $y = s_j$, then $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho'(s_j) = \rho(l_k)$. Hence, by Lemma 5.7,
$$x \rightsquigarrow^{\omega'} s_j \Leftrightarrow x \rightsquigarrow^{\omega} l_k \Rightarrow x \rightsquigarrow l_k \in R.$$
If $x = l_k$, then $l_k \rightsquigarrow l_k \in R$, hence $x \rightsquigarrow y = l_k \rightsquigarrow s_j \in R_1 \subseteq \Pi(R)$. If $x \neq l_k$, then $x \rightsquigarrow l_k \in R$ implies that $x \rightsquigarrow y = x \rightsquigarrow s_j \in R[l_k/s_j] \subseteq \Pi(R)$.

- if $x = y = s_j$, then $\rho'(x) = \rho(l_k)$ and $\rho'(y) = \rho(l_k)$. Hence, by Lemma 5.7,

$$s_j \rightsquigarrow^{\omega'} s_j \Leftrightarrow l_k \rightsquigarrow^{\omega} l_k \Rightarrow l_k \rightsquigarrow l_k \in R,$$

  and therefore $x \rightsquigarrow y = s_j \rightsquigarrow s_j \in R[l_k/s_j] \subseteq \Pi(R)$.

If $\mathsf{ins} = \mathsf{store}\ k\ \mathsf{t}$. We have $L' = L$, $S' = S \setminus \{s_{j-1}\}$, $\mu' = \mu$, and for every $a \in \mathsf{dom}(\tau') \setminus \{l_k\}$, $\rho'(a) = \rho(a)$, while $\rho'(l_k) = \rho(s_{j-1})$. By definition of the propagation rules, $\Pi(R) = \{(a \rightsquigarrow b)[s_{j-1}/l_k] \mid a \rightsquigarrow b \in R \wedge a, b \neq l_k\}$. We distinguish the following cases:

- if $x, y \neq l_k$, then, $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 5.7,

$$x \rightsquigarrow^{\omega'} y \Leftrightarrow x \rightsquigarrow^{\omega} y \Rightarrow x \rightsquigarrow y \in R,$$

  and therefore $x \rightsquigarrow y \in \Pi(R)$.
- if $x = l_k$ and $y \neq l_k$, then $\rho'(x) = \rho'(l_k) = \rho(s_{j-1})$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 5.7,

$$l_k \rightsquigarrow^{\omega'} y \Rightarrow s_{j-1} \rightsquigarrow^{\omega} y \Rightarrow s_{j-1} \rightsquigarrow y \in R,$$

  and therefore $x \rightsquigarrow y = l_k \rightsquigarrow y = (s_{j-1} \rightsquigarrow y)[s_{j-1}/l_k] \in \Pi(R)$.
- if $x \neq l_k$ and $y = l_k$, then $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho'(l_k) = \rho(s_{j-1})$. Hence, by Lemma 5.7,

$$x \rightsquigarrow^{\omega'} l_k \Leftrightarrow x \rightsquigarrow^{\omega} s_{j-1} \Rightarrow x \rightsquigarrow s_{j-1} \in R,$$

  and therefore $x \rightsquigarrow y = x \rightsquigarrow l_k = (x \rightsquigarrow s_{j-1})[s_{j-1}/l_k] \in \Pi(R)$.
- if $x = y = l_k$, then $\rho'(x) = \rho'(y) = \rho'(l_k) = \rho(s_{j-1})$. Hence, by Lemma 5.7,

$$l_k \rightsquigarrow^{\omega'} l_k \Leftrightarrow s_{j-1} \rightsquigarrow^{\omega} s_{j-1} \Rightarrow s_{j-1} \rightsquigarrow s_{j-1} \in R,$$

  and therefore $x \rightsquigarrow y = l_k \rightsquigarrow l_k = (s_{j-1} \rightsquigarrow s_{j-1})[s_{j-1}/l_k] \in \Pi(R)$.

$\mathsf{ins} = \mathsf{const}\ x$. We have $L' = L$, $S' = S \cup \{s_j\}$, $\rho'(s_j) = v \in \mathbb{Z} \cup \{\mathtt{null}\}$, $\mu' = \mu$ and, for every $a \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\rho'(a) = \rho(a)$. By definition of the propagation rules, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$. We distinguish the following cases:

- if $x = s_j$ or $y = s_j$, since $\rho'(s_j) \in \mathbb{Z} \cup \{\mathtt{null}\}$, no variable reaches x nor y nor can be reached from them; hence $x \not\rightsquigarrow^{\omega'} y$.
- if $x, y \neq s_j$, then $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 5.7, $x \rightsquigarrow^{\omega'} y$ iff $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$, and since $x, y \in \mathsf{dom}(\tau')$, we have $x \rightsquigarrow y \in \Pi(R)$.

If $\mathsf{ins} = \mathsf{new}\ \kappa$. We have $L' = L$ and $S' = S \cup \{s_j\}$. Moreover, for every $a \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\rho'(a) = \rho(a)$, while $\rho'(s_j) = \ell \in \mathbb{L}$, where ℓ is a fresh location, hence only reachable from itself, and $\mu' = \mu[\ell \mapsto o]$, where o is a new object of class κ. Since ℓ is a fresh location, we have $\mathsf{L}_{\mu'}(\ell) = \{\ell\}$ and for every $\ell' \in \mathsf{dom}(\mu')$, $\ell \notin \mathsf{L}_{\mu'}(\ell')$. By definition of the propagation rules, $\Pi(R) = R \cup \{s_j \rightsquigarrow s_j\}$. We distinguish the following cases:

- if $x, y \neq s_j$, then, $\rho'(x) = \rho(x)$, $\rho'(y) = \rho(y)$ and for every $\ell \in \mathsf{dom}(\mu)$, $\mu'(\ell) = \mu(\ell)$. Hence, by Lemma 5.7,

$$x \rightsquigarrow^{\omega'} y \Leftrightarrow x \rightsquigarrow^{\omega} y \Rightarrow x \rightsquigarrow y \in R \subseteq \Pi(R).$$

- if $x = y = s_j$, then $x \rightsquigarrow y = s_j \rightsquigarrow s_j \in \Pi(R)$.
- if $x = s_j$ and $y \neq s_j$, then since ℓ is fresh, $\rho'(y) \notin \{\ell\} = \mathsf{L}_{\omega'}(x)$, hence $x \not\rightsquigarrow^{\omega'} y$.

- if $x \neq s_j$ and $y = s_j$, then since ℓ is fresh, $\rho'(y) = \ell \notin \mathsf{L}_{\omega'}(x)$, hence $x \not\rightsquigarrow^{\omega'} y$.

If $\mathsf{ins} = \mathsf{getfield}\, f$. We have $L' = L$, $S' = S$, $\mu' = \mu$ and for every $a \in \mathsf{dom}(\tau') \setminus \{s_{j-1}\}$, $\rho'(a) = \rho(a)$, while $\rho'(s_{j-1}) = (\mu\rho(s_{j-1}).\phi)(f)$. Moreover, for every $a \in \mathsf{dom}(\tau') \setminus \{s_{j-1}\}$, $\tau'(a) = \tau(a)$, while $\tau'(s_{j-1}) \leq \mathsf{t}$ and $\tau(s_{j-1}) \rightsquigarrow \tau'(s_{j-1})$. By definition of the propagation rules,

$$\Pi(R) = \overbrace{\{a \rightsquigarrow b \in R \mid a, b \neq s_{j-1}\}}^{R_1} \cup \overbrace{\{s_{j-1} \rightsquigarrow b \in R \mid \mathsf{t} \rightsquigarrow \tau(b)\}}^{R_2} \underbrace{\{a \rightsquigarrow s_{j-1} \mid \tau(a) \rightsquigarrow \mathsf{t} \neq \mathtt{int} \wedge [a \text{ and } s_{j-1} \text{ might share at } \mathsf{ins}]\}}_{R_3}.$$

We distinguish the following cases:

- if $x, y \neq s_{j-1}$, then, $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 5.7,

  $$x \rightsquigarrow^{\omega'} y \Leftrightarrow x \rightsquigarrow^{\omega} y \Rightarrow x \rightsquigarrow y \in R,$$

  and therefore $x \rightsquigarrow y \in R_1 \subseteq \Pi(R)$.
- if $x = s_{j-1}$ and $y \neq s_{j-1}$, if $s_{j-1} \rightsquigarrow^{\omega'} y$ then, by Lemma 5.17 and Definition 5.12, we have $\tau'(s_{j-1}) \rightsquigarrow \tau'(y) = \tau(y) \Rightarrow \tau(y) \in \mathsf{T}(\tau'(s_{j-1}))$. On the other hand, $\tau'(s_{j-1}) \leq \mathsf{t}$ and, by Lemma 5.15, $\mathsf{T}(\tau'(s_{j-1})) \subseteq \mathsf{T}(\mathsf{t})$, hence $\tau(y) \in \mathsf{T}(\tau'(s_{j-1})) \in \mathsf{T}(\mathsf{t})$, i.e.,

  $$\mathsf{t} \rightsquigarrow \tau(y). \tag{5.7}$$

  The locations reachable from a field of an object are included in those reachable from the object itself. More precisely, since $\rho'(s_{j-1}) = (\mu\rho(s_{j-1}).\phi)(f)$ and $\mu' = \mu$, we have $\mathsf{L}_{\omega'}(s_{j-1}) \subseteq \mathsf{L}_{\omega}(s_{j-1})$, and therefore $s_{j-1} \rightsquigarrow^{\omega'} y$ entails:

  $$\rho(y) = \rho'(y) \in \mathsf{L}_{\omega'}(s_{j-1}) \subseteq \mathsf{L}_{\omega}(s_{j-1}).$$

  Therefore, $s_{j-1} \rightsquigarrow^{\omega} y$ and

  $$s_{j-1} \rightsquigarrow y \in R. \tag{5.8}$$

  From (5.7) and (5.8) we conclude that $x \rightsquigarrow y = s_{j-1} \rightsquigarrow y \in R_2 \subseteq \Pi(R)$.
- if $y = s_{j-1}$, then if $x \rightsquigarrow^{\omega'} s_{j-1}$, by Definition 5.4 we have $\rho'(s_{j-1}) = (\mu\rho(s_{j-1}).\phi)(f) \in \mathbb{L}$, which entails $\tau'(s_{j-1}) \neq \mathtt{int}$. By type correctness, $\tau'(s_{j-1}) \leq \mathsf{t}$ and hence $\mathsf{t} \neq \mathtt{int}$. By Lemma 5.17, $x \rightsquigarrow^{\omega'} s_{j-1}$ implies that $\tau'(x) \rightsquigarrow \tau'(s_{j-1})$ and, since $\tau'(s_{j-1}) \leq \mathsf{t}$, Lemma 5.13 entails $\tau'(x) \rightsquigarrow \mathsf{t}$. If $x = s_{j-1}$ then, by Definition 5.12, we have that $\tau(s_{j-1}) \rightsquigarrow \tau'(s_{j-1}) \rightsquigarrow \mathsf{t}$, hence $\tau(s_{j-1}) \neq \mathtt{int}$ and it is obvious that s_{j-1} shares with itself at $\mathsf{getfield}$. Otherwise, if $x \neq s_{j-1}$, then $\tau(x) = \tau'(x) \rightsquigarrow \mathsf{t}$. In this case, $x \rightsquigarrow^{\omega'} s_{j-1}$ and Lemma 5.7 entail:

  $$\rho'(s_{j-1}) \in \mathsf{L}_{\omega'}(x) = \mathsf{L}_{\omega}(x). \tag{5.9}$$

  Moreover, Definition 5.1 and the fact that $\rho'(s_{j-1}) \in \mathbb{L}$ ensure that

  $$\rho'(s_{j-1}) = (\mu\rho(s_{j-1}).\phi)(f) \in \mathsf{rng}(\mu\rho(s_{j-1}).\phi) \cap \mathbb{L} \subseteq \mathsf{L}_{\omega}(s_{j-1}). \tag{5.10}$$

  Thus, (5.9) and (5.10) entail that $\mathsf{L}_{\omega}(x) \cap \mathsf{L}_{\omega}(s_{j-1}) \neq \emptyset$, i.e., x and s_{j-1} share at ins. Therefore, $\tau(x) \rightsquigarrow \mathsf{t} \neq \mathtt{int}$ and x and s_{j-1} share at ins, which entails that $x \rightsquigarrow y = x \rightsquigarrow s_{j-1} \in R_3 \subseteq \Pi(R)$.

If $\mathsf{ins} = \mathsf{putfield}\ f$. We have $L' = L$ and $S' = S \setminus \{s_{j-2}, s_{j-1}\}$. Moreover, $\rho' = \rho$ and $\mu' = \mu[(\mu\rho(s_{j-2}).\phi)(\kappa.f\colon \mathsf{t}) \mapsto \rho(s_{j-1})]$. By definition of the propagation rules,

$$\Pi(R) = \overbrace{\{a \rightsquigarrow b \in R \mid a, b \notin \{s_{j-1}, s_{j-2}\}\}}^{R_1} \cup \underbrace{\{a \rightsquigarrow b \mid a, b \notin \{s_{j-1}, s_{j-2}\} \wedge a \rightsquigarrow s_{j-2} \in R \wedge s_{j-1} \rightsquigarrow b \in R\}}_{R_2}$$

Assume that $x \rightsquigarrow^{\omega'} y$. We distinguish two cases:

- if $x \rightsquigarrow^{\omega} y$ then $x \rightsquigarrow y \in R$ and since $x, y \notin \{s_{j-2}, s_{j-1}\}$ we have $x \rightsquigarrow y \in R_1 \subseteq \Pi(R)$.
- if $x \not\rightsquigarrow^{\omega} y$ then (since $x \rightsquigarrow^{\omega'} y$) we show that the following relations hold: $x \rightsquigarrow^{\omega} s_{j-2}$ and $s_{j-1} \rightsquigarrow^{\omega} y$. Suppose that $x \not\rightsquigarrow^{\omega} s_{j-2}$, then by Definition 5.4, $\rho(s_{j-2}) \notin \mathsf{L}_{\omega}(x)$. Recall that μ and μ' differ on location $\rho(s_{j-2})$ only, and since $\rho(s_{j-2}) \notin \mathsf{L}_{\omega}(x)$, we have that for every $\ell \in \mathsf{L}_{\omega}(x).\mu'(\ell) = \mu(\ell)$. Moreover, $\rho'(x) = \rho(x)$, $\rho'(y) = \rho(y)$ and $\mathsf{dom}(\mu') = \mathsf{dom}(\mu)$, hence, by Lemma 5.7, $x \rightsquigarrow^{\omega'} y$ entails $x \rightsquigarrow^{\omega} y$, a contradiction. Therefore, $x \rightsquigarrow^{\omega} s_{j-2}$ and $x \rightsquigarrow s_{j-2} \in R$. Since $\mu' = \mu[(\mu\rho(s_{j-2}).\phi)(\kappa.f\colon \mathsf{t}) \mapsto \rho(s_{j-1})]$, we have, by Definition 5.3 and Lemma 5.2,

  $$\mathsf{L}_{\omega'}(x) = \mathsf{L}_{\mu'}(\rho'(x)) = \mathsf{L}_{\mu'}(\rho(x)) \subseteq \mathsf{L}_{\mu}(\rho(x)) \cup \mathsf{L}_{\mu}(\rho(s_{j-1})) = \mathsf{L}_{\omega}(x) \cup \mathsf{L}_{\omega}(s_{j-1}).$$

  It is worth noting that $x \rightsquigarrow^{\omega'} y$ entails $\rho(y) = \rho'(y) \in \mathsf{L}_{\omega'}(x) \subseteq \mathsf{L}_{\omega}(x) \cup \mathsf{L}_{\omega}(s_{j-1})$. By hypothesis, $\rho(y) \notin \mathsf{L}_{\omega}(x)$ (since $x \not\rightsquigarrow^{\omega} y$), and therefore $\rho(y) \in \mathsf{L}_{\omega}(s_{j-1})$, i.e., $s_{j-1} \rightsquigarrow^{\omega} y$ and $s_{j-1} \rightsquigarrow y \in R$. In conclusion we have $x \rightsquigarrow s_{j-2} \in R$ and $s_{j-1} \rightsquigarrow y \in R$ and hence $x \rightsquigarrow y \in R_2 \subseteq \Pi(R)$.

If $\mathsf{ins} = \mathsf{arraynew}\ \alpha$. We have $L' = L$ and $S' = S$. Moreover, for every $a \in \mathsf{dom}(\tau') \setminus \{s_{j-1}\}$, $\rho'(a) = \rho(a)$, while $\rho'(s_{j-1}) = \ell \in \mathbb{L}$, where ℓ is a fresh location, hence only reachable from itself, and $\mu' = \mu[\ell \mapsto a]$, where a is a new array of class α containing $\rho(s_{j-1})$ elements. By definition of the propagation rules, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \neq s_{j-1}\} \cup \{s_{j-1} \rightsquigarrow s_{j-1}\}$. We distinguish the following cases:

- if $x, y \neq s_{j-1}$, then, $\rho'(x) = \rho(x)$, $\rho'(y) = \rho(y)$ and for every $\ell \in \mathsf{dom}(\mu)$, $\mu'(\ell) = \mu(\ell)$. Hence, by Lemma 5.7,

  $$x \rightsquigarrow^{\omega'} y \Leftrightarrow x \rightsquigarrow^{\omega} y \Rightarrow x \rightsquigarrow y \in R \subseteq \Pi(R).$$

- if $x = y = s_{j-1}$, then $x \rightsquigarrow y = s_{j-1} \rightsquigarrow s_{j-1} \in \Pi(R)$.
- if $x = s_{j-1}$ and $y \neq s_{j-1}$, then since ℓ is fresh, $\rho'(y) \notin \{\ell\} = \mathsf{L}_{\omega'}(x)$, hence $x \not\rightsquigarrow^{\omega'} y$.
- if $x \neq s_{j-1}$ and $y = s_{j-1}$, then since ℓ is fresh, $\rho'(y) = \ell \notin \mathsf{L}_{\omega'}(x)$, hence $x \not\rightsquigarrow^{\omega'} y$.

If $\mathsf{ins} = \mathsf{arraylength}\ \alpha$. We have $L' = L$ and $S' = S$. Moreover, for every $a \in \mathsf{dom}(\tau') \setminus \{s_{j-1}\}$, $\rho'(a) = \rho(a)$, while $\rho'(s_{j-1}) = \mu\rho(s_{j-1}).\mathtt{length} \in \mathbb{Z}$, $\mu' = \mu$. By Definition 4.1, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \neq s_{j-1}\}$. We distinguish the following cases:

- if $x, y \neq s_{j-1}$, then, $\rho'(x) = \rho(x)$, $\rho'(y) = \rho(y)$ and for every $\ell \in \mathsf{dom}(\mu)$, $\mu'(\ell) = \mu(\ell)$. Hence, by Lemma 5.7,

  $$x \rightsquigarrow^{\omega'} y \Leftrightarrow x \rightsquigarrow^{\omega} y \Rightarrow x \rightsquigarrow y \in R \subseteq \Pi(R).$$

- if $x = s_{j-1}$ or $y = s_{j-1}$, then $x \not\rightsquigarrow^{\omega'} y$, since at least one of x and y is of type `int` and these variables do not reach anything and are not reachable from anything.

If $\mathsf{ins} = \mathsf{arrayload}\ \mathsf{t}[\ ]$. Analogously to the case $ins = \mathsf{getfield}\ f$, we have $L' = L$, $S' = \overline{S \setminus \{s_{j-1}\}}$, $\mu' = \mu$ and for every $a \in \mathsf{dom}(\tau') \setminus \{s_{j-2}\}$, $\rho'(a) = \rho(a)$, while $\rho'(s_{j-2}) = (\mu\rho(s_{j-2}).\phi)(\rho(s_{j-1}))$. Moreover, for every $a \in \mathsf{dom}(\tau') \setminus \{s_{j-2}\}$, $\tau'(a) = \tau(a)$, while $\tau'(s_{j-2}) \leq \mathsf{t}$ and $\tau(s_{j-2}) \rightsquigarrow \tau'(s_{j-2})$. By definition of the propagation rules,

$$\begin{aligned} \Pi(R) = \overbrace{\{a \rightsquigarrow b \in R \mid a, b \notin \{s_{j-1}, s_{j-2}\}\}}^{R_1} \cup \overbrace{\{s_{j-2} \rightsquigarrow b \in R \mid \mathsf{t} \rightsquigarrow \tau(b)\}}^{R_2} \\ \underbrace{\{a \rightsquigarrow s_{j-2} \mid \tau(a) \rightsquigarrow \mathsf{t} \neq \mathtt{int} \wedge [a \text{ and } s_{j-2} \text{ might share at } \mathsf{ins}]\}}_{R_3}. \end{aligned}$$

We distinguish the following cases:

- if $x, y \neq s_{j-2}$, then, $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 5.7,

  $$x \rightsquigarrow^{\omega'} y \Leftrightarrow x \rightsquigarrow^{\omega} y \Rightarrow x \rightsquigarrow y \in R,$$

  and therefore $x \rightsquigarrow y \in R_1 \subseteq \Pi(R)$.
- if $x = s_{j-2}$ and $y \neq s_{j-2}$, if $s_{j-2} \rightsquigarrow^{\omega'} y$ then, by Lemma 5.17 and Definition 5.12, we have $\tau'(s_{j-2}) \rightsquigarrow \tau'(y) = \tau(y) \Rightarrow \tau(y) \in \mathsf{T}(\tau'(s_{j-2}))$. On the other hand, $\tau'(s_{j-2}) \leq \mathsf{t}$ and, by Lemma 5.15, $\mathsf{T}(\tau'(s_{j-2})) \subseteq \mathsf{T}(\mathsf{t})$, hence $\tau(y) \in \mathsf{T}(\tau'(s_{j-2})) \in \mathsf{T}(\mathsf{t})$, i.e.,

  $$\mathsf{t} \rightsquigarrow \tau(y). \tag{5.11}$$

  The locations reachable from a array element are included in those reachable from the array itself. More precisely, since $\rho'(s_{j-2}) = (\mu\rho(s_{j-2}).\phi)(\rho(s_{j-1}))$ and $\mu' = \mu$, we have $\mathsf{L}_{\omega'}(s_{j-2}) \subseteq \mathsf{L}_{\omega}(s_{j-2})$, and therefore $s_{j-2} \rightsquigarrow^{\omega'} y$ entails:

  $$\rho(y) = \rho'(y) \in \mathsf{L}_{\omega'}(s_{j-2}) \subseteq \mathsf{L}_{\omega}(s_{j-2}).$$

  Therefore, $s_{j-2} \rightsquigarrow^{\omega} y$ and

  $$s_{j-2} \rightsquigarrow y \in R. \tag{5.12}$$

  From (5.11) and (5.12) we conclude that $x \rightsquigarrow y = s_{j-2} \rightsquigarrow y \in R_2 \subseteq \Pi(R)$.
- if $y = s_{j-2}$, then if $x \rightsquigarrow^{\omega'} s_{j-2}$, by Definition 5.4, $\rho'(s_{j-2}) = (\mu\rho(s_{j-2}).\phi)(\rho(s_{j-1})) \in \mathbb{L}$, which entails $\tau'(s_{j-2}) \neq \mathtt{int}$. By type correctness, $\tau'(s_{j-2}) \leq \mathsf{t}$ and hence $\mathsf{t} \neq \mathtt{int}$. By Lemma 5.17, $x \rightsquigarrow^{\omega'} s_{j-2}$ implies that $\tau'(x) \rightsquigarrow \tau'(s_{j-2})$ and, since $\tau'(s_{j-2}) \leq \mathsf{t}$, Lemma 5.13 entails $\tau'(x) \rightsquigarrow \mathsf{t}$. If $x = s_{j-2}$ then, by Definition 5.12, we have that $\tau(s_{j-2}) \rightsquigarrow \tau'(s_{j-2}) \rightsquigarrow \mathsf{t}$, hence $\tau(s_{j-2}) \neq \mathtt{int}$ and it is obvious that s_{j-2} shares with itself at $\mathsf{arrayload}$. Otherwise, if $x \neq s_{j-2}$, then $\tau(x) = \tau'(x) \rightsquigarrow \mathsf{t}$. In this case, $x \rightsquigarrow^{\omega'} s_{j-2}$ and Lemma 5.7 entail:

  $$\rho'(s_{j-2}) \in \mathsf{L}_{\omega'}(x) = \mathsf{L}_{\omega}(x). \tag{5.13}$$

  Moreover, Definition 5.1 and the fact that $\rho'(s_{j-2}) \in \mathbb{L}$ ensure that

  $$\rho'(s_{j-2}) = (\mu\rho(s_{j-2}).\phi)(\rho(s_{j-1})) \in \mathsf{rng}(\mu\rho(s_{j-2}).\phi) \cap \mathbb{L} \subseteq \mathsf{L}_{\omega}(s_{j-2}). \tag{5.14}$$

  Thus, (5.13) and (5.14) entail that $\mathsf{L}_{\omega}(x) \cap \mathsf{L}_{\omega}(s_{j-2}) \neq \emptyset$, i.e., x and s_{j-2} share at ins. Therefore, $\tau(x) \rightsquigarrow \mathsf{t} \neq \mathtt{int}$ and x and s_{j-2} share at ins, which entails that $x \rightsquigarrow y = x \rightsquigarrow s_{j-2} \in R_3 \subseteq \Pi(R)$.

If $\mathsf{ins} = \mathsf{arraystore\ t[\ ]}$. Analogously to the case $\mathsf{ins} = \mathsf{putfield}\ f$, we have $L' = L$, $S' = S \setminus \{s_{j-3}, s_{j-2}, s_{j-1}\}$, $\rho' = \rho$ and $\mu' = \mu[(\mu\rho(s_{j-3}).\phi)(\rho(s_{j-2})) \mapsto \rho(s_{j-1})]$. By definition of the propagation rules,

$$\Pi(R) = \overbrace{\{a \rightsquigarrow b \in R \mid a, b \notin \{s_{j-1}, s_{j-2}, s_{j-3}\}\}}^{R_1} \cup \underbrace{\{a \rightsquigarrow b \mid a, b \notin \{s_{j-1}, s_{j-2}, s_{j-3}\} \wedge a \rightsquigarrow s_{j-3} \in R \wedge s_{j-1} \rightsquigarrow b \in R\}}_{R_2}$$

Assume that $x \rightsquigarrow^{\omega'} y$. We distinguish two cases:

- if $x \rightsquigarrow^{\omega} y$ then $x \rightsquigarrow y \in R$ and since $x, y \notin \{s_{j-3}, s_{j-2}, s_{j-1}\}$ we have $x \rightsquigarrow y \in R_1 \subseteq \Pi(R)$.
- if $x \not\rightsquigarrow^{\omega} y$ then (since $x \rightsquigarrow^{\omega'} y$) we show that the following relations hold: $x \rightsquigarrow^{\omega} s_{j-3}$ and $s_{j-1} \rightsquigarrow^{\omega} y$. Suppose that $x \not\rightsquigarrow^{\omega} s_{j-3}$, then by Definition 5.4, $\rho(s_{j-3}) \notin \mathsf{L}_{\omega}(x)$. Recall that μ and μ' differ on location $\rho(s_{j-3})$ only, and since $\rho(s_{j-3}) \notin \mathsf{L}_{\omega}(x)$, we have that for every $\ell \in \mathsf{L}_{\omega}(x).\mu'(\ell) = \mu(\ell)$. Moreover, $\rho'(x) = \rho(x)$, $\rho'(y) = \rho(y)$ and $\mathsf{dom}(\mu') = \mathsf{dom}(\mu)$, hence, by Lemma 5.7, $x \rightsquigarrow^{\omega'} y$ entails $x \rightsquigarrow^{\omega} y$, a contradiction. Therefore, $x \rightsquigarrow^{\omega} s_{j-3}$ and $x \rightsquigarrow s_{j-3} \in R$. Since $\mu' = \mu[(\mu\rho(s_{j-3}).\phi)(\rho(s_{j-2})) \mapsto \rho(s_{j-1})]$, we have, by Definition 5.3 and Lemma 5.2,

  $$\mathsf{L}_{\omega'}(x) = \mathsf{L}_{\mu'}(\rho'(x)) = \mathsf{L}_{\mu'}(\rho(x)) \subseteq \mathsf{L}_{\mu}(\rho(x)) \cup \mathsf{L}_{\mu}(\rho(s_{j-1})) = \mathsf{L}_{\omega}(x) \cup \mathsf{L}_{\omega}(s_{j-1}).$$

  It is worth noting that $x \rightsquigarrow^{\omega'} y$ entails $\rho(y) = \rho'(y) \in \mathsf{L}_{\omega'}(x) \subseteq \mathsf{L}_{\omega}(x) \cup \mathsf{L}_{\omega}(s_{j-1})$. By hypothesis, $\rho(y) \notin \mathsf{L}_{\omega}(x)$ (since $x \not\rightsquigarrow^{\omega} y$), and therefore $\rho(y) \in \mathsf{L}_{\omega}(s_{j-1})$, i.e., $s_{j-1} \rightsquigarrow^{\omega} y$ and $s_{j-1} \rightsquigarrow y \in R$. In conclusion we have $x \rightsquigarrow s_{j-3} \in R$ and $s_{j-1} \rightsquigarrow y \in R$ and hence $x \rightsquigarrow y \in R_2 \subseteq \Pi(R)$.

$\mathsf{ins} = \mathsf{dup\ t}$. We have $L' = L$, $S' = S \cup \{s_j\}$, $\mu' = \mu$ and for every $a \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\rho'(a) = \rho(a)$, while $\rho'(s_j) = \rho'(s_{j-1})$. By definition of the propagation rules, $\Pi(R) = R \cup R[s_{j-1} \mapsto s_j] \cup R_1$, where $R_1 = \{s_{j-1} \rightsquigarrow s_j, s_j \rightsquigarrow s_{j-1} \mid s_{j-1} \rightsquigarrow s_{j-1} \in R\}$. We distinguish the following cases:

- if $x, y \neq s_j$, then, $\rho'(x) = \rho(x)$, $\rho'(y) = \rho(y)$. Hence, by Lemma 5.7,

  $$x \rightsquigarrow^{\omega'} y \Leftrightarrow x \rightsquigarrow^{\omega} y \Rightarrow x \rightsquigarrow y \in R \subseteq \Pi(R).$$

- if $x = s_j$ or $y = s_j$, we consider $\tau'(s_j)$: if $\tau'(s_j) = \tau(s_{j-1}) = \mathtt{int}$, we have $x \not\rightsquigarrow^{\omega'} y$. Otherwise we distinguish 3 cases:
  1. if $x = s_j$ and $y \neq s_j$, then $\rho'(x) = \rho'(s_j) = \rho(s_{j-1})$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 5.7,

     $$s_j \rightsquigarrow^{\omega'} y \Leftrightarrow s_{j-1} \rightsquigarrow^{\omega} y \Rightarrow s_{j-1} \rightsquigarrow y \in R.$$

     This implies $x \rightsquigarrow y = s_j \rightsquigarrow y \in R[s_{j-1}/s_j] \cup R_1 \subseteq \Pi(R)$.
  2. if $x \neq s_j$ and $y = s_j$, then $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho'(s_j) = \rho(s_{j-1})$. Hence, by Lemma 5.7,

     $$x \rightsquigarrow^{\omega'} s_j \Leftrightarrow x \rightsquigarrow^{\omega} s_{j-1} \Rightarrow x \rightsquigarrow s_{j-1} \in R.$$

     This implies $x \rightsquigarrow y = x \rightsquigarrow s_j \in R[s_{j-1}/s_j] \cup R_1 \subseteq \Pi(R)$.

3. if $x = y = s_j$, then $\rho'(x) = \rho(s_{j-1})$ and $\rho'(y) = \rho(s_{j-1})$. Hence, by Lemma 5.7,

$$x \rightsquigarrow^{\omega'} y \Leftrightarrow s_{j-1} \rightsquigarrow^{\omega} s_{j-1} \Rightarrow s_{j-1} \rightsquigarrow s_{j-1} \in R.$$

This implies $x \rightsquigarrow y = s_j \rightsquigarrow s_j \in R[s_{j-1}/s_j] \subseteq \Pi(R)$.

If ins = ifne t (ifeq t). We have $L' = L$, $S' = S \setminus \{s_{j-1}\}$, $\mu' = \mu$ and for every $a \in \mathsf{dom}(\tau')$, $\rho'(a) = \rho(a)$. By definition of the propagation rules, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$. By Lemma 5.7, $x \rightsquigarrow^{\omega'} y$ iff $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$, and therefore $x \rightsquigarrow y \in \Pi(R)$, since $x, y \in \mathsf{dom}(\tau')$.
If ins $\in$ {add, sub, mul, div, rem}. We have $L' = L$, $S' = S \setminus \{s_{j-1}\}$, $\mu' = \mu$ and for every $a \in \mathsf{dom}(\tau') \setminus \{s_{j-1}\}$, $\rho'(a) = \rho(a)$, while $\rho'(s_{j-1}) = \rho(s_{j-2}) \oplus \rho(s_{j-1}) \in \mathbb{Z}$, where $\oplus$ is the arithmetic operation corresponding to ins. Hence, for every variable $a \in \mathsf{dom}(\tau')$, both $s_{j-1} \not\rightsquigarrow^{\omega'} a$ and $a \not\rightsquigarrow^{\omega'} s_{j-1}$ hold. By definition of the propagation rules, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$. Suppose that $x \rightsquigarrow^{\omega'} y$, then $x, y \neq s_{j-1}$. By Lemma 5.7, $x \rightsquigarrow^{\omega'} y$ iff $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$, and therefore $x \rightsquigarrow y \in \Pi(R)$, since $x, y \in \mathsf{dom}(\tau')$.
If ins = inc k x. We have $L' = L$, $S' = S$, $\mu' = \mu$ and for every $a \in \mathsf{dom}(\tau') \setminus \{l_k\}$, $\rho'(a) = \rho(a)$, while $\rho'(l_k) = \rho(l_k) + x \in \mathbb{Z}$. Hence, for every variable $a \in \mathsf{dom}(\tau')$, both $l_k \not\rightsquigarrow^{\omega'} a$ and $a \not\rightsquigarrow^{\omega'} l_k$ hold. By definition of the propagation rules, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$. Suppose that $x \rightsquigarrow^{\omega'} y$, then $x, y \neq l_k$. By Lemma 5.7, $x \rightsquigarrow^{\omega'} y$ iff $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$, and therefore $x \rightsquigarrow y \in \Pi(R)$, since $x, y \in \mathsf{dom}(\tau')$.
ins $\in$ {catch, exception_is K}. We have $L' = L$, $S' = S = \{s_0\}$, $\mu' = \mu$ and, for every $a \in \mathsf{dom}(\tau')$, $\rho'(a) = \rho(a)$. By definition of the propagation rules, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$. By Lemma 5.7, we have $x \rightsquigarrow^{\omega'} y$ iff $x \rightsquigarrow^{\omega} y$, and since $x, y \in \mathsf{dom}(\tau')$, it entails $x \rightsquigarrow y \in R \in \Pi(R)$. ■

5.4.5 Final Arcs

This subsection is dedicated to Requirement 4.5, which states that in the case of the propagation rules of the final arcs, both exceptional and non-exceptional concrete states belonging to the concretization of a correct approximation of the property of interest before a bytecode instruction is executed, are correctly propagated by the corresponding rule. It means that the propagation rules of the final arcs must soundly approximate the concrete behavior of each final bytecode instruction (return t, return void, throw κ) of a method or a constructor belonging to the program under analysis. Let us show that this property actually holds.

Lemma 5.36. *The propagation rules* #12-#14 *from Fig. 5.3 satisfy Requirement 4.5. More precisely, let us consider a final arc from a bytecode* ins *and its propagation rule* Π*. 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*, its exceptional execution if* ins *is a* throw κ*). Then, for every* $R \in \textsc{Reach}_\tau$ *we have:*

$$\mathit{ins}(\gamma_\tau(R)) \subseteq \gamma_{\tau'}(\Pi(R))$$

(we recall that ins is the semantics of ins*, see Fig. 3.6).*

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

operand stack variables of $\mathsf{dom}(\tau')$. Consider an arbitrary abstract element $R \in \textsc{Reach}\tau$ and a state $\omega' = \langle \rho', \mu' \rangle \in \mathit{ins}(\gamma_\tau(R)) \cap \Xi_{\tau'}$. We prove that $\omega' \in \gamma_{\tau'}(\Pi(R))$, i.e., (Definition 5.20) that

$$\text{for every } x, y \in \mathsf{dom}(\tau'),\ x \rightsquigarrow^{\omega'} y \text{ entails } x \rightsquigarrow y \in \Pi(R).$$

The latter can be proved by showing that either $x \not\rightsquigarrow^{\omega'} y$ or $x \rightsquigarrow y \in \Pi(R)$. Note that, by the choice of ω', there exists $\omega = \langle \rho, \mu \rangle \in \gamma_\tau(R)$ such that $\omega' = \mathit{ins}(\omega)$. Moreover, $\omega \in \gamma_\tau(R)$ implies that for every $x, y \in \mathsf{dom}(\tau)$, $x \rightsquigarrow^{\omega} y$ entails $x \rightsquigarrow y \in R$. We analyze different propagation rules corresponding to different types of final arcs.

ins = return void. We have $L' = L$, $S' = \emptyset$, $\mu' = \mu$ and for every $a \in \mathsf{dom}(\tau')$, $\rho'(a) = \rho(a)$. By Definition 4.1, $\Pi(R) = \{a \rightsquigarrow b \mid a, b \notin S\}$. By Lemma 5.7, $x \rightsquigarrow^{\omega'} y$ iff $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$ and then $x \rightsquigarrow y \in \Pi(R)$ (since x and y are local variables).

ins = return t. We have $L' = L$, $S' = \{s_0\}$, $\mu' = \mu$ and for every $a \in \mathsf{dom}(\tau') \setminus \{s_0\}$, $\rho'(a) = \rho(a)$, while $\rho'(s_0) = \rho(s_{j-1})$. By Definition 4.1, $\Pi(R) = \{(a \rightsquigarrow b)[s_{j-1}/s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\}$. We consider the following cases:

- if $x, y \neq s_0$, then x and y are local variables, $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 5.7,
$$x \rightsquigarrow^{\omega'} y \Leftrightarrow x \rightsquigarrow^{\omega} y \Rightarrow x \rightsquigarrow y \in R.$$
Therefore, since x and y are local variables, we conclude $x \rightsquigarrow y \in \Pi(R)$.
- if $x \neq s_0$ and $y = s_0$, then $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho'(s_0) = \rho(s_{j-1})$. Hence, by Lemma 5.7,
$$x \rightsquigarrow^{\omega'} s_0 \Leftrightarrow x \rightsquigarrow^{\omega} s_{j-1} \Rightarrow x \rightsquigarrow y \in R.$$
Therefore, since x is a local variable and $s_{j-1} \notin \{s_0, \ldots, s_{j-2}\}$, we conclude $x \rightsquigarrow y = x \rightsquigarrow s_0 = (x \rightsquigarrow s_{j-1})[s_{j-1}/s_0] \in \Pi(R)$.
- if $x = s_0$ and $y \neq s_0$, then $\rho'(x) = \rho'(s_0) = \rho(s_{j-1})$ and $\rho'(y) = \rho(y)$; hence, by Lemma 5.7,
$$s_0 \rightsquigarrow^{\omega'} y \Leftrightarrow s_{j-1} \rightsquigarrow^{\omega} y \Rightarrow x \rightsquigarrow y \in R.$$
Therefore, since $s_{j-1} \notin \{s_0, \ldots, s_{j-2}\}$ and y is a local variable, we condlude $x \rightsquigarrow y = s_0 \rightsquigarrow y = (s_{j-1} \rightsquigarrow y)[s_{j-1}/s_0] \in \Pi(R)$.
- if $x = y = s_0$, then $\rho'(x) = \rho'(y) = \rho'(s_0) = \rho(s_{j-1})$. Hence, by Lemma 5.7,
$$s_0 \rightsquigarrow^{\omega'} s_0 \Leftrightarrow s_{j-1} \rightsquigarrow^{\omega} s_{j-1} \Rightarrow x \rightsquigarrow y \in R.$$
Therefore, since $s_{j-1} \notin \{s_0, \ldots, s_{j-2}\}$, we conclude
$$x \rightsquigarrow y = s_0 \rightsquigarrow s_0 = (s_{j-1} \rightsquigarrow s_{j-1})[s_{j-1}/s_0] \in \Pi(R).$$

ins = throw κ. We have $L' = L$, $S' = \{s_0\}$ and for every $a \in L'$, $\rho'(a) = \rho(a)$. By Definition 4.1, $\Pi(R) = \{(a \rightsquigarrow b)[s_{j-1}/s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\} \cup \{s_0 \rightsquigarrow s_0\}$. From Fig. 3.6 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 that if $x \rightsquigarrow^{\omega'} y$ then $x \rightsquigarrow y \in \{(a \rightsquigarrow b)[s_{j-1}/s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\} \subseteq \Pi(R)$. Or otherwise $\rho'(s_0) = \ell$ where ℓ is fresh and $\mu' = \mu[\ell \mapsto \mathit{npe}]$, 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 we have the following cases:

- if $x, y \neq s_0$, then, $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$; hence, by Lemma 5.7, $x \rightsquigarrow^{\omega'} y$ iff $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$ and therefore $x \rightsquigarrow y \in \Pi(R)$ (since x and y are local variables).
- if $x \neq s_0$ and $y = s_0$, then $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho'(s_0) = \ell$. Since ℓ is fresh, $\mathsf{L}_{\mu'}(\rho'(x)) \subseteq \mathsf{dom}(\mu)$ and $\rho'(y) \notin \mathsf{dom}(\mu)$, which entails $x \not\rightsquigarrow^{\omega'} y$.
- if $x = s_0$ and $y \neq s_0$, then $\rho'(y) = \rho(y)$ and $\rho'(x) = \rho'(s_0) = \ell$; then $\rho'(y) \in \mathsf{dom}(\mu)$ and $\mathsf{L}_{\mu'}(\rho'(x)) \cap \mathsf{dom}(\mu) = \emptyset$. Then $x \not\rightsquigarrow^{\omega'} y$.
- if $x = y = s_0$ we have $s_0 \rightsquigarrow s_0 \in \Pi(R)$.

■

5.4.6 Exceptional Arcs

This subsection is dedicated to Requirements 4.6 and 4.7. The latter states that in the case of the propagation rules of the exceptional arcs, the exceptional concrete states belonging to the concretization of a correct approximation of the property of interest before a bytecode instruction is executed, are correctly propagated by the corresponding rule. It means that the propagation rules of the exceptional arcs simulating the exceptional executions of the bytecode instructions which can throw an exception have to be sound. Let us show that this property actually holds.

Lemma 5.37. *The propagation rules* #15 *and* #17 *from Fig. 5.3 satisfy Requirement 4.6. More precisely, let us consider an exceptional arc from a bytecode* ins *distinct from* call *and its propagation rule Π. Assume that* ins *has static type information τ at its beginning and τ' immediately after its exceptional execution. Then, for every $R \in \textsc{Reach}_\tau$ we have:*

$$ins(\gamma_\tau(R)) \cap \Xi_{\tau'} \subseteq \gamma_{\tau'}(\Pi(R))$$

(we recall that ins is the semantics of ins*, see Fig. 3.6).*

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contain i local variables $L = \{l_0, \ldots, l_{i-1}\}$ and j operand stack elements $S = \{s_0, \ldots, s_{j-1}\}$; let $\mathsf{dom}(\tau') = L' \cup S'$, where L' and $S' = \{s_0\}$ are the local and operand stack variables of $\mathsf{dom}(\tau')$. Consider an arbitrary abstract element $R \in \textsc{Reach}_\tau$ and a state $\omega' = \langle \rho', \mu' \rangle \in ins(\gamma_\tau(R)) \cap \Xi_{\tau'}$. We prove that $\omega' \in \gamma_{\tau'}(\Pi(R))$, i.e., (Definition 5.20) that

$$\text{for every } x, y \in \mathsf{dom}(\tau'),\ x \rightsquigarrow^{\omega'} y \text{ entails } x \rightsquigarrow y \in \Pi(R).$$

The latter can be proved by showing that either $x \not\rightsquigarrow^{\omega'} y$ or $x \rightsquigarrow y \in \Pi(R)$. Note that, by the choice of ω', there exists $\omega = \langle \rho, \mu \rangle \in \gamma_\tau(R)$ such that $\omega' = ins(\omega)$. Moreover, $\omega \in \gamma_\tau(R)$ implies that for every $x, y \in \mathsf{dom}(\tau)$, $x \rightsquigarrow^{\omega} y$ entails $x \rightsquigarrow y \in R$. We analyze different propagation rules corresponding to different types of exceptional arcs.

If ins $\in$ {div, rem, new, getfield, putfield, arraynew, arraylength, arrayload, arraystore}.

In this case we have $L' = L$ and $S' = \{s_0\}$. Moreover, for every $a \in L'$, $\rho'(a) = \rho(a)$, while $\rho'(s_0) = \ell \in \mathbb{L}$, where ℓ is a fresh location and $\mu' = \mu[\ell \mapsto o]$, where o is a new instance of the subclass of `Throwable` thrown by ins containing only fresh locations ($\mathsf{L}_{\mu'}(\ell) \cap \mathsf{dom}(\mu) = \emptyset$). By Definition 4.1, $\Pi(R) = \{a \rightsquigarrow b \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-1}\}\} \cup \{s_0 \rightsquigarrow s_0\}$. We distinguish the following cases:

- if $x, y \neq s_0$, then, $\rho'(x) = \rho(x)$, $\rho'(y) = \rho(y)$ and for every $\ell \in \mathsf{dom}(\mu)$, $\mu'(\ell) = \mu(\ell)$; hence, by Lemma 5.7, $x \rightsquigarrow^{\omega'} y$ iff $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$ and therefore $x \rightsquigarrow y \in \Pi(R)$ (since $x, y \in L'$).
- if $x = y = s_0$ we have $s_0 \rightsquigarrow s_0 \in \Pi(R)$.
- if $x = s_0$ and $y \neq s_0$, then $\rho'(y) \in \mathsf{dom}(\mu)$ and $\mathsf{L}_{\mu'}(\rho'(x)) \cap \mathsf{dom}(\mu) = \emptyset$. Hence $x \not\rightsquigarrow^{\omega'} y$.
- if $x \neq s_0$ and $y = s_0$, then $\rho'(y) = \ell \notin \mathsf{dom}(\mu)$ and $\mathsf{L}_{\mu'}(\rho'(x)) \subseteq \mathsf{dom}(\mu)$ (since ℓ is fresh). Then $x \not\rightsquigarrow^{\omega'} y$.

$\underline{\mathsf{ins} = \mathsf{throw}\ \kappa}$. Analogously to the proof of Lemma 4.2 for $\mathsf{throw}\kappa$, when $\rho'(s_0) = \ell$, where $\underline{\ell \text{ is a fresh location}}$. ■

On the other hand, Requirement 4.7 deals with one particular case of the exceptional arcs: when a method is invoked on a `null` receiver. In that case we require that the exceptional states launched by the method are included in the approximation of the property of interest after the call to that method. Let us show that our propagation rules satisfy this requirement.

Lemma 5.38. *The propagation rule* #16 *from Fig. 5.3 satisfies Requirement 4.7. More precisely, consider an exceptional arc from a method invocation* $\mathsf{ins}_C = \mathsf{call}\, m_1 \ldots m_n$ *and its propagation rule* Π*, and let* π *be the number of its actual arguments (*this *included). Then, for each* $1 \leq w \leq q$*, and every* $\sigma = \langle\langle \mathsf{l} \parallel v_{\pi-1} :: \ldots :: v_1 :: \texttt{null} : \mathsf{s}\rangle, \mu\rangle \in \gamma_\tau(R)$ *(*σ *assigns* `null` *to the receiver of* ins_C *right before it is executed), where* $R \in \textsc{Reach}_\tau$ *is an arbitrary abstract element, we have:*

$$\underline{\langle\langle \mathsf{l} \parallel \ell\rangle, \mu[\ell \mapsto npe]\rangle} \subseteq \gamma_{\tau'}(\Pi(R)),$$

where ℓ *is a fresh location, and* npe *is a new instance of* `NullPointerException`*.*

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contain local variables L and $j \geq \pi$ operand stack elements $S = \{s_0, \ldots, s_{j-\pi}, \ldots, s_{j-1}\}$, where π is the number of parameters of method m_w (including this). Consider an arbitrary abstract element $R \in \textsc{Reach}_\tau$ and a state $\sigma = \langle \rho, \mu\rangle = \langle\langle \mathsf{l} \parallel v_{\pi-1} :: \ldots :: v_1 :: \mathtt{null} : \mathsf{s}\rangle, \mu\rangle \in \gamma_\tau(R)$. Then, by Rule 3 from Fig. 3.7, we have that $\mathsf{dom}(\tau') = L \cup \{s_0\}$, and the resulting state $\sigma' = \langle \rho', \mu'\rangle$ is such that for each $a \in \mathsf{dom}(\tau') \setminus \{s_0\}$, $\rho'(a) = \rho(a)$, $\rho(s_0) = \ell$, where ℓ is a fresh location and $\mu' = \mu[\ell \mapsto npe]$, where npe is a new instance of `NullPointerException`. Hence, $\sigma' = \langle\langle \mathsf{l} \parallel \ell\rangle, \mu[\ell \mapsto npe]\rangle$. Moreover, by definition of the propagation rules,

$$\begin{array}{c} \Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \in \{l_0, \ldots, l_{i-1}\} \cup \{s_0 \rightsquigarrow s_0\} \\ \cup \underbrace{\{a \rightsquigarrow s_0 \mid a \in L \wedge \tau(a) \rightsquigarrow \mathtt{Throwable}\}}_{R_1} \cup \underbrace{\{s_0 \rightsquigarrow a \mid a \in L \wedge \mathtt{Throwable} \rightsquigarrow \tau(a)\}}_{R_2}. \end{array}$$

We must prove that $\sigma' \in \gamma_{\tau'}(\Pi(R))$ i.e., (Definition 5.20) that

$$\text{for every } x, y \in \mathsf{dom}(\tau'),\ x \rightsquigarrow^{\sigma'} y \Rightarrow x \rightsquigarrow y \in \Pi(R).$$

The latter can be proved by showing that either $x \not\rightsquigarrow^{\sigma'} y$ or $x \rightsquigarrow y \in \Pi(R)$. Note that, by hypothesis, $\sigma \in \gamma_\tau(R)$, i.e., for every $x, y \in \mathsf{dom}(\tau)$, $x \rightsquigarrow^{\sigma} y \Rightarrow x \rightsquigarrow y \in R$. Let x and y be arbitrary variables from $\mathsf{dom}(\tau')$. We distinguish the following cases:

- if $x, y \in L$, then $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 5.7, $x \rightsquigarrow^{\sigma} y$ iff $x \rightsquigarrow^{\sigma} y$, which entails $x \rightsquigarrow y \in R$, and therefore $x \rightsquigarrow y \in \Pi(R)$.
- if $x = s_0$ and $y \neq s_0$, then if there exists a variable $z \in L$ such that $s_0 \rightsquigarrow^{\sigma'} z$, then by Lemma 5.17, $\tau'(s_0) \rightsquigarrow \tau'(z) = \tau(z)$, i.e., $\tau(z) \in \mathsf{T}(\tau'(s_0))$. Moreover, $\tau'(s_0) \leq \texttt{Throwable}$, hence by Lemma 5.15, $\mathsf{T}(\tau'(s_0)) \subseteq \mathsf{T}(\texttt{Throwable})$, which entails $\tau(z) \in \mathsf{T}(\texttt{Throwable})$, i.e., $\texttt{Throwable} \rightsquigarrow \tau(z)$. Hence, $s_0 \rightsquigarrow z \in R_2$.
- if $x \neq s_0$ and $y = s_0$, then if there exists a variable $z \in L$ such that $z \rightsquigarrow^{\sigma'} s_0$, then by Lemma 5.17, $\tau(z) = \tau'(z) \rightsquigarrow \tau'(s_0)$. Moreover, $\tau'(s_0) \leq \texttt{Throwable}$, hence, by Lemma 5.13, we have that $\tau(z) \rightsquigarrow \texttt{Throwable}$. Hence, $z \rightsquigarrow s_0 \in R_1$.
- if $x = y = s_0$, then $s_0 \rightsquigarrow s_0 \in \Pi(R_p)$.

Therefore, $\langle\langle \mathsf{I} \parallel \ell \rangle, \mu[\ell \mapsto npe]\rangle = \sigma' \in \gamma_{\tau'}(\Pi(R))$. ■

5.4.7 Parameter Passing Arcs

This subsection is dedicated to Requirement 4.8 which states that the propagation rules of the parameter passing arcs are sound. Namely, this rule soundly approximates the behavior of the *makescope* function. Let us show that this property holds.

Lemma 5.39. *The propagation rule* #18 *from Fig. 5.3 satisfies Requirement 4.8. More precisely, let us consider a parameter passing arc from a method invocation* $\mathsf{ins}_C = \mathsf{call}\ m_1 \dots m_n$ *to the first bytecode of a callee* m_w*, for some* $w \in [1..k]$*, and its propagation rule* Π*. Assume that* ins_C *has static type information* τ *at its beginning and that* τ' *is the static type information at the beginning of* m_w*. Then, for every* $R \in \textsc{Reach}_\tau$ *we have:*

$$(\textit{makescope}\ m_w)(\gamma_\tau(R)) \subseteq \gamma_{\tau'}(\Pi(R))$$

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contain local variables L and $j \geq \pi$ operand stack elements $S = \{s_0, \dots, s_{j-\pi}, \dots, s_{j-1}\}$, where π is the number of parameters of method m_w (including this). Then, $\mathsf{dom}(\tau') = \{l_0, \dots, l_{\pi-1}\}$. Consider an arbitrary abstract element $R \in \textsc{Reach}_\tau$ and a state $\omega' = \langle \rho', \mu' \rangle \in \mathit{ins}(\gamma_\tau(R)) \cap \Xi_{\tau'}$. We prove that $\omega' \in \gamma_{\tau'}(\Pi(R))$, i.e., (Definition 5.20) that

$$\text{for every } x, y \in \mathsf{dom}(\tau'),\ x \rightsquigarrow^{\omega'} y \text{ entails } x \rightsquigarrow y \in \Pi(R).$$

By the choice of ω', there exists $\omega = \langle \rho, \mu \rangle \in \gamma_\tau(R)$ such that $\omega' = (\textit{makescope}\ m_w)(\omega)$. Moreover, $\omega \in \gamma_\tau(R)$ implies that for every $x, y \in \mathsf{dom}(\tau)$, $x \rightsquigarrow^{\omega} y$ entails $x \rightsquigarrow y \in R$. By definition of the propagation rules we have:

$$\Pi(R) = \left\{ (a \rightsquigarrow b) \left[\begin{smallmatrix} s_{j-\pi}/l_0 \\ \dots \\ s_{j-1}/l_{\pi-1} \end{smallmatrix} \right] \middle| \ a \rightsquigarrow b \in R \text{ and } a, b \in \{s_{j-\pi}, \dots, s_{j-1}\} \right\}$$

By Definition 3.17, for every $p \in [0, \pi)$, $\rho'(l_p) = \rho(s_{j-\pi+p})$ and $\mu' = \mu$. Consider $x, y \in \mathsf{dom}(\tau') = L'$. There exist $p, q \in [0..\pi)$ such that $x = l_p$ and $y = l_q$, and therefore $\rho'(x) = \rho'(l_p) = \rho(s_{j-\pi+p})$ and $\rho'(y) = \rho(l_q) = \rho(s_{j-\pi+q})$. Hence, by Lemma 5.7,

$$x \rightsquigarrow^{\omega'} y \Leftrightarrow s_{j-\pi+p} \rightsquigarrow^{\omega} s_{j-\pi+q} \Rightarrow s_{j-\pi+p} \rightsquigarrow s_{j-\pi+q} \in R.$$

Therefore, $x \rightsquigarrow y = l_p \rightsquigarrow l_q = (s_{j-\pi+p} \rightsquigarrow s_{j-\pi+q})[s_{j-\pi+p}/l_p, s_{j-\pi+q}/l_q] \in \Pi(R)$. ■

5.4.8 Return and Side-Effects Arcs at Non-Exceptional Ends

In this subsection we show that our reachability analysis satisfies Requirements 4.9 and 4.10. The following lemmas deal with the return values and side-effects of the non-exceptional executions of methods. Namely, in the case of a non-void method, the propagation rule of the return value arc enriches the resulting approximation of the reachability immediately after the call to that method by adding all those reachability pairs that the returned value might correspond to. On the other hand, that method might modify the initial memory from which the method has been executed. These modifications must be captured by the propagation rules of the side-effects arcs. The approximation of the property of interest after the call to the method is, therefore, determined as the union of the approximations obtained from the propagation rules of the return value and the side-effects arcs, and it is sound, like Lemma 5.40 shows. Lemma 5.41 handles the case of a void method, and therefore only the corresponding side-effects arc is considered there.

Lemma 5.40. *The propagation rules from Fig. 5.4 satisfy Requirement 4.9. More precisely, the propagation rules for the return value arcs and side-effects arcs are sound at* a non-void *method return. Namely, let* $w \in [1..n]$ *and consider a return value and a side-effect arc from nodes* $\mathsf{C} = \boxed{\mathsf{call}\ m_1 \dots m_n}$ *and* $\mathsf{E} = \boxed{\mathsf{exit@}m_w}$ *to a node* $\mathsf{Q} = \boxed{\mathsf{ins}_q}$ *and their propagation rules* $\Pi^{\#19}$ *and* $\Pi^{\#20}$*, respectively. Let* τ_c*,* τ_q *and* τ_e *be the static type information at* C*,* Q *and* E*, respectively, and let* d *be 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* $R_c \in \textsc{Reach}_{\tau_c}$ *and* $R_e \in \textsc{Reach}_{\tau_e}$*, we have:*

$$d((\mathit{makescope}\ m_w)(\gamma_{\tau_c}(R_c)) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}(\Pi^{\#19}(R_c, R_e) \cup \Pi^{\#20}(R_c, R_e)).$$

Proof. Consider states $\sigma_c \in \gamma_{\tau_c}(R_c)$, $\sigma_e \in \gamma_{\tau_e}(R_e)$ and $\sigma_q = d((\mathit{makescope}\ m_w)(\sigma_c)) \in \Xi_{\tau_q}$ and let us show that $\sigma_q \in \gamma_{\tau_q}(R_q)$, where $R_q = \Pi^{\#19}(R_c, R_e) \cup \Pi^{\#20}(R_c, R_e)$. By Definition 5.20,

$$\begin{aligned}\sigma_q \in \gamma_{\tau_q}(R_q) &\Leftrightarrow \forall a, b \in \mathsf{dom}(\tau_q).a \rightsquigarrow^{\sigma_q} b \Rightarrow a \rightsquigarrow b \in R_q \\ &\Leftrightarrow \forall a, b \in \mathsf{dom}(\tau_q).a \rightsquigarrow b \notin R_q \Rightarrow a \not\rightsquigarrow^{\sigma_q} b.\end{aligned}$$

In the following we use the following hypotheses: $\mathsf{dom}(\tau_a) = L_a \cup S_a$, where a can be c, e or q, $S_c = \{s_0, \dots, s_j\}$, $S_e = \{s_0\}$, $S_q = \{s_0, \dots, s_{j-\pi-1}, s_{j-\pi}\}$, $L_q = L_c$ and $\{l_0, \dots, l_{\pi-1}\} \subseteq L_e$, where j and π are number of operand stack elements in $\mathsf{dom}(\tau_c)$ and number of parameters of method m, respectively. By Definition 3.19, σ_c, σ_e and σ_q have to satisfy the following conditions: $\sigma_c = \langle\langle \mathsf{l}_c \parallel v_{j-1} :: \dots :: v_{j-\pi} :: \dots :: v_0\rangle, \mu_c\rangle$, $\sigma_e = \langle\langle \mathsf{l}_e \parallel \mathit{top}\rangle, \mu_e\rangle$ and $\sigma_q = \langle\langle \mathsf{l}_c \parallel \mathit{top} :: v_{j-\pi-1} :: \dots :: v_0\rangle, \mu_e\rangle$. Let a and b be two arbitrary variables from $\mathsf{dom}(\tau_q)$ and suppose that $a \rightsquigarrow b \notin R_q$. We show that in that case $a \not\rightsquigarrow^{\sigma_q} b$. We distinguish the following cases:

Case a: If $a = b = s_{j-\pi}$: Rule $\Pi^{\#15}$ adds the pair $s_{j-\pi} \rightsquigarrow s_{j-\pi}$ only if $s_0 \rightsquigarrow s_0 \in R_e$. Thus, $s_{j-\pi} \rightsquigarrow s_{j-\pi} \notin R_q$ implies $s_0 \rightsquigarrow s_0 \notin R_e$, which entails $s_0 \not\rightsquigarrow^{\sigma_e} s_0$ ($\sigma_e \in \gamma_{\tau_e}(R_e)$), and it is possible only if $\tau_e(s_0) = \texttt{int}$. Since $\tau_q(s_{j-\pi}) = \tau_e(s_0) = \texttt{int}$, we conclude $s_{j-\pi} \not\rightsquigarrow^{\sigma_q} s_{j-\pi}$.

Case b: If $a \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\}$ and $b = s_{j-\pi}$: If $a \rightsquigarrow s_{j-\pi} \notin R_q$ then, by rule $\Pi^{\#19}$:

1.$\tau_q(a) \not\rightsquigarrow \mathsf{t}$ or

2.there is no $j-\pi \leq p < j$, such that a might share with s_p at C or

3.there exists a $j-\pi \leq p < j$, such that a is definitely aliased to s_p at C and no store $l_{p-j+\pi}$ occurs in m_w and $l_{p-j+\pi} \rightsquigarrow s_0 \notin R_e$.

We analyze these 3 cases:

1. if $\tau_q(a) \not\rightsquigarrow \mathsf{t}$, then since $\tau_q(s_{j-\pi}) \leq \mathsf{t}$ ($s_{j-\pi}$ contains the value returned by method m, whose type is t), by Lemma 5.13, $\tau_q(a) \not\rightsquigarrow \tau_q(s_{j-\pi})$ holds, which, by Lemma 5.17 implies $a \not\rightsquigarrow^{\sigma_q} s_{j-\pi}$.
2. if there is no $j-\pi \leq p < j$, such that a might share with s_p at C, i.e., if at call-time a cannot share with the parameters of method m then, by Proposition 5.8, $\mathsf{L}_{\sigma_c}(a) \subseteq \mathcal{L}_{\sigma_c}$ and $\rho_q(s_{j-\pi}) \notin \mathcal{L}_{\sigma_c}$, which entails $\rho_q(s_{j-\pi}) \notin \mathsf{L}_{\sigma_c}(a)$. Since $\rho_c(a) = \rho_q(a) \in \mathcal{L}_{\sigma_c}$, by Lemma 5.10, we have $\mathsf{L}_{\sigma_c}(a) = \mathsf{L}_{\sigma_q}(a)$. Thus, $\rho_q(s_{j-\pi}) \notin \mathsf{L}_{\sigma_c}(a) = \mathsf{L}_{\sigma_q}(a)$, i.e., $a \not\rightsquigarrow^{\sigma_q} s_{j-\pi}$.
3. if there exists a $j-\pi \leq p < j$, such that a is definitely aliased to s_p at C, no store $l_{p-j+\pi}$ occurs in m_w and $l_{p-j+\pi} \rightsquigarrow s_0 \notin R_e$, we have:

$\rho_c(a) = \rho_c(s_p)$ [a is definitely aliased to s_p at C]
$\rho_c(s_p) = \rho_e(l_{p-j+\pi})$ [s_p at C corresponds to $l_{p-j+\pi}$ at E and there is no store $l_{p-j+\pi}$ in m_w]
$\rho_c(a) = \rho_q(a)$ [$a \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\}$ and Definition 3.19]
$l_{p-j+\pi} \rightsquigarrow s_0 \notin R_e \Rightarrow l_{p-j+\pi} \not\rightsquigarrow^{\sigma_e} s_0$ [By Definition 5.20]

Since $\rho_q(a) = \rho_e(l_{p-j+\pi})$, $\rho_q(s_{j-\pi}) = \rho_e(s_0)$ and $\mu_q = \mu_e$ (hypotheses about σ_q and σ_e), by Lemma 5.7, $a \rightsquigarrow^{\sigma_q} s_{j-\pi} \Leftrightarrow l_{p-j+\pi} \rightsquigarrow^{\sigma_e} s_0$. Since $l_{p-j+\pi} \not\rightsquigarrow^{\sigma_e} s_0$, we conclude that $a \not\rightsquigarrow^{\sigma_q} s_{j-\pi}$.

Thus, we proved that $a \not\rightsquigarrow s_{j-\pi} \notin R_q$ entails $a \not\rightsquigarrow^{\sigma_q} s_{j-\pi}$.

Case c: If $a = s_{j-\pi}$ and $b \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\}$: If $s_{j-\pi} \rightsquigarrow b \notin R_q$, then, by rule $\Pi^{\#19}$:

1.$\mathsf{t} \not\rightsquigarrow \tau_q(b)$ or

2.there is no $j-\pi \leq p < j$, such that $s_p \rightsquigarrow b \in R_c$ or

3.there exists a $j-\pi \leq p < j$, such that b is definitely aliased to s_p at C and no store $l_{p-j+\pi}$ occurs in m_w and $s_0 \rightsquigarrow l_{p-j+\pi} \notin R_e$.

We analyze these 3 cases:

1. if $\mathsf{t} \not\rightsquigarrow \tau_q(b)$, then $\tau_q(b) \notin \mathsf{T}(\mathsf{t})$. Since $s_{j-\pi}$ contains the value returned by method m, whose type is t, we have $\tau_q(s_{j-\pi}) \leq \mathsf{t}$. By Lemma 5.15, $\mathsf{T}(\tau_q(s_{j-\pi})) \subseteq \mathsf{T}(\mathsf{t})$, which entails $\tau_q(b) \notin \mathsf{T}(\tau_q(s_{j-\pi}))$, i.e., $\tau_q(s_{j-\pi}) \not\rightsquigarrow \tau_q(b)$ and, by Lemma 5.17, $s_{j-\pi} \not\rightsquigarrow^{\sigma_q} b$.
2. if there is no $j-\pi \leq p < j$, such that $s_p \rightsquigarrow b \in R_c$ then, by Definition 5.20, for each $j-\pi \leq p < j$, $s_p \not\rightsquigarrow^{\sigma_c} b$, i.e., $\rho_c(b) \notin \bigcup_{p \in [j-\pi, j)} \mathsf{L}_{\sigma_c}(s_p)$. By Proposition 5.8, $\rho_c(b) \in \mathcal{L}_{\sigma_c}$ and $\rho_q(s_{j-\pi}) \notin \mathcal{L}_{\sigma_c}$. By Lemma 5.9, $\mathsf{L}_{\sigma_q}(s_{j-\pi}) \cap \mathcal{L}_{\sigma_c} = \emptyset$, hence $\rho_c(b) \notin \mathsf{L}_{\sigma_q}(s_{j-\pi})$ and, since $\rho_c(b) = \rho_q(b)$, we conclude that $\rho_q(b) \notin \mathsf{L}_{\sigma_q}(s_{j-\pi})$, i.e., $s_{j-\pi} \not\rightsquigarrow^{\sigma_q} b$.
3. if there exists a $j-\pi \leq p < j$, such that b is definitely aliased to s_p at C, no store $l_{p-j+\pi}$ occurs in m_w and $s_0 \rightsquigarrow l_{p-j+\pi} \notin R_e$, we have:

$\rho_c(b) = \rho_c(s_p)$ [b is definitely aliased to s_p at C]
$\rho_c(s_p) = \rho_e(l_{p-j+\pi})$ [[s_p at C corresponds to $l_{p-j+\pi}$ at E and there is no store $l_{p-j+\pi}$ in m_w]
$\rho_c(b) = \rho_q(b)$ [$b \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\}$ and Definition 3.19]
$s_0 \rightsquigarrow l_{p-j+\pi} \notin R_e \Rightarrow s_0 \not\rightsquigarrow^{\sigma_e} l_{p-j+\pi}$ [By Definition 5.20]

Since $\rho_q(s_{j-\pi}) = \rho_e(s_0)$, $\rho_q(b) = \rho_e(l_{p-j+\pi})$ and $\mu_q = \mu_e$ (hypotheses about σ_q and σ_e), by Lemma 5.7, $s_{j-\pi} \rightsquigarrow^{\sigma_q} b \Leftrightarrow s_0 \rightsquigarrow^{\sigma_e} l_{p-j+\pi}$. Since $s_0 \not\rightsquigarrow^{\sigma_e} l_{p-j+\pi}$, we conclude that $s_{j-\pi} \not\rightsquigarrow^{\sigma_q} b$.

Thus, we proved that $s_{j-\pi} \rightsquigarrow b \notin R_q$ imples that $s_{j-\pi} \not\rightsquigarrow^{\sigma_q} b$.

Case d: If $a, b \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\}$: In this case, $\rho_c(a) = \rho_q(a)$ and $\rho_c(b) = \rho_q(b)$ (Definition 3.19). If $a \rightsquigarrow b \notin R_q$ then, by rule $\Pi^{\#20}$:

1. [$a \rightsquigarrow b \notin R_c$ and $\tau_q(a) \not\rightsquigarrow \tau_q(b)$] or
2. [$a \rightsquigarrow b \notin R_c$ and $\forall j - \pi \leq p_a < j$, a does not share with s_{p_a} at C] or
3. [$a \rightsquigarrow b \notin R_c$ and $\forall j - \pi \leq p_b < j$, $s_{p_b} \rightsquigarrow b \notin R_c$] or
4. [$a \rightsquigarrow b \notin R_c$ and $\forall j - \pi \leq q_a, q_b < j$, a is definitely aliased to s_{q_a} at C and b is definitely aliased to s_{q_b} at C and no store $l_{q_a-j+\pi}$ nor store $l_{q_b-j+\pi}$ occurs in m_w and $l_{q_a-j+\pi} \rightsquigarrow l_{q_b-j+\pi} \notin R_e$]

We analyze these 4 cases:

1. if $\tau_q(a) \not\rightsquigarrow \tau_q(b)$ then, by Lemma 5.17, $a \not\rightsquigarrow^{\sigma_q} b$.
2. if $a \rightsquigarrow b \notin R_c$ and for each $j - \pi \leq p_a < j$, a does not share with s_{p_a}, then from the latter condition we realize that a does not share with any actual parameter of m at call-time and by Proposition 5.8, $\mathsf{L}_{\sigma_c}(a) \subseteq \mathcal{L}_{\sigma_c}$. By Lemma 5.10, $\rho_c(a) = \rho_q(a)$ entails $\mathsf{L}_{\sigma_c}(a) = \mathsf{L}_{\sigma_q}(a)$. Since $a \rightsquigarrow b \notin R_c$, by Definition 5.20, $a \not\rightsquigarrow^{\sigma_c} b$, i.e. $\rho_c(b) \notin \mathsf{L}_{\sigma_c}(a) = \mathsf{L}_{\sigma_q}(a)$. Moreover, $\rho_q(b) = \rho_c(b)$, hence $\rho_q(b) \notin \mathsf{L}_{\sigma_c}(a) = \mathsf{L}_{\sigma_q}(a)$, and therefore $a \not\rightsquigarrow^{\sigma_q} b$.
3. if $a \rightsquigarrow b \notin R_c$ and for each $j - \pi \leq p_b < j$, $s_{p_b} \rightsquigarrow b \notin R_c$, then from the latter condition we realize that $\rho_c(b)$ is not a location reachable from the actual parameters of m at call-time, and therefore, by Proposition 5.8, $\rho_c(b) \in \mathcal{L}_{\sigma_c}$. Since $a \rightsquigarrow b \notin R_c$, by Definition 5.20, $a \not\rightsquigarrow^{\sigma_c} b$, i.e. $\rho_c(b) \notin \mathsf{L}_{\sigma_c}(a)$, and therefore $\rho_c(b) \notin \mathsf{L}_{\sigma_c}(a) \cap \mathcal{L}_{\sigma_c}$. By Lemma 5.11, $\rho_c(a) = \rho_q(a)$ entails $\mathsf{L}_{\sigma_c}(a) \cap \mathcal{L}_{\sigma_c} = \mathsf{L}_{\sigma_q}(a) \cap \mathcal{L}_{\sigma_c}$, and since $\rho_c(b) = \rho_q(b)$, we have $\rho_q(b) \notin \mathsf{L}_{\sigma_q}(a)$, i.e., $b \not\rightsquigarrow^{\sigma_q} a$.
4. in this case, for every $j - \pi \leq q_a, q_b < j$ we have:

$\rho_c(a) = \rho_c(s_{q_a})$ [a is definitely aliased to $s_{j-\pi+q_a}$ at C]
$\rho_c(s_{q_a}) = \rho_e(l_{q_a-j+\pi})$ [s_{q_a} at C corresponds to $l_{q_a-j+\pi}$ at E and no store $l_{q_a-j+\pi}$ occurs in m_w]
$\rho_c(a) = \rho_q(a)$ [$a \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\}$ and Definition 3.19]
$\rho_c(b) = \rho_c(s_{q_b})$ [b is definitely aliased to s_{q_b} at C]
$\rho_c(s_{q_b}) = \rho_e(l_{q_b-j+\pi})$ [s_{q_b} at C corresponds to $l_{q_b-j+\pi}$ at E and no store $l_{q_b-j+\pi}$ occurs in m_w]
$\rho_c(b) = \rho_q(b)$ [$b \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\}$ and Definition 3.19]
$l_{q_a-j+\pi} \rightsquigarrow l_{q_b-j+\pi} \notin R_e \Rightarrow l_{q_a-j+\pi} \not\rightsquigarrow^{\sigma_e} l_{q_b-j+\pi}$ [By Definition 5.20]

Since, $\rho_q(a) = \rho_e(l_{q_a-j+\pi})$, $\rho_q(b) = \rho_e(l_{q_b-j+\pi})$ and $\mu_q = \mu_e$, by Lemma 5.7, $a \rightsquigarrow^{\sigma_q} b \Leftrightarrow l_{q_a-j+\pi} \rightsquigarrow^{\sigma_e} l_{q_b-j+\pi}$. Since $l_{q_a-j+\pi} \not\rightsquigarrow^{\sigma_e} l_{q_b-j+\pi}$, we conclude that $a \not\rightsquigarrow^{\sigma_q} b$.

■

Lemma 5.41. *The propagation rule* #20 *from Fig. 5.4 satisfies Requirement 4.10. More precisely, the propagation rules for the side-effects arcs is sound at a* void *method return. Namely, let* $w \in [1..n]$ *and consider a side-effect arc from nodes* $\mathsf{C} = \boxed{\mathsf{call}\ m_1 \dots m_n}$ *and* $\mathsf{E} = \boxed{\mathsf{exit@}m_w}$ *to a node* $\mathsf{Q} = \boxed{\mathsf{ins}_q}$ *and its propagation rule* $\Pi^{\#20}$*. Let* τ_c*,* τ_q *and* τ_e *be the static type information at* C*,* Q *and* E*, respectively, and let* d *be 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* $R_c \in \textsc{Reach}_{\tau_c}$ *and* $R_e \in \textsc{Reach}_{\tau_e}$*, we have:*

$$d((\mathit{makescope}\ m_w)(\gamma_{\tau_c}(R_c)) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}(\Pi^{\#20}(R_c, R_e)).$$

Proof. The proof is analogous to the proof of **Case d** of Lemma 4.6. ■

5.4.9 Side-Effects and Exceptional Arcs at Exceptional Ends

In this section we show that our reachability analysis satisfies also Requirement 4.11. The following lemma deals with the exceptional executions of the methods. Namely, the approximation of the reachability information at the catch which captures the exceptional states of the method we are interested in, has to be affected by all possible modifications of the initial memory due to the side-effects of the method. This is the task of the propagation rules of the side-effects arcs. On the other hand, the final approximation of the reachability property at the point of interest (catch) has to be affected by the exceptions launched by the method when it is invoked on a `null` object too. Like in the previous case, the approximation of the reachability information is determined as the union of the two approximations mentioned above, and Lemma 5.42 shows that it is correct.

Lemma 5.42. *The propagation rules* #16 *and* #20 *from Figures 5.3 and 5.4 satisfy Requirement 4.11. More precisely, the propagation rules for the exceptional arcs of the* call *and side-effects arcs are sound when a method throws an exception. Namely, given nodes* $\mathsf{Q} = \boxed{\mathsf{catch}}$*,* $\mathsf{C} = \boxed{\mathsf{call}\ m_1 \dots m_n}$ *and* $\mathsf{E} = \boxed{\mathsf{exception@}m_w}$*, for a suitable* $w \in [1..n]$*, consider an exceptional arc from* C *to* Q *and a side-effect arc from* C *and* E *to* Q*, with their propagation rules* $\Pi^{\#16}$ *and* $\Pi^{\#20}$*, respectively. Let* τ_c*,* τ_q *and* τ_e *be the static type information at* C*,* Q *and* E*, respectively, and let* d *be 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* $R_c \in \textsc{Reach}_{\tau_c}$ *and* $R_e \in \textsc{Reach}_{\tau_e}$*, we have:*

$$d((\mathit{makescope}\ m_w)(\gamma_{\tau_c}(R_c)) \cap \underline{\Xi}_{\tau_q} \subseteq \gamma_{\tau_q}(\Pi^{\#16}(R_c) \cup \Pi^{\#20}(R_c, R_e)).$$

Proof. Consider states $\sigma_c \in \gamma_{\tau_c}(R_c)$, $\sigma_e \in \gamma_{\tau_e}(R_e)$ and $\sigma_q = d((\mathit{makescope}\ m_w)(\sigma)) \in \underline{\Xi}_{\tau_q}$ and let us show that $\sigma_q \in \gamma_{\tau_q}(R_q)$, where $R_q = \Pi^{\#16}(R_c) \cup \Pi^{\#20}(R_c, R_e)$. By Definition 5.20,

$$\begin{aligned}\sigma_q \in \gamma_{\tau_q}(R_q) &\Leftrightarrow \forall a, b \in \mathsf{dom}(\tau_q).\, a \rightsquigarrow^{\sigma_q} b \Rightarrow a \rightsquigarrow b \in R_q \\ &\Leftrightarrow \forall a, b \in \mathsf{dom}(\tau_q).\, a \rightsquigarrow b \notin R_q \Rightarrow a \not\rightsquigarrow^{\sigma_q} b.\end{aligned}$$

In the following we use the following hypotheses: $\mathsf{dom}(\tau_a) = L_a \cup S_a$, where a can be c, e or q, $S_c = \{s_0, \dots, s_j\}$, $S_e = \{s_0\}$, $S_q = \{s_0\}$, $L_q = L_c$ and $\{l_0, \dots, l_{\pi-1}\} \subseteq L_e$, where j and π are number of operand stack elements in $\mathsf{dom}(\tau_c)$ and number of

parameters of method m respectively. By Definition 3.19, σ_c, σ_e and σ_q have to satisfy the following conditions: $\sigma_c = \langle\langle \mathsf{l}_c \parallel v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0\rangle, \mu_c\rangle$, $\sigma_e = \underline{\langle\langle \mathsf{l}_e \parallel \ell\rangle, \mu_e\rangle}$ and $\sigma_q = \langle\langle \mathsf{l}_c \parallel \ell\rangle, \mu_e\rangle$, where ℓ represents the location holding the exception thrown by m_w. Hence, $\underline{\tau_e(s_0) \leq \texttt{Throwable}}$ and $\tau_q(s_0) \leq \texttt{Throwable}$. Let a and b be two arbitrary variables from $\mathsf{dom}(\tau_q)$ and suppose that $a \rightsquigarrow^{\sigma_q} b$. We show that in that case $a \rightsquigarrow b \in R_q$. We distinguish the following cases:

If $a = s_0$ and $b \neq s_0$: Since $s_0 \rightsquigarrow^{\sigma_q} x$ then, by Lemma 5.7 and Definition 5.12, we have that $\tau_q(s_0) \rightsquigarrow \tau_q(b)$, i.e., $\tau_q(b) \in \mathsf{T}(\tau_q(s_0))$. Since $\tau_q(s_0) \leq \texttt{Throwable}$ we have, by Lemma 5.15, $\mathsf{T}(\tau_q(s_0)) \subseteq \mathsf{T}(\texttt{Throwable})$, and therefore $\tau_q(b) \in \mathsf{T}(\texttt{Throwable})$, i.e., $\texttt{Throwable} \rightsquigarrow \tau_q(b)$. Moreover, $x \notin S_q$, hence $a \rightsquigarrow b = s_0 \rightsquigarrow b \in \Pi^{\#16}(R_c) \subseteq R_q$.

If $a \neq s_0$ and $b = s_0$: Since $a \rightsquigarrow^{\sigma_q} s_0$ then, by Lemma 5.7, $\tau_q(a) \rightsquigarrow \tau_q(s_0)$. Moreover, $\tau_q(s_0) \leq \texttt{Throwable}$ and, by Lemma 5.13, $\tau_q(a) \rightsquigarrow \texttt{Throwable}$. Since $a \notin S_q$, we have $a \rightsquigarrow b = a \rightsquigarrow s_0 \in \Pi^{\#16}(R_c) \subseteq R_q$.

If $a = b = s_0$: In this case $a \rightsquigarrow b = s_0 \rightsquigarrow s_0$ trivially belongs to $\Pi^{\#16} \subseteq R_q$.

If $a, b \in \mathsf{dom}(\tau_q) \setminus \{s_0\}$: In this case we suppose that $a \rightsquigarrow b \notin R_q$, and we show that $a \not\rightsquigarrow^{\sigma_q} b$. The proof is analogous to the proof of **Case d** of Lemma 4.6. ■

5.4.10 Conclusion

In this section we have shown that all the requirements provided in Chapter 4 are satisfied by the abstract domain Reach introduced in Section 5.3.1 and by the propagation rules introduced in Section 5.3.2. This fact allows us to assert the following two results.

Theorem 5.43. *There exists the least solution to the reachability analysis introduced in this chapter.*

Proof. This proof directly follows from the results obtained in Section 4.5, where instead of a generic abstract domain

$$\mathsf{A}_{\tau_k} = \langle \mathcal{A}_{\tau_k}, \sqsubseteq, \sqcup, \sqcap, \top_{\tau_k}, \bot_{\tau_k} \rangle,$$

for a type environment τ_k corresponding to a node k, we use the abstract domain

$$\textsc{Reach}_{\tau_k} = \langle \wp(\mathsf{dom}(\tau_k) \times \mathsf{dom}(\tau_k)), \subseteq, \cup, \cap, \wp(\mathsf{dom}(\tau_k) \times \mathsf{dom}(\tau_k)), \emptyset \rangle,$$

defined in this chapter. Theorem 4.11 shows that when Requirements 4.1 and 4.3 are satisfied, then there exists the least solution to the system of constraints constructed in Section 4.5, representing the actual static analysis of interest. Lemmas 5.31 and 5.34 show that Requirements 4.1 and 4.3 are satisfied by the instantiation of the framework concerning the reachability analysis, hence hypotheses of Theorem 4.11 are satisfied and we can use its results. ■

Example 5.44. In Figure 5.7 we give the least solution to the system of constraints introduced in Example 5.30 (Fig. 5.6) and concerning the ACG from Fig. 5.5. □

Finally, we can state that our reachability analysis is sound, i.e., at each program point the set of reachability pairs obtained by our analysis represent an over-approximation of the actual reachability information available at that point.

NODE	SOLUTION OF REACHABILITY APPROXIMATION
A	$\left\{ \begin{array}{c} l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_3 \rightsquigarrow l_3, l_1 \rightsquigarrow s_3, l_3 \rightsquigarrow s_2, s_2 \rightsquigarrow l_3, \\ s_0 \rightsquigarrow s_0, s_0 \rightsquigarrow s_1, s_1 \rightsquigarrow s_0, s_1 \rightsquigarrow s_1, s_2 \rightsquigarrow s_2, s_3 \rightsquigarrow s_3 \end{array} \right\}$
1	$\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$
2	$\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, s_0 \rightsquigarrow l_0, s_0 \rightsquigarrow s_0\}$
3	$\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$
4	$\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, s_0 \rightsquigarrow l_0, s_0 \rightsquigarrow s_0\}$
5	$\left\{ \begin{array}{c} l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, \\ l_1 \rightsquigarrow s_1, s_0 \rightsquigarrow l_0, s_1 \rightsquigarrow l_1, s_0 \rightsquigarrow s_0, s_1 \rightsquigarrow s_1 \end{array} \right\}$
6	$\{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$
7	$\left\{ \begin{array}{c} l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, \\ l_0 \rightsquigarrow s_0, s_0 \rightsquigarrow l_0, s_0 \rightsquigarrow l_1, s_0 \rightsquigarrow s_0 \end{array} \right\}$
8	$\left\{ \begin{array}{c} l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, \\ l_2 \rightsquigarrow s_1, s_0 \rightsquigarrow l_0, s_0 \rightsquigarrow l_1, s_1 \rightsquigarrow l_2, s_0 \rightsquigarrow s_0, s_1 \rightsquigarrow s_1 \end{array} \right\}$
9	$\{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_0 \rightsquigarrow l_2, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$
10	
11	$\{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, s_0 \rightsquigarrow s_0\}$
12	
13	
B	$\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_1 \rightsquigarrow s_0, l_1 \rightsquigarrow l_3, l_3 \rightsquigarrow l_3, s_0 \rightsquigarrow l_3, s_0 \rightsquigarrow s_0\}$
C	

Fig. 5.7. The solution of the constraint system from Fig. 5.6

Theorem 5.45. *Let* $\langle b_{first(\texttt{main})} \parallel \xi \rangle \Rightarrow^* \langle \boxed{\begin{array}{l}\mathsf{ins} \\ rest\end{array}}^{\rightarrow b_1}_{\rightarrow b_m} \parallel \sigma \rangle :: a$ *be the execution of our operational semantics, from the block* $b_{first(\texttt{main})}$ *starting with the first bytecode instruction of method* `main`*,* ins_0*, and an initial state* $\xi \in \Sigma_{\tau_0}$ *(containing no reachability except the* `this` *object which reaches itself), to a bytecode instruction* ins *and assume that this execution leads to a state* $\sigma \in \Sigma_\tau$*, where* τ_0 *and* τ *are the static type information at* ins_0 *and* ins*, respectively. Moreover, let* $R_0 = \{l_0 \rightsquigarrow l_0\} \in \text{REACH}_{\tau_0}$*, and let* $R \in \text{REACH}_\tau$ *be the reachability approximation at* ins*, as computed by our reachability analysis starting from* R_0*. Then,* $\sigma \in \gamma_\tau(R)$ *holds.*

Proof. This proof directly follows from Theorem 4.14 and the fact that the requirements provided in Chapter 4 are satisfied, which has been shown in this section (Lemmas 5.31-5.42). ■

Therefore, the reachability analysis instantiated in the parameterized general framework for constraint-based static analyses of Java bytecode programs is sound and has the least solution.

5.5 Experimental Evaluation of the Reachability Analysis

We have implemented our reachability analysis inside the Julia analyzer for Java and Android [4]. It is a commercial tool developed by Julia Srl, a spin-off company of the University of Verona. In the next sections we describe the tool and the experiments that we have performed in order to evaluate our reachability analysis.

5.5.1 The Julia Analyzer

Julia is a static analyzer for bytecode, completely written in Java, that includes classes for the definition of denotational (bottom-up) analyses, constraint-based analyses and automaton-based analyses.

Denotational analyses define a functional abstract behavior (*denotation*) for each single bytecode instruction and compose such behaviors in a bottom-up way, computing fixpoints in order to analyze loops and recursion. An example is the nullness analysis in [84]. Their strength is that these analyses are fully context-sensitive, since the denotation of a method is a function from its context at call-time to its context at return-time; however, it is difficult to provide context-sensitive approximations for the fields of the objects, since the analyses become computationally too expensive. Binary decision diagrams [26] are typically used to implement denotations and Julia provides support for this choice.

Constraint-based analyses follow the approach used, for instance, in this thesis. A system of constraints is built from the program under analysis (and the libraries that it uses). Nodes might stand for program points, as in this article, when the abstract interpretation abstracts states (see in our case Definition 5.20). They might also stand for local variables, stack elements, fields or return values, when the abstract interpretation abstracts values rather than states (examples are [63, 86]). Or they might stand for whole methods and constructors, as in the case of side-effect analysis. Moreover, the approximation at a node can be the union of the approximations of the incoming arcs (as in this chapter) or their intersection. In the first case, we deal with a possible analysis (an over-approximation of the property under analysis is computed); in the second case, we deal with a definite analysis (an under-approximation of the property under analysis is computed). In all cases, Julia provides standard implementations of the construction of a system of constraints, that can be personalized by subclassing, if needed. The elements of the abstract domain (in our case, sets of ordered pairs of variables) are represented through bitsets of singletons, in order to make set operations very fast (they become bitwise operations over arrays of Java 64 bits `long`s) and keep the memory footprint small (singletons are created once; bitsets are very compact). The fixpoint algorithm that finds a solution, in two versions for possible and definite analysis, is implemented in Julia through a working-set, demand-driven approach: the arcs of the constraint are put in a stack and processed one at a time; when the approximation of a node changes, all its outgoing arcs are added again to the stack, until stabilization. This means, in particular, that the programmer of a static analysis does not need to care about the fixpoint algorithm or the bitset implementation, since the infrastructure is available, debugged and optimized once and for all inside Julia. Its code is shared by all constraint-based analyses.

Automaton-based static analyses abstract execution traces into states of a finite state automaton. The automaton is *executed* from the initial bytecode instruction of the program and each bytecode instruction induces a state transition in the automaton. Possible states of the automaton at a program point are an abstraction of all the execution paths that might lead to that point. The advantage of this approach is that one can easily abstract traces rather than states and the analysis has a very efficient and relatively simple implementation. An example is the determination of program points where a given array has been fully initialized [62, 65].

The reachability analysis of this article uses preliminary, supporting analyses, namely definite aliasing and possible sharing analysis. They are both implemented as constraint-based analyses themselves and computed before our reachability analysis starts. That is,

we do not use a reduced product of more analyses, but implement a sequence of analyses. The result of reachability is then used by client analyses, namely, side-effects, field initialization, cyclicity and path-length analysis. The propagation of sharing information is described in [82].

Side-effects analysis collects the fields that might be read or modified by a method or constructor and that were already allocated before the call to that method or constructor. It is a constraint-based analysis where the node for each method or constructor collects the fields explicitly read or modified. Arcs propagate these sets from callees to callers. Reachability improves the precision of the side-effects analysis (Section 5.1). Field initialization determines the fields f that are always initialized by all constructors of their defining class, before being read. Hence, the fact that `null` is the default value for reference fields becomes irrelevant for f, since that value is never read. This is important, for instance, for a subsequent nullness analysis. This field initialization analysis is implemented through a dataflow algorithm in Julia, that collects the fields of `this` definitely written at each program point of the constructors and the fields of `this` possibly read at the same program points. This algorithm is described in [65, 84, 85]. It exploits the available reachability information (Section 5.1). Cyclicity analysis is a denotational analysis, where each variable is approximated through a Boolean variable stating if it might be cyclical or not, and exploits reachability at field updates (Section 5.1). Path-length analysis is denotational, again, and uses polyhedra or simpler domains to represent the size of the numerical values bound to variables of primitive type, or the maximal height of the data structures bound to variables of reference type. Reachability helps here by restricting the set of variables whose path-length might be affected by a field update (Section 5.1). More detail in [87].

The nullness analyzer of Julia is a sequential composition of many analyses, through an oracle-based semantics for the nullness of the fields. Its detailed description can be found in [85]. It uses a denotational nullness analysis for local and stack variables [84] combined with an array initialization analysis that guarantees that the default value (`null`) for the elements of some arrays of reference type is never read [62,65]. It is also combined with constraint-based analyses for tracking arrays, collections or iterators whose elements have only been assigned to non-`null` values, and for tracking the expressions that definitely evaluate to non-`null` values (for instance, expressions explicitly compared against `null` or already dereferenced in a previous statement, identified through the constraint-based analysis in [63]). All these analyses exploit reachability, sharing and side-effects to restrict the effects of a field update on variables distinct from its receiver, or of a method call on the variables of the caller. For instance, a method call might invalidate, by side-effects, the fact that the evaluation of an expression is a non-`null` value.

The termination analyzer of Julia is based on path-length, used to determine if loops or recursion happen on integers or data structures of strictly decreasing (yet positive) size. Here, again, we use preliminary reachability and sharing information in order to restrict the effects of field updates and method calls to the path-length of the variables. The detailed definitions are in [87]. Moreover, termination analysis often uses expressions as symbolic constants (for instance, expressions used as upper bounds of loops) and side-effect analysis provides the information needed to be sure that such expressions keep their value unchanged across iterations and can hence be used as actual, constant upper bounds to loops.

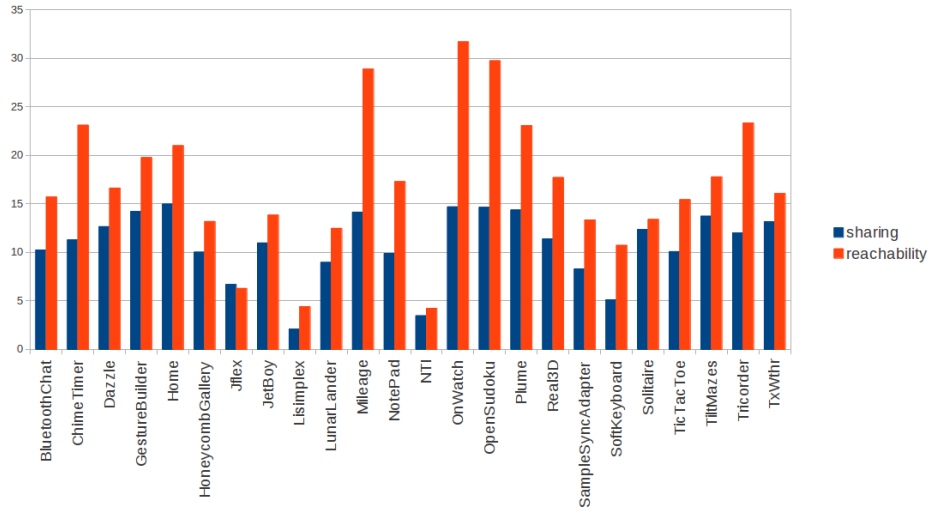


Fig. 5.8. Run-times in seconds of sharing and reachability analyses of our sample programs.

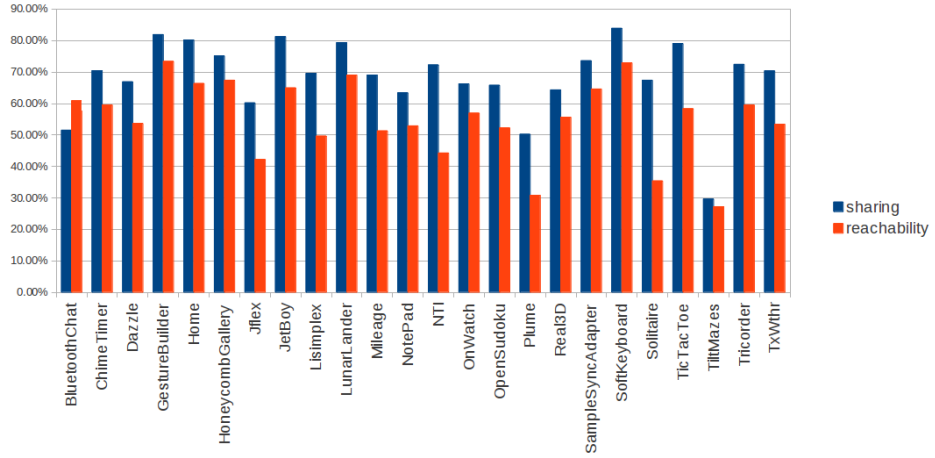


Fig. 5.9. Precision of sharing and reachability analyses of our sample programs w.r.t. the reachability property. Here, precision is the ratio of ordered pairs of distinct variables $\langle v, w \rangle$ s.t. the analysis concludes that v might reach w, over the total number of ordered pairs of variables of reference type: the lower the ratio, the higher the precision (this ratio never reaches 0% in practice, since real-life programs contain reachability). For sharing, we assume that v might reach w if v might share with w. Both for sharing and reachability analysis, if the static type of v does not reach the static type of w (Def. 5.12), the ordered pair (v, w) is not counted in this figure, since in that case it is statically known that the value held in v will never be able to reach the value held in w (Lemma 5.17).

program	source lines	analyzed lines	reach. analysis time	reach. analysis prec	precision of side-effects analysis without reach.	precision of side-effects analysis with reach.	precision of field initial. analysis without reach.	precision of field initial. analysis with reach.	cyclicity analysis without reach.	cyclicity analysis with reach.
BluetoothChat	616	84415	21.26	56.01%	645.99	540.23	2185	2325	12.85%	12.85%
ChimeTimer	1090	89565	23.39	47.04%	730.68	618.08	2348	2486	13.54%	13.54%
Dazzle	1791	77828	24.23	46.99%	309.89	225.96	2417	2447	22.19%	22.19%
GestureBuilder	502	84346	23.90	64.11%	667.70	557.52	2162	2282	16.57%	16.57%
Home	870	87413	18.54	55.78%	693.80	584.01	2274	2415	10.89%	10.89%
HoneycombGallery	948	71558	16.71	23.84%	333.25	242.32	2131	2175	33.33%	33.33%
JFlex	7681	40779	7.19	39.59%	357.59	243.89	1092	1146	33.67%	33.67%
JetBoy	839	65174	16.37	64.54%	281.48	198.71	2173	2202	11.79%	11.79%
Lisimplex	768	49303	16.26	47.98%	637.69	347.96	1356	1433	14.13%	14.13%
LunarLander	538	57675	14.92	66.40%	270.87	191.07	1880	1911	18.11%	18.11%
Mileage	5877	104009	32.12	43.73%	959.30	804.98	2636	2794	25.45%	25.45%
NotePad	705	73742	17.96	36.59%	293.57	218.17	2108	2139	37.50%	37.50%
NTI	2372	13486	2.44	47.90%	24.11	13.51	465	467	32.59%	32.59%
OnWatch	6295	112423	29.59	41.00%	1299.51	796.89	3232	3399	32.59%	32.59%
OpenSudoku	5877	90810	40.68	44.81%	440.36	344.92	2622	2660	22.57%	22.57%
Plume	8586	43637	17.75	24.17%	186.31	126.71	1316	1335	57.11%	57.11%
Real3D	1228	74350	17.81	43.55%	497.94	400.73	2093	2189	36.43%	36.43%
SampleSyncAdapter	978	65971	18.48	34.59%	328.80	235.68	2111	2142	42.77%	42.77%
SoftKeyboard	703	58088	10.96	51.90%	174.01	116.96	2112	2131	11.21%	11.21%
Solitaire	3905	62065	18.67	32.23%	243.19	166.57	1957	1982	50.06%	50.06%
TicTacToe	607	59160	13.40	58.56%	228.27	154.35	1919	1943	20.73%	20.73%
TiltMazes	1853	89653	21.14	15.66%	650.45	562.57	2313	2454	71.57%	71.57%
Tricorder	5317	98389	26.69	46.39%	783.59	663.23	2806	2942	33.91%	33.91%
TxWthr	2024	74537	16.97	48.33%	309.24	229.79	2220	2258	15.39%	15.39%
average precision				**45.07%**	**472.81**	**361.86 (-23.47%)**	**2080.33**	**2152.37 (+3.46%)**	**26.84%**	**26.84% (+0.00%)**

Fig. 5.10. The effects of reachability analysis on the precision of side-effects, field initialization and cyclicity analyses. *Source lines* counts non-comment non-blank lines of codes. *Analyzed lines* includes the portion of `java.*`, `javax.*` and `android.*` libraries analyzed with each program and is a more faithful measure of the analyzed codebase. Times are in seconds. For side-effects analysis, precision is the average number of fields modified or read by a method or constructor: the lower the numbers, the better the precision. For field initialization analysis, precision is the number of fields of reference type proven to be always initialized before being read, in all constructors of their defining class: the higher the numbers, the better the precision. For cyclicity analysis, precision is the average number of variables of reference type proven to hold a non-cyclical data structure; the higher the numbers, the better the precision

program	null. without reach.			null. with reach.			term. without reach.			term. with reach.		
	time	ws	prec	time	ws	prec	time	ws	prec	time	ws	prec
BluetoothChat	368.43	**22*****	93.65%	301.31	**19*****	94.23%	158.96	2	33.33%	141.78	2	33.33%
ChimeTimer	343.01	4	98.36%	360.28	4	98.36%	178.87	1	83.33%	183.81	1	83.33%
Dazzle	223.16	26	97.99%	220.78	26	97.99%	120.34	0	100.00%	126.07	0	100.00%
GestureBuilder	261.25	16	92.37%	288.51	16	92.37%	153.33	0	100.00%	151.83	0	100.00%
Home	314.66	27	94.27%	312.55	27	94.27%	166.98	8	38.46%	163.39	8	38.46%
HoneycombGallery	177.32	12	97.79%	179.90	12	97.79%	105.96	0	100.00%	101.47	0	100.00%
JFlex	87.06	71	97.03%	86.10	71	97.03%	300.84	66	53.52%	321.03	66	53.52%
JetBoy	138.99	20**	97.42%	140.64	20**	97.42%	85.91	3	57.14%	85.38	3	57.14%
Lisimplex	251.09	20**	96.94%	202.76	20**	96.94%	160.07	9	70.97%	153.36	9	70.97%
LunarLander	118.75	4	99.30%	121.25	4	99.30%	72.49	3*	0.00%	68.41	3*	0.00%
Mileage	503.90	**102**	97.40%	501.02	**95**	97.67%	387.68	12	69.23%	381.99	12	69.23%
NotePad	194.52	**18**	96.50%	199.19	**17**	96.50%	103.64	0	100.00%	101.49	0	100.00%
NTI	14.06	12	98.93%	16.15	12	98.93%	43.70	70	36.94%	43.53	70	36.94%
OnWatch	898.36	**74**	97.91%	518.55	**65**	98.18%	385.00	6	86.96%	371.32	6	86.96%
OpenSudoku	284.30	124*	95.93%	286.72	124*	95.93%	458.01	6	90.32%	467.34	6	90.32%
Plume	106.67	**59**	98.82%	116.75	**58**	98.83%	208.81	86	60.00%	187.92	86	60.00%
Real3D	203.62	19*	98.14%	195.76	19*	98.14%	116.42	2	60.00%	112.22	2	60.00%
SampleSyncAdapter	156.31	3	99.51%	152.45	3	99.51%	91.90	2	60.00%	89.61	2	60.00%
SoftKeyboard	104.21	**14**	95.78%	103.83	**13**	95.94%	70.45	0	100.00%	67.96	0	100.00%
Solitaire	153.51	63	92.59%	147.54	63	92.59%	207.09	11	86.08%	203.92	11	86.08%
TicTacToe	115.38	0	100.00%	118.27	0	100.00%	79.69	1	85.71%	78.02	1	85.71%
TiltMazes	281.43	**18**	98.20%	276.54	**14**	98.83%	188.56	1	88.89%	174.63	1	88.89%
Tricorder	415.17	**54**	98.29%	407.51	**52**	98.41%	252.25	12	80.33%	257.36	12	80.33%
TxWthr	200.16	48	97.85%	191.88	48	97.85%	109.76	6	70.00%	105.08	6	70.00%
total run-times	**5915.32**			**5456.24 (-7.77%)**			**4206.71**			**4138.92 (-1.62%)**		
total warnings	**830**			**802 (-3.38%)**			**307**			**307 (+0.00%)**		

Fig. 5.11. Our experiments with the nullness and termination tools of Julia. Times are in seconds. For nullness analysis, *ws* counts the warnings issued by Julia (possible dereference of `null`, possibly passing `null` to a library method) and *prec* reports its precision, as the ratio of the dereferences proved safe over their total number (100% is the maximal precision). For termination analysis, *ws* counts the warnings issued by Julia (constructors or methods possibly diverging) and *prec* reports its precision, as the ratio of the constructors or methods proved to terminate over the total number of constructors or methods containing loops or recursive (100% is the maximal precision). Asterisks stand for actual bugs in the programs. Boldface highlights the cases where reachability improves the precision of the tools

5.5.2 Sample Programs

We have analyzed a set of sample programs. Most of our benchmarks are Android applications: Mileage, OpenSudoku, Solitaire and TiltMazes [11]; ChimeTimer, Dazzle, OnWatch and Tricorder [5]; TxWthr [10]. There are also some Java programs: JFlex is a lexical analyzers generator [3]; Plume is a library by Michael D. Ernst [8]; NTI is a nontermination analyzer by Étienne Payet [6]; Lisimplex is a numerical simplex implementation by Ricardo Gobbo [7]. The others are sample programs taken from the Android 3.1 distribution by Google. and are bundled with the Android SDK Tools r12 [1].

Experiments have been performed on a Linux quad-core Intel Xeon machine running at 2.66GHz, with 8 gigabytes of RAM.

5.5.3 Sharing vs. Reachability Analysis

Figure 5.8 shows that reachability analysis is in general more expensive than sharing analysis. Moreover, reachability analysis needs to be supported by a preliminary sharing analysis. The extra cost of reachability analysis is compensated by its increased precision when it comes to compute reachability information itself. To prove this claim, in Figure 5.9 we have built reachability analysis from sharing analysis, by assuming that a variable v might reach a distinct variable w whenever v and w might possibly share, according to the results of sharing analysis. We have compared these reachability information with that gathered through the reachability analysis computed as described in this article. The latter yields around 20% fewer reachability pairs than reachability analysis built from sharing. Fewer pairs, here, mean better precision.

Although reachability analysis is more expensive than sharing analysis, we are going to show that it actually reduces the cost in time of larger static analyses, where it is used as a supporting analysis (Section 5.5.5). This is because its extra precision simplifies the subsequent analyses. Moreover, in the overall economy of a parallel static analyzer such as Julia, the few extra seconds required by reachability analysis are a small fraction of the time required by nullness or termination analyses, that use reachability as a supporting analysis and are greatly benefited by any increase in precision of the latter.

5.5.4 Reachability vs. Shape Analysis

Reachability might be abstracted from a more concrete analysis, such as some flavor of shape analysis. The Julia analyzer does not include any shape analysis and there is no plan in that direction. In particular, we are not aware of any static shape analysis for Java bytecode that deals with exceptional paths. There are dynamic shape analyses for Java, such as for instance [47, 72]. But dynamic analyses are only sound w.r.t. the execution traces that are generated at run-time and analyzed. As a consequence, they cannot be taken as basis for a sound static reachability analysis. We are aware of two static shape analyses for Java. The first [30] is intraprocedural only; experiments do not report its cost in time. The second [54] is able to analyze interprocedural Java programs; exceptional paths are not mentioned. Experiments reported in that article show that the analysis of a program of $3,705$ statements requires 35.11 seconds; libraries have not been included in the analysis. Our reachability analysis analyzes $112,423$ statements in 32 seconds (Figures 5.8 and 5.10, see the case of `OnWatch`). Libraries are analyzed along the application. If one

considers that sharing is needed before reachability, the total time of our analysis amounts to 47 seconds, but the analyzed code base of 112,423 statements is 30 times larger than their 3,705 statements. There is no report on the precision of the analysis in [54] w.r.t. reachability information, but the major difference in the computational cost of the two analyses is apparent. It is true that our hardware is multicore, so potentially faster than that used in [54], but sharing and reachability analyses are performed sequentially in Julia, so that only one core is used for them.

5.5.5 Effects of Reachability Analysis on Other Analyses

We verify here whether reachability analysis actually improves the precision of side-effects, field initialization and cyclicity, as hinted in Section 5.1. We also verify if the extra reachability information improves the precision of the nullness and termination checking tools available in Julia, that use side-effects, field initialization, cyclicity and path-length as (some of their) supporting analyses. We do not have any measure of precision for path-length analysis, so we do not evaluate its improvements directly but only as a component of the termination checking tool. To reach these goals, we have analyzed our sample programs with reachability analysis turned off (hence relying on sharing analysis as an approximation of reachability analysis) and then on.

Figure 5.10 shows that reachability analysis improves the precision of the side-effects analysis and has positive effects on field initialization as well. Instead, cyclicity analysis seems unaffected. Sharing analysis is always used in these experiments, both when we use reachability information and when we do not compute it. Thus, this figure shows the importance of having also reachability information instead of just sharing information.

Figure 5.11 presents our experiments with the nullness and termination tools of Julia and reports their run-time, including reachability analysis. In 8 cases over 24, the extra reachability information improves the precision of the nullness checking tool. But this never happens for termination, consistently with the fact that cyclicity is not improved (Figure 5.10). This is because the methods of the programs that we have analyzed terminate since they perform loops over numerical counters or iterators. There is no complex case of recursion over data structures dynamically allocated in memory (lists or trees) where cyclicity would help. To investigate further the case of termination analysis, we have applied Julia to the set of (very tiny) programs used for the international termination competition[2] that is performed every year. Those programs, although small and often unrealistic, are nevertheless interesting since the proof of their termination often requires non-trivial arguments, also related to objects dynamically allocated in memory. Over a total of 164 test programs, the reachability information allows Julia to prove the termination of six more tests: LinkedList, List, ListDuplicate, PartitionList, Test5 and Test6, by supporting a more precise cyclicity and path-length analysis.

For both nullness and termination checking, the presence of reachability analysis actually reduces the total run-time of the tools. This is because reachability helps subsequent analyses, in particular side-effects analysis, and prevents them from generating too much spurious information. For instance, side-effects analysis computes much smaller sets of affected fields per method (Figure 5.10, compare the 5th and the 6th columns).

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

6

Definite Expression Aliasing Analysis

In this chapter we define a novel static analysis for Java bytecode, called *definite expression aliasing*. It infers, for each variable v at each program point p, a set of expressions whose value at p is equal to the value of v at p, for every possible execution of the program. Namely, it determines which expressions *must* be aliased to local variables and stack elements of the Java Virtual Machine. This is a useful piece of information for an inter-procedural static analyzer, such as Julia, since it can be used to refine other analyses at conditional statements or assignments. This analysis is formalized like an instantiation of the general parameterized framework for constraint-based static analyses introduced in Chapter 4. The latter allows us to easily prove our new analysis sound. Our definite expression aliasing analysis has been implemented inside the Julia analyzer, and we show its benefits for nullness and termination tools of Julia.

The present chapter is based on the work already published in [63] and its extended version [60].

6.1 Introduction

Static analyses infer properties of computer programs and and prove the absence of some classes of bugs inside those programs. Modern programming languages are, however, very complex. Static analysis must cope with that complexity and remain precise enough to be of practical interest. This is particularly true for low-level languages such as Java bytecode [53], whose instructions operate on stack and local variables, which are typically aliased to expressions. Consider, for instance, the method `onOptionsItemSelected` in Fig. 6.1, taken from the Google's `HoneycombGallery` Android application. The statement `if (mCamera!=null)` at line 4 is compiled into the following bytecode instructions:

```
aload_0
getfield mCamera:Landroid/hardware/Camera;
ifnull [go to the else branch]
[then branch]
```

Bytecode `ifnull` checks whether the topmost variable of the stack, *top*, is `null` and passes control to the opportune branch. A static analysis that infers non-`null` variables

```
1  public boolean onOptionsItemSelected(MenuItem item) {
2   switch (item.getItemId()) {
3   case R.id.menu_switch_cam:
4     if (mCamera != null) {
5       mCamera.stopPreview();
6       mPreview.setCamera(null);
7       mCamera.release();
8       mCamera = null;
9     }
10    mCurrentCamera = (mCameraCurrentlyLocked+1)%mNumberOfCameras;
11    mCamera = Camera.open(mCurrentCamera);
12    mCameraCurrentlyLocked = mCurrentCamera;
13    mCamera.startPreview();
14    return true;
15  case ....
16   ....
17 }
```

Fig. 6.1. A method of the `CameraFragment` class by Google

can, therefore, conclude that *top* is non-`null` at the `[then branch]`. But this information is irrelevant: *top* gets consumed by the `ifnull` and disappears from the stack. It is, instead, much more important to know that *top* was a definite alias of the field `mCamera` of local 0, i.e., of `this.mCamera`, because of the previous two bytecodes (local 0 stands for `this`). That observation is important at the subsequent call to `mCamera.stopPreview()` at line 5, since it allows us to conclude that `this.mCamera` is still non-`null` there: line 5 is part of the then branch starting at line 4 and we proved that *top* (definitely aliased to `this.mCamera`) is non-`null` at that point.

As another example of the importance of definite aliasing for static analysis, suppose that we statically determined that the value returned by the method `open` and written in `this.mCamera` at line 11 is non-`null`. The compilation of that assignment is:

```
aload_0
aload_0
getfield mCurrentCamera:I
invokestatic android/hardware/Camera.open:(I)Landroid/hardware/Camera;
putfield mCamera:Landroid/hardware/Camera;
```

and the `putfield` bytecode writes the top of the stack (`open`'s returned value) into the field `mCamera` of the underlying stack element s. Hence s.`mCamera` becomes non-`null`, but this information is irrelevant, since s disappears from the stack after the `putfield` is executed. The actual useful piece of information at this point is that s was a definite alias of expression `this` (local variable 0) at the `putfield`, which is guaranteed by the first `aload_0` bytecode. Hence, `this.mCamera` becomes non-`null` there, which is much more interesting for the analysis of the subsequent statements.

The previous examples show the importance of definite expression aliasing analysis for nullness analysis. However, the former is useful for other analyses as well. For in-

stance, consider the termination analysis of a loop whose upper bound is the return value of a function call:

```
for (i = 0; i < max(a, b); i++)
  body
```

In order to prove its termination, a static analyzer needs to prove that the upper bound `max(a, b)` remains constant during the loop. However, in Java bytecode, that upper bound is just a stack element and the static analyzer must rather know that the latter is a definite alias of the return value of the call `max(a, b)`.

These examples show that it is important to know which expressions are *definitely* aliased to stack and local variables of the Java Virtual Machine (JVM) at a given program point. In this chapter, we introduce a static analysis called *definite expression aliasing analysis*, which provides, for each program point p and each variable v, a set of expressions E such that the values of E and v at point p coincide, for every possible execution path. We call these expressions *definite expression aliasing information*. In general, we want to deal with relatively complex expressions (e.g., a field of a field of a variable, the return value of a method call, possibly non-pure, and so on). We show, experimentally, that this analysis supports nullness and termination analyses of our tool Julia, but this chapter is only concerned with the expression aliasing analysis itself. Our definite expression aliasing analysis is an instantiation of the parameterized framework for constraint-based static analyses of Java bytecode programs introduced in Chapter 4. We introduce the *abstract domain for definite expression aliasing analysis* Alias and the *propagation rules* representing an *abstract semantics* of bytecode instructions over Alias. Moreover, we show that Alias and the propagation rules mentioned above satisfy Requirements 4.1-4.11 introduced in Chapter 4., which entails the soundness of our approach.

We opt for a semantical analysis rather than simple syntactical checks. For instance, in Fig. 6.1, the result of the analysis must not change if we introduce a temporary variable `temp = this.mCamera` and then check whether `temp != null`: it is still the value of `this.mCamera` that is compared to `null` there. Moreover, since we analyze Java bytecode, a semantical approach is important in order to be independent from the specific compilation style of high-level expressions and be able to analyze obfuscated code (for instance, malware) or code not decompilable into Java (for instance, not organized into scopes).

The rest of the chapter is organized as follows. Section 6.2 defines the notion of alias expressions, their non-standard evaluation and specifies which bytecode instructions might modify the value of these expressions. Section 6.3 introduces our definite expression aliasing analysis and shows how the parameters of the general constraint-based framework introduced in Chapter 4 are instantiated to deal with that analysis: we define the abstract domain Alias and the propagation rules for definite expression aliasing analysis. Section 6.4 shows that the parameters mentioned above satisfy the requirements imposed by the framework, which entails the soundness of our static analysis. Section 6.5 introduces an implementation of our analysis, discus its improvements and shows the results of an application of our analysis to many real-life benchmarks, its precision and the way it affects the other analyses performed by our static analyzer Julia.

6.2 Alias Expressions

Tho goal of this chapter is to define a static analysis which infers, for each program point, and each variable available at that point, a set of expressions which must have the same value assigned to the variable, for every possible execution of the program. In order to reach that goal, we choose a subset of all possible expressions available at each program point, and we deal with these expressions only.

In this section, we define our expressions of interest (Definition 6.1), their *non-standard evaluation* (Definition 6.7), which might modify the content of some memory locations and we introduce the notion of *alias expression* (Definition 6.9). Moreover, we specify in which cases *a bytecode instruction might affect the value of an expression* (Definition 6.11), and when *the evaluation of an expression might modify a field* (Definition 6.12).

Definition 6.1 (Expressions). *Given $\tau \in \mathcal{T}$, let $\mathcal{F}_\tau$ and $\mathcal{M}_\tau$ respectively denote the sets of the names of all possible fields and methods of all the objects available in Σ_τ. We define the set of expressions over* $\mathsf{dom}(\tau)$*:*

$$
\begin{array}{rll}
\mathbb{E}_\tau \ni \mathsf{E} ::= & n & \textit{constants} \\
| & v & \textit{variables} \\
| & \mathsf{E} \oplus \mathsf{E} & \textit{arithmetical operations} \\
| & \mathsf{E}.f & \textit{field accesses} \\
| & \mathsf{E}.\mathtt{length} & \textit{array lengths} \\
| & \mathsf{E}[\mathsf{E}] & \textit{array elements} \\
| & \mathsf{E}.m(\mathsf{E}, \ldots) & \textit{results of method invocations},
\end{array}
$$

where $n \in \mathbb{Z}$, $v \in \mathsf{dom}(\tau)$, $\oplus \in \{+, -, \times, \mathsf{div}, \%\}$, $f \in \mathcal{F}_\tau$ and $m \in \mathcal{M}_\tau$.

For every expression E we define the set of its sub-expressions.

Definition 6.2 (Sub-expressions). *Given a type environment $\tau \in \mathcal{T}$ and an expression $\mathsf{E} \in \mathbb{E}_\tau$, we define* $\mathsf{subExp}(\mathsf{E}) \subseteq \mathbb{E}_\tau$*, the set of* E*'s* sub-expressions*, according to the form of* E*:*

E	$\mathsf{subExp}(\mathsf{E})$
n	$\{n\}$
v	$\{v\}$
$\mathsf{E}_1 \oplus \mathsf{E}_2$	$\{\mathsf{E}_1 \oplus \mathsf{E}_2\} \cup \mathsf{subExp}(\mathsf{E}_1) \cup \mathsf{subExp}(\mathsf{E}_2)$
$\mathsf{E}_1.f$	$\{\mathsf{E}_1.f\} \cup \mathsf{subExp}(\mathsf{E}_1)$
$\mathsf{E}_1.\mathtt{length}$	$\{\mathsf{E}_1.\mathtt{length}\} \cup \mathsf{subExp}(\mathsf{E}_1)$
$\mathsf{E}_1[\mathsf{E}_2]$	$\{\mathsf{E}_1[\mathsf{E}_2]\} \cup \mathsf{subExp}(\mathsf{E}_1) \cup \mathsf{subExp}(\mathsf{E}_2)$
$\mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_\pi)$	$\{\mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_\pi)\} \cup \bigcup_{i=0}^{\pi} \mathsf{subExp}(\mathsf{E}_i)$

Every expression has some important properties that can be determined statically. We formally define the notions of *depth*, *variables occurring in an expression* and *fields an expression might read*.

Definition 6.3 (Expressions depth). *For every type environment $\tau \in \mathcal{T}$, we define a function* $\mathsf{depth} : \mathbb{E}_\tau \to \mathbb{N}$ *mapping expressions to their depths:*

$$\begin{aligned}
\mathsf{depth}(n) &= 0, \forall n \in \mathbb{V} \\
\mathsf{depth}(v) &= 0, \forall v \in \mathsf{dom}(\tau) \\
\mathsf{depth}(\mathsf{E}_1 \oplus \mathsf{E}_2) &= 1 + max_{i\in\{1,2\}}\{\mathsf{depth}(\mathsf{E}_i)\} \\
\mathsf{depth}(\mathsf{E}.f) &= 1 + \mathsf{depth}(\mathsf{E}) \\
\mathsf{depth}(\mathsf{E}.\texttt{length}) &= 1 + \mathsf{depth}(\mathsf{E}) \\
\mathsf{depth}(\mathsf{E}_1[\mathsf{E}_2]) &= 1 + max_{i\in\{1,2\}}\{\mathsf{depth}(\mathsf{E}_i)\} \\
\mathsf{depth}(\mathsf{E}_0.m(\mathsf{E}_1,\ldots,\mathsf{E}_\pi)) &= 1 + max_{0\le i\le\pi}\{\mathsf{depth}(\mathsf{E}_i)\}.
\end{aligned}$$

Definition 6.4 (Variables). *For every type environment $\tau \in \mathcal{T}$, we define a map* variables : $\mathbb{E}_\tau \to \wp(\mathsf{dom}(\tau))$ *yielding the variables occurring in an expression as:*

$$\begin{aligned}
\mathsf{variables}(n) &= \varnothing \\
\mathsf{variables}(v) &= \{v\} \\
\mathsf{variables}(\mathsf{E}_1 \oplus \mathsf{E}_2) &= \mathsf{variables}(\mathsf{E}_1) \cup \mathsf{variables}(\mathsf{E}_2) \\
\mathsf{variables}(\mathsf{E}.f) &= \mathsf{variables}(\mathsf{E}) \\
\mathsf{variables}(\mathsf{E}.\texttt{length}) &= \mathsf{variables}(\mathsf{E}) \\
\mathsf{variables}(\mathsf{E}_1[\mathsf{E}_2]) &= \mathsf{variables}(\mathsf{E}_1) \cup \mathsf{variables}(\mathsf{E}_2) \\
\mathsf{variables}(\mathsf{E}_0.m(\mathsf{E}_1,\ldots,\mathsf{E}_\pi)) &= \bigcup_{i=0}^{\pi} \mathsf{variables}(\mathsf{E}_i),
\end{aligned}$$

where $n \in \mathbb{Z}$, $v \in \mathsf{dom}(\tau)$, $f \in \mathcal{F}_\tau$ and $m \in \mathcal{M}_\tau$.

Definition 6.5 (Fields). *For every type environment $\tau \in \mathcal{T}$, we define a map* fields : $\mathbb{E}_\tau \to \wp(\mathcal{F}_\tau)$ *yielding the fields that might be read during the evaluations of an expression as:*

$$\begin{aligned}
\mathsf{fields}(n) &= \varnothing \\
\mathsf{fields}(v) &= \varnothing \\
\mathsf{fields}(\mathsf{E}_1 \oplus \mathsf{E}_2) &= \mathsf{fields}(\mathsf{E}_1) \cup \mathsf{fields}(\mathsf{E}_2) \\
\mathsf{fields}(\mathsf{E}.f) &= \mathsf{fields}(\mathsf{E}) \cup \{f\} \\
\mathsf{fields}(\mathsf{E}.\texttt{length}) &= \mathsf{fields}(\mathsf{E}) \\
\mathsf{fields}(\mathsf{E}_1[\mathsf{E}_2]) &= \mathsf{fields}(\mathsf{E}_1) \cup \mathsf{fields}(\mathsf{E}_2) \\
\mathsf{fields}(\mathsf{E}_0.m(\mathsf{E}_1,\ldots,\mathsf{E}_\pi)) &= \bigcup_{i=0}^{\pi} \mathsf{fields}(\mathsf{E}_i) \cup \{f \mid m \textit{ might read } f\},
\end{aligned}$$

where $n \in \mathbb{Z}$, $v \in \mathsf{dom}(\tau)$, $f \in \mathcal{F}_\tau$ and $m \in \mathcal{M}_\tau$.

Note that the definition of flds requires a preliminary computation of the fields possibly read by a method m, which might just be a transitive closure of the fields f for which a getfield occurs in m or in at least one method invoked by m. There exist some more precise approximations of this useful piece of information, e.g., the one determined by our Julia tool. Anyway, in the absence of this approximation, we can always assume the least precise sound hypothesis: every method can read every field.

Example 6.6. Consider the class `Event` introduced in Fig. 4.1 and let us assume that the static type of the local variable l_2 is `Event`. Then expression $\mathsf{E} = l_2$.`delayMinBy`(15) satisfies the following equalities:

- $\mathsf{depth}(l_2.\texttt{delayMinBy}(15)) = 1 + \max\{\mathsf{depth}(l_2), \mathsf{depth}(15)\} = 1 + \max\{0, 0\} = 1$;
- $\mathsf{variables}(l_2.\texttt{delayMinBy}(15)) = \mathsf{variables}(l_2) \cup \mathsf{variables}(15) = \{l_2\}$;
- $\mathsf{fields}(l_2.\texttt{delayMinBy}(15)) = \mathsf{fields}(l_2) \cup \mathsf{fields}(15) \cup \{f \mid \texttt{delayMinBy}$ might read $f\} = \{\texttt{min}\}$.

The latter follows from the fact that `delayMinBy` contains only one getfield concerning the field `min` and no call instruction (Fig. 4.1). □

In the following we show how the expressions introduced in Definition 6.1 are evaluated in an arbitrary state $\langle \rho, \mu \rangle$. It is worth noting that some of these expressions represent the result of a method invocation. Their evaluation, in general, might modify the initial memory μ, so we must be aware of the side-effects of the methods appearing in these expressions. We define the *non-standard evaluation* of an expression e in a state $\langle \rho, \mu \rangle$ as a pair $\langle w, \mu' \rangle$, where w is the computed value of e, while μ' is the updated memory obtained from μ after the evaluation of e.

Definition 6.7 (Non-standard evaluation of expressions). *A* non-standard evaluation of expressions *in a state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ is a partial map $[\![\cdot]\!]^* : \mathbb{E}_\tau \to \Sigma_\tau \to \mathbb{V} \times \mathbb{M}$ defined as:*

1. *for every $n \in \mathbb{Z}$, $[\![n]\!]^*\sigma = \langle n, \mu \rangle$;*
2. *for every $v \in \mathsf{dom}(\tau)$, $[\![v]\!]^*\sigma = \langle \rho(v), \mu \rangle$;*
3. *$[\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\sigma$ is defined only if*
   - *$[\![\mathsf{E}_1]\!]^*\sigma = \langle n_1, \mu_1 \rangle$,*
   - *$[\![\mathsf{E}_2]\!]^*\langle \rho, \mu_1 \rangle = \langle n_2, \mu_2 \rangle$ and*
   - *$n_1, n_2 \in \mathbb{Z}$.*

   In that case $[\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\sigma = \langle n_1 \oplus n_2, \mu_2 \rangle$, otherwise it is undefined;
4. *$[\![\mathsf{E}.f]\!]^*\sigma$ is defined only if*
   - *$[\![\mathsf{E}]\!]^*\sigma = \langle \ell, \mu_1 \rangle$,*
   - *$\ell \in \mathbb{L}$,*
   - *$\mu_1(\ell).\texttt{type} \in \mathbb{K}$ and*
   - *$f \in \mathsf{F}(\mu_1(\ell).\texttt{type})$.*

   In that case $[\![\mathsf{E}.f]\!]^*\sigma = \langle (\mu_1(\ell).\phi)(f), \mu_1 \rangle$, otherwise it is undefined;
5. *$[\![\mathsf{E}.\texttt{length}]\!]^*\sigma$ is defined only if*
   - *$[\![\mathsf{E}]\!]^*\sigma = \langle \ell, \mu_1 \rangle$,*
   - *$\ell \in \mathbb{L}$ and*
   - *$\mu_1(\ell).\texttt{type} \in \mathbb{A}$.*

   In that case $[\![\mathsf{E}.\texttt{length}]\!]^*\sigma = \langle \mu_1(\ell).\texttt{length}, \mu_1 \rangle$, otherwise it is undefined;
6. *$[\![\mathsf{E}_1[\mathsf{E}_2]]\!]^*\sigma$ is defined only if*
   - *$[\![\mathsf{E}_1]\!]^*\sigma = \langle \ell, \mu_1 \rangle$,*
   - *$[\![\mathsf{E}_2]\!]^*\langle \rho, \mu_1 \rangle = \langle n, \mu_2 \rangle$,*
   - *$\ell \in \mathbb{L}$,*
   - *$\mu_2(\ell).\texttt{type} \in \mathbb{A}$,*
   - *$n \in \mathbb{Z}$ and*
   - *$0 \leq n < \mu_2(\ell).\texttt{length}$.*

   In that case $[\![\mathsf{E}_1[\mathsf{E}_2]]\!]^*\sigma = \langle (\mu_2(\ell).\phi)(n), \mu_2 \rangle$, otherwise it is undefined;
7. *in order to compute $[\![\mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_\pi)]\!]^*\sigma$, we determine $[\![\mathsf{E}_0]\!]^*\langle \rho, \mu \rangle = \langle w_0, \mu_0 \rangle$, and for each $1 \leq i < \pi$, we evaluate E_{i+1} in the state $\langle \rho, \mu_i \rangle$: $[\![\mathsf{E}_{i+1}]\!]^*\langle \rho, \mu_i \rangle = \langle w_{i+1}, \mu_{i+1} \rangle$.*

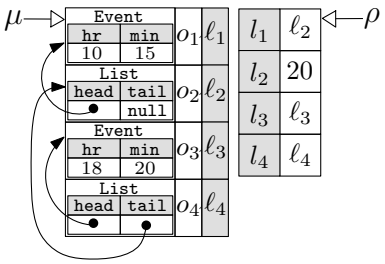


Fig. 6.2. A JVM state $\sigma = \langle \rho, \mu \rangle$

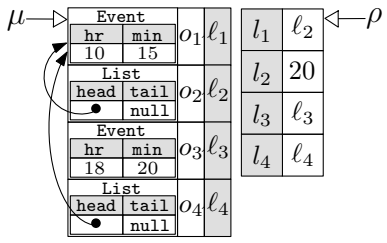


Fig. 6.3. A JVM state $\sigma_1 = \langle \rho, \mu_1 \rangle$

If $w_0 \in \mathbb{L}$ and $\mu_\pi(w_0).\texttt{type} \in \mathbb{K}$, we run the method m of the object $\mu_\pi(w_0)$ with parameters $w_1, \ldots, w_\pi$ and if it terminates with no exception, the result of the evaluation is the pair composed of m's return value w and the memory μ' obtained from μ_π as a side-effect of m.

We write $[\![\mathsf{E}]\!]\sigma$ do denote the actual value of E in σ, without the updated memory.

The following example illustrates the non-standard evaluation of some simple expressions.

Example 6.8. Consider the state $\sigma = \langle \rho, \mu \rangle$ given in Fig. 6.2, where `Event` and `List` are classes introduced in Figures 4.1 and 3.2. Let us evaluate the following expressions in σ: l_3.`min`, l_4.`head.min`, l_3.`delayMinBy`(15) and l_4.`removeFirst`().`setDelay`(15), where `removeFirst` is a method of the class `List` which returns the head of a list of events and removes it from that list.

$[\![l_3.\texttt{min}]\!]^*\sigma$: By Definition 6.7 (case 2), we have $[\![l_3]\!]^*\langle \rho, \mu \rangle = \langle \rho(l_3), \mu \rangle = \langle \ell_3, \mu \rangle$, where $\ell_3 \in \mathbb{L}$. Moreover, $\mu(\ell_3).\texttt{type} = \texttt{Event} \in \mathbb{K}$ and $\texttt{min} \in \mathsf{F}(\texttt{Event})$. Hence, the conditions imposed by the case 4 are satisfied and

$$[\![l_3.\texttt{min}]\!]^*\langle \rho, \mu \rangle = \langle (\mu(\ell_3).\phi)(\texttt{min}), \mu \rangle = \langle 20, \mu \rangle,$$

while $[\![l_3.\texttt{min}]\!]\langle \rho, \mu \rangle = 20$.

$[\![l_4.\texttt{head.min}]\!]^*\sigma$: By Definition 6.7 (case 2), we have $[\![l_4]\!]^*\langle \rho, \mu \rangle = \langle \rho(l_4), \mu \rangle = \langle \ell_4, \mu \rangle$, where $\ell_4 \in \mathbb{L}$. Moreover, $\mu(\ell_4).\texttt{type} = \texttt{List} \in \mathbb{K}$ and $\texttt{head} \in \mathsf{F}(\texttt{List})$. Hence, the conditions imposed by the case 4 are satisfied and

$$[\![l_4.\texttt{head}]\!]^*\langle \rho, \mu \rangle = \langle (\mu(\ell_4).\phi)(\texttt{head}), \mu \rangle = \langle \ell_3, \mu \rangle,$$

while $[\![l_4.\texttt{head}]\!]\langle \rho, \mu \rangle = \ell_3$. Similarly, $\mu(\ell_3).\texttt{type} = \texttt{Event} \in \mathbb{K}$ and $\texttt{min} \in \mathsf{F}(\texttt{Event})$, hence:

$$[\![l_4.\texttt{head.min}]\!]^*\langle \rho, \mu \rangle = \langle (\mu(\ell_3).\phi)(\texttt{min}), \mu \rangle = \langle 20, \mu \rangle.$$

while $[\![l_4.\texttt{head.min}]\!]\langle \rho, \mu \rangle = 20$.

$[\![l_3.\texttt{delayMinBy}(15)]\!]^*\sigma$: We have already shown that $[\![l_3]\!]^*\langle \rho, \mu \rangle = \langle \ell_3, \mu \rangle$, where $\ell_3 \in \mathbb{L}$ and $\mu(\ell_3).\texttt{type} \in \mathbb{K}$. By Definition 6.7 (case 1), we have $[\![15]\!]^*\langle \rho, \mu \rangle = \langle 15, \mu \rangle$. Since the resulting memory did not change, we run the method `delayMinBy` of the object $\mu(\ell_3) = o_3$ with parameter 15 (case 5 of Definiton 6.7). This method has no side effects (Fig. 4.1) and returns $((o_3.\phi)(\texttt{min}) + 15)\%60 = 20 + 15 = 35$. Therefore,

$$[\![l_3.\texttt{delayMinBy}(15)]\!]^*\langle\rho,\mu\rangle = \langle 35,\mu\rangle,$$

while $[\![l_3.\texttt{delayMinBy}(15)]\!]\langle\rho,\mu\rangle = 35$.

$[\![l_4.\texttt{removeFirst}().\texttt{setDelay}(15)]\!]^*\sigma$: We first determine $[\![l_4.\texttt{removeFirst}()]\!]^*\sigma$. Similarly to the previous two cases, we have $[\![l_4]\!]^*\langle\rho,\mu\rangle = \langle\rho(l_4),\mu\rangle = \langle\ell_4,\mu\rangle$, where $\ell_4 \in \mathbb{L}$ and $\mu(\ell_4).\texttt{type} = \texttt{List} \in \mathbb{K}$. Then we run the method `removeFirst` of the object $\mu(\ell_4) = o_4$, which returns the value of its field `head`, i.e., $(o_4.\phi)(\texttt{head}) = \ell_3 \in \mathbb{L}$, which is then removed from the list. Therefore, $[\![l_4.\texttt{removeFirst}()]\!]^*\langle\rho,\mu\rangle = \langle\ell_3,\mu_1\rangle$, where μ_1 is the updated memory depicted in Fig. 6.3. Since $[\![l_3]\!]^*\langle\rho,\mu_1\rangle = \langle\rho(l_3),\mu_1\rangle = \langle\ell_3,\mu_1\rangle$, with $\ell_3 \in \mathbb{L}$, $\mu_1(\ell_3).\texttt{type} = \texttt{Event} \in \mathbb{K}$ and $[\![15]\!]^*\langle\rho,\mu_1\rangle = \langle 15,\mu\rangle$, we can run the method `setDelay` of the object $\mu_1(\ell_3) = o_3$, which updates the value of the field `min` of the latter and returns that updated value, i.e., $((o_3.\phi)(\texttt{min}) + 15)\%60 = 20 + 15 = 35$. Thus, we obtain

$$[\![l_4.\texttt{removeFirst}().\texttt{setDelay}(15)]\!]^*\langle\rho,\mu\rangle = \langle 35,\mu_1[(\mu(\ell_3).\phi)(\texttt{min}) \mapsto 35]\rangle,$$

while $[\![l_4.\texttt{removeFirst}().\texttt{setDelay}(15)]\!]\langle\rho,\mu\rangle = 35$.

□

Finally, we define the notion of *alias expression*.

Definition 6.9 (Alias Expression). *We say that an expression* $\mathsf{E} \in \mathbb{E}_\tau$ *is an* alias expression *of a variable* $v \in \mathsf{dom}(\tau)$ *in a state* $\sigma = \langle\rho,\mu\rangle \in \Sigma_\tau$ *if and only if* $[\![\mathsf{E}]\!]\sigma = \rho(v)$.

Example 6.10. Consider again the state $\langle\rho,\mu\rangle$ given in Fig. 6.2. The value of the local variable 2 in that state is $\rho(l_2) = 20$. Moreover, in Example 6.8 we showed that $[\![l_3.\texttt{min}]\!]\langle\rho,\mu\rangle = 20$ and $[\![l_4.\texttt{head.min}]\!]\langle\rho,\mu\rangle = 20$. Thus,

$$[\![l_3.\texttt{min}]\!]\langle\rho,\mu\rangle = [\![l_4.\texttt{head.min}]\!]\langle\rho,\mu\rangle = \rho(l_2) = 20$$

and, by Definition 6.9, we can state that both $l_3.\texttt{min}$ and $l_4.\texttt{head.min}$ are alias expressions of l_2 in $\langle\rho,\mu\rangle$. Similarly,

$$[\![l_4.\texttt{head}]\!]\langle\rho,\mu\rangle = \rho(l_3) = \ell_3,$$

and we can state that $l_4.\texttt{head}$ is an alias expression of l_3 in $\langle\rho,\mu\rangle$. □

We specify when an execution of a bytecode instruction might affect the value of an expression. The following definition requires an additional information about the fields that might be updated and about the static types of the arrays that might be updated by an execution of a method. These pieces of information can be computed statically, for example, by the side-effects analysis of Julia. It is worth noting that when this information is not available, our analysis is still sound, although less precise: we may assume that every field and every array of any array type might be updated.

Definition 6.11 (canBeAffected)**.** *Let* τ *and* τ' *be the static type information at and immediately after a bytecode instruction* ins. *Suppose that* $\mathsf{dom}(\tau)$ *contains* i *local variables and* j *stack elements. In Fig. 6.4, we define a map* $\mathsf{canBeAffected}(\cdot,\mathsf{ins}) : \mathbb{E}_\tau \to \{\mathit{true},\mathit{false}\}$ *which, for every expression* $\mathsf{E} \in \mathbb{E}_\tau$*, determines whether* E *might be affected by an execution of* ins.

ins	canBeAffected(E, ins) = *true* if and only if
const x	never
load k t	
store k t	variables(E) $\cap \{l_k, s_{j-1}\} \neq \emptyset$
add	variables(E) $\cap \{s_{j-1}, s_{j-2}\} \neq \emptyset$
sub	
mul	
div	
rem	
inc k x	$l_k \in$ variables(E)
new κ	never
getfield $\kappa.f$:t	$s_{j-1} \in$ variables(E)
putfield $\kappa.f$:t	variables(E) $\cap \{s_{j-1}, s_{j-2}\} \neq \emptyset \ \vee \ \kappa.f$:t $\in$ fields(E)
arraynew t[ ]	$s_{j-1} \in$ variables(E)
arraylength t[ ]	$s_{j-1} \in$ variables(E)
arrayload t[ ]	variables(E) $\cap \{s_{j-1}, s_{j-2}\} \neq \emptyset$
arraystore t[ ]	variables(E) $\cap \{s_{j-1}, s_{j-2}, s_{j-3}\} \neq \emptyset \vee$ [there exists an evaluation of E which might read an element of an array of type t′[ ], where t′ $\in$ compatible(t)]
dup t	never
ifeq t ifne t	$s_{j-1} \in$ variables(E)
return t return void	variables(E) $\cap S \neq \emptyset$
throw κ	
catch	never
exception_is K	
call $m_1, \ldots, m_n$	there exists an execution of a dynamic target m_w, where $1 \leq w \leq n$, 1. [which might modify a field from fields(E)] or 2. [which might write into an element of an array of type t[ ] and there exists an evaluation of E which might read an element of an array of type t′[ ], where t′ $\in$ compatible(t)]

Fig. 6.4. Definition of a map canBeAffected($\cdot$, ins): $\mathbb{E}_\tau \to \{true, false\}$

That is, instructions that remove some variables from the stack (store, add, sub, mul, div, rem, putfield, arrayload, arraystore, ifeq, ifne, return and throw) affect the evaluation of all the expressions in which these variables appear. For instance, the execution of ifne t modifies the value of all the expressions containing the topmost stack element s_{j-1}. Instructions that write into a variable (store, add, sub, mul, div, rem, inc, getfield, arraylength and arrayload) might affect the evaluation of the expressions containing that variable. For instance, the execution of store k t might modify the value of all the expressions containing the local variable l_k, since this instruction writes a new value into that variable. Instruction putfield f might modify the evaluation of all the expressions that might read f. Instruction arraystore t[ ] might modify the evaluation of all the expressions that might read an array element whose type is compatible with t. Finally, call $m_1 \ldots m_k$ might modify the evaluation of all the expressions that might read a field f possibly mod-

ified by an m_w and of all the expressions which might read an element of an array of type t′[ ] if there exists a dynamic target m_w which writes into an array of type t[ ], where t′ and t are compatible types. For example, putfield `min` and call `setDelay` might modify the value of the expressions we evaluated in Example 6.8: l_3.`min`, l_3.`delayMinBy`(15) and l_4.`removeFirst`().`setDelay`(15).

On the other hand, the evaluation of an expression in a state might update the memory component of that state by modifying the value of some fields. In the following we specify whether any evaluation of an expression might modify some fields of interest.

Definition 6.12 (mightMdf)**.** *Function* mightMdf *specifies whether a field belonging to a set of fields $F \subseteq \mathcal{F}_\tau$ might be modified during the evaluation of an expression* E*:*

$$
\begin{aligned}
\mathsf{mightModify}(n, F) &= \mathit{false}\\
\mathsf{mightModify}(v, F) &= \mathit{false}\\
\mathsf{mightModify}(\mathsf{E}_1 \oplus \mathsf{E}_2, F) &= \mathsf{mightModify}(\mathsf{E}_1, F) \vee \mathsf{mightModify}(\mathsf{E}_2, F)\\
\mathsf{mightModify}(\mathsf{E}.f, F) &= \mathsf{mightModify}(\mathsf{E}, F)\\
\mathsf{mightModify}(\mathsf{E}.\mathit{length}, F) &= \mathsf{mightModify}(\mathsf{E}, F)\\
\mathsf{mightModify}(\mathsf{E}_1[\mathsf{E}_2], F) &= \mathsf{mightModify}(\mathsf{E}_1, F) \vee \mathsf{mightModify}(\mathsf{E}_2, F)\\
\mathsf{mightModify}(\mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_\pi), F) &= \bigvee_{i=0}^{\pi} \mathsf{mightModify}(\mathsf{E}_i, F) \ \vee \ [\text{an execution of } m\\
&\qquad \text{might modify a field from } F],
\end{aligned}
$$

where $n \in \mathbb{Z}$, $v \in \mathsf{dom}(\tau)$, $f \in \mathcal{F}_\tau$ and $m \in \mathcal{M}_\tau$.

Namely, evaluations of constants and variables do not modify any field. Evaluations of $\mathsf{E}_1 \oplus \mathsf{E}_2$ and $\mathsf{E}_1[\mathsf{E}_2]$ modify a field from F if there exists an evaluation of E_1 or E_2 modifying a field from F. Similarly, evaluations of $\mathsf{E}.f$ and $\mathsf{E}.\mathtt{length}$ modify a field from F if there exists an evaluation of E modifying a field from F. Evaluations of $\mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_\pi)$ might modify a field from F if there is an evaluation of any of E_is modifying a field from F or if the execution of m might modify a field from F.

Example 6.13. Consider one more time class `Event` shown in Fig. 4.1. Since the method `setDelay`() writes into the field `min` of class `Event`, we have

$$\mathsf{mightModify}(l_4.\mathtt{removeFirst}().\mathtt{setDelay}(15), \{\mathtt{Event.min}\colon \mathtt{int}\}) = \mathit{true}.$$

□

In the following we prove some technical lemmas. The first one states that if two environments ρ and ρ' coincide on all the variables appearing in an arbitrary expression E but on a given variable a, and if, for a given constant n, $\rho'(a) \oplus n = \rho(a)$ holds then, for an arbitrary memory μ, the evaluation of E in $\langle \rho, \mu \rangle$ coincides with the evaluation of E in which all occurrences of a are replaced by $a \oplus n$ in $\langle \rho', \mu \rangle$.

Lemma 6.14. *Consider a type environment $\tau \in \mathcal{T}$, a variable $a \in \mathsf{dom}(\tau)$, a constant $n \in \mathbb{Z}$ and two environments ρ and ρ' such that $\rho'(a) \oplus n = \rho(a)$. Let $\mathsf{E} \in \mathbb{E}_\tau$ be an*

arbitrary expression. If for every variable $v \in \mathsf{variables}(\mathsf{E}) \setminus \{a\}$, $\rho'(v) = \rho(v)$ *holds, then for every memory* μ,

$$[\![\mathsf{E}[(a \oplus n)/a]]\!]^*\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]^*\langle\rho, \mu\rangle,$$

where $\mathsf{E}[(a \oplus n)/a]$ *denotes the expression* E *with all the occurrences of* a *replaced with* $a \oplus n$.

Proof. We proof this lemma by induction on the depth of E.
Base case: Let μ be an arbitrary memory. If $\mathsf{depth}(\mathsf{E}) = 0$, then $\mathsf{E} = m \in \mathbb{V}$ or $\mathsf{E} = v \in \mathsf{dom}(\tau)$. In the former case, by Definition 6.7,

$$[\![m]\!]^*\langle\rho, \mu\rangle = \langle m, \mu\rangle = [\![m]\!]^*\langle\rho', \mu\rangle = [\![m[(a \oplus n)/a]]\!]^*\langle\rho', \mu\rangle.$$

In the latter case, if $v \neq a$ then, by Definition 6.7,

$$[\![v]\!]^*\langle\rho, \mu\rangle = \langle\rho(v), \mu\rangle = \langle\rho'(v), \mu\rangle = [\![v]\!]^*\langle\rho', \mu\rangle = [\![v[(a \oplus n)/a]]\!]^*\langle\rho', \mu\rangle.$$

If $v = a$, by hypothesis, $\rho(a) = \rho'(a) \oplus n$ and by Definition 6.7, we obtain

$$[\![a]\!]^*\langle\rho, \mu\rangle = \langle\rho(a), \mu\rangle = \langle\rho'(a) \oplus n, \mu\rangle = [\![(a \oplus n)]\!]^*\langle\rho', \mu\rangle = [\![a[(a \oplus n)/a]]\!]^*\langle\rho', \mu\rangle.$$

Induction: Suppose that for every expression E' of depth at most k hypothesis holds, i.e., if for every variable $v \in \mathsf{variables}(\mathsf{E}') \setminus \{a\}$, $\rho'(v) = \rho(v)$ holds, then for every memory μ, $[\![\mathsf{E}[(a \oplus n)/a]]\!]\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]\langle\rho, \mu\rangle$ holds. Let E be an expression such that $\mathsf{depth}(\mathsf{E}) = k + 1$. If there exists a variable $v \in \mathsf{variables}(\mathsf{E}) \setminus \{a\}$ such that $\rho(v) \neq \rho(v')$, then the result trivially holds. Otherwise, $\forall v \in \mathsf{variables}(\mathsf{E}) \setminus \{a\}.\rho'(v) = \rho(v)$ holds and we distinguish different possible forms of E:

- If $\mathsf{E} = \mathsf{E}_1 \oplus \mathsf{E}_2$, we have $k + 1 = \mathsf{depth}(\mathsf{E}) = 1 + \max\{\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2)\}$ (Definition 6.3), which entails $\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2) \leq k$. Since $\mathsf{variables}(\mathsf{E}) = \mathsf{variables}(\mathsf{E}_1) \cup \mathsf{variables}(\mathsf{E}_2)$, and for every $v \in \mathsf{variables}(\mathsf{E}) \setminus \{a\}$, $\rho'(v) = \rho(v)$, we conclude that ρ and ρ' also agree on the values of all the variables different from a which appear in E_1 and E_2. Let μ be an arbitrary memory, then by inductive hypothesis on E_1 and E_2 we have:

  $$\begin{aligned} [\![\mathsf{E}_1]\!]^*\langle\rho, \mu\rangle &= [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle\rho', \mu\rangle = \langle w_1, \mu_1\rangle \\ [\![\mathsf{E}_2]\!]^*\langle\rho, \mu_1\rangle &= [\![\mathsf{E}_2[(a \oplus n)/a]]\!]^*\langle\rho', \mu_1\rangle = \langle w_2, \mu_2\rangle, \end{aligned}$$

  where $w_1, w_2 \in \mathbb{Z}$. Therefore:

  $$\begin{aligned} [\![\mathsf{E}]\!]^*\langle\rho, \mu\rangle &= [\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\langle\rho, \mu\rangle \\ &= \langle w_1 \oplus w_2, \mu_2\rangle && \text{[By Definition 6.7]} \\ &= [\![\mathsf{E}_1[(a \oplus n)/a] \oplus \mathsf{E}_2[(a \oplus n)/a]]\!]^*\langle\rho', \mu\rangle && \text{[By hypothesis and Def. 6.7]} \\ &= [\![\mathsf{E}[(a \oplus n)/a]]\!]^*\langle\rho', \mu\rangle. \end{aligned}$$

- If $\mathsf{E} = \mathsf{E}_1.f$, then $k + 1 = \mathsf{depth}(\mathsf{E}) = 1 + \mathsf{depth}(\mathsf{E}_1)$ (Definition 6.3), which entails $\mathsf{depth}(\mathsf{E}_1) = k$. Since $\mathsf{variables}(\mathsf{E}) = \mathsf{variables}(\mathsf{E}_1)$, we have that for every $v \in \mathsf{variables}(\mathsf{E}_1) \setminus \{a\}$, $\rho'(v) = \rho(v)$ and, by inductive hypothesis on E_1, $[\![\mathsf{E}_1]\!]^*\langle\rho, \mu\rangle = [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle\rho', \mu\rangle = \langle\ell, \mu_1\rangle$, where $\ell \in \mathbb{L}$ and $\mu_1(\ell).\mathtt{type} \in \mathbb{K}$. Therefore:

  $$\begin{aligned} [\![\mathsf{E}]\!]^*\langle\rho, \mu\rangle &= [\![\mathsf{E}_1.f]\!]^*\langle\rho, \mu\rangle \\ &= \langle(\mu_1(\ell).\phi)(f), \mu_1\rangle && \text{[By Definition 6.7]} \\ &= [\![\mathsf{E}_1[(a \oplus n)/a].f]\!]^*\langle\rho', \mu\rangle && \text{[By hypothesis and Definition 6.7]} \\ &= [\![\mathsf{E}[(a \oplus n)/a]]\!]^*\langle\rho', \mu\rangle. \end{aligned}$$

- If $\mathsf{E} = \mathsf{E}_1.\texttt{length}$, then $k+1 = \mathsf{depth}(\mathsf{E}) = 1 + \mathsf{depth}(\mathsf{E}_1)$ (Definition 6.3), which entails $\mathsf{depth}(\mathsf{E}_1) = k$. Since $\mathsf{variables}(\mathsf{E}) = \mathsf{variables}(\mathsf{E}_1)$, we have that for every $v \in \mathsf{variables}(\mathsf{E}_1) \setminus \{a\}$, $\rho'(v) = \rho(v)$ and, by inductive hypothesis on E_1, $[\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle = [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle\rho',\mu\rangle = \langle\ell,\mu_1\rangle$, where $\ell \in \mathbb{L}$ and $\mu_1(\ell).\texttt{type} \in \mathbb{A}$. Therefore:

$$\begin{aligned}
[\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_1.\texttt{length}]\!]^*\langle\rho,\mu\rangle \\
&= \langle\mu_1(\ell).\texttt{length},\mu_1\rangle && \text{[By Definition 6.7]} \\
&= [\![\mathsf{E}_1[(a \oplus n)/a].\texttt{length}]\!]^*\langle\rho',\mu\rangle && \text{[By hypothesis and Definition 6.7]} \\
&= [\![\mathsf{E}[(a \oplus n)/a]]\!]^*\langle\rho',\mu\rangle.
\end{aligned}$$

- If $\mathsf{E} = \mathsf{E}_1[\mathsf{E}_2]$, we have $k+1 = \mathsf{depth}(\mathsf{E}) = 1 + \max\{\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2)\}$ (Definition 6.3), which entails $\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2) \leq k$. Since $\mathsf{variables}(\mathsf{E}) = \mathsf{variables}(\mathsf{E}_1) \cup \mathsf{variables}(\mathsf{E}_2)$, and for every $v \in \mathsf{variables}(\mathsf{E}) \setminus \{a\}$, $\rho'(v) = \rho(v)$, we conclude that ρ and ρ' also agree on the values of all the variables different from a which appear in E_1 and E_2. Let μ be an arbitrary memory, then by inductive hypothesis on E_1 and E_2 we have:

$$\begin{aligned}
[\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle\rho',\mu\rangle = \langle\ell,\mu_1\rangle \\
[\![\mathsf{E}_2]\!]^*\langle\rho,\mu_1\rangle &= [\![\mathsf{E}_2[(a \oplus n)/a]]\!]^*\langle\rho',\mu_1\rangle = \langle k,\mu_2\rangle,
\end{aligned}$$

where $\ell \in \mathbb{L}$, $\mu_2(\ell).\texttt{type} \in \mathbb{A}$ and $k \in \mathbb{Z}$. Therefore:

$$\begin{aligned}
[\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_1[\mathsf{E}_2]]\!]^*\langle\rho,\mu\rangle \\
&= \langle(\mu_2(\ell).\phi)(k),\mu_2\rangle && \text{[By Definition 6.7]} \\
&= [\![\mathsf{E}_1[(a \oplus n)/a][\mathsf{E}_2[(a \oplus n)/a]]]\!]^*\langle\rho',\mu\rangle && \text{[By hypothesis and Def. 6.7]} \\
&= [\![\mathsf{E}[(a \oplus n)/a]]\!]^*\langle\rho',\mu\rangle.
\end{aligned}$$

- If $\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1,\ldots,\mathsf{E}_\pi)$, we have $k+1 = \mathsf{depth}(\mathsf{E}) = 1 + \max_{0\leq i\leq\pi}\{\mathsf{depth}(\mathsf{E}_i)\}$, and therefore $\mathsf{depth}(\mathsf{E}_i) \leq k$, for each $0 \leq i \leq \pi$. Since $\mathsf{variables}(\mathsf{E}) = \bigcup_{i=0}^{\pi} \mathsf{variables}(\mathsf{E}_i)$, and for every $v \in \mathsf{variables}(\mathsf{E}) \setminus \{a\}$, $\rho'(v) = \rho(v)$, we conclude that ρ and ρ' agree on the values of all the variables different from a which appear in each E_i. Let μ be an arbitrary memory, then by inductive hypothesis on E_1 and E_2 we have:

$$\begin{aligned}
[\![\mathsf{E}_0]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_0[(a \oplus n)/a]]\!]^*\langle\rho',\mu\rangle &&= \langle w_0,\mu_0\rangle \\
[\![\mathsf{E}_1]\!]^*\langle\rho,\mu_0\rangle &= [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle\rho',\mu_0\rangle &&= \langle w_1,\mu_1\rangle \\
&\cdots \\
[\![\mathsf{E}_\pi]\!]^*\langle\rho,\mu_{\pi-1}\rangle &= [\![\mathsf{E}_\pi[(a \oplus n)/a]]\!]^*\langle\rho',\mu_{\pi-1}\rangle &&= \langle w_\pi,\mu_\pi\rangle
\end{aligned}$$

Hence, for each $1 \leq i \leq \pi$, evaluations of both E_i and $\mathsf{E}_i[(a \oplus n)/a]$ in $\langle\rho,\mu_{i-1}\rangle$ and $\langle\rho',\mu_{i-1}\rangle$ respectively give equal results $\langle w_i,\mu_i\rangle$ and, by Definition 6.7, it implies that evaluations of both E in $\langle\rho,\mu\rangle$ and $\mathsf{E}[(a \oplus n)/a]$ in $\langle\rho',\mu\rangle$ are equal and correspond to the value returned by the method m. Namely, in both cases, the execution of m is deterministic since we fixed the actual parameters (receiver $\mu_\pi(w_0)$ and parameters $w_1,\ldots,w_\pi$) and the memory (μ_π), hence in both cases it will produce the same return value. This value is enriched with the resulting memory μ' obtained from μ_π as a side-effect of m's execution.

■

One particular case of Lemma 6.14 is when the constant n is 0. Namely, when ρ and ρ' coincide on all the variables appearing in E then the evaluations of the latter in $\langle\rho,\mu\rangle$ and $\langle\rho',\mu\rangle$ coincide too.

Corollary 6.15. *Consider a type environment $\tau \in \mathcal{T}$ and two environments ρ and ρ'. Let $\mathsf{E} \in \mathbb{E}_\tau$ be an arbitrary expression. If for every variable $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$ holds, then for every memory μ,*

$$[\![\mathsf{E}]\!]^*\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]^*\langle\rho, \mu\rangle.$$

Similarly, we prove another important result. The following lemma shows that if a state assigns the same values to two fixed variables a and b, then the evaluation of an arbitrary expression in that state does not change if we replace all the occurrences of a with b.

Lemma 6.16. *Consider a type environment $\tau \in \mathcal{T}$, variables $a, b \in \mathsf{dom}(\tau)$ and an environment ρ such that $\rho(a) = \rho(b)$. Let $\mathsf{E} \in \mathbb{E}_\tau$ be an arbitrary expression. Then, for every memory μ,*

$$[\![\mathsf{E}]\!]^*\langle\rho, \mu\rangle = [\![\mathsf{E}[b/a]]\!]^*\langle\rho, \mu\rangle,$$

where $\mathsf{E}[b/a]$ denotes the expression E with all the occurrences of a replaced with b.

Proof. We proof this lemma by induction on the depth of E.
Base case: Let μ be an arbitrary memory. If $\mathsf{depth}(\mathsf{E}) = 0$, then $\mathsf{E} = n \in \mathbb{V}$ or $\mathsf{E} = v \in \mathsf{dom}(\tau)$. In the former case, by Definition 6.7,

$$[\![n]\!]^*\langle\rho, \mu\rangle = \langle n, \mu\rangle = [\![n[b/a]]\!]^*\langle\rho, \mu\rangle.$$

In the latter case, if $v \neq a$ then, by Definition 6.7,

$$[\![v]\!]^*\langle\rho, \mu\rangle = \langle\rho(v), \mu\rangle = \langle\rho'(v), \mu\rangle = [\![v[b/a]]\!]^*\langle\rho, \mu\rangle.$$

If $v = a$, by hypothesis, $\rho(a) = \rho(b)$ and by Definition 6.7, we obtain

$$[\![a]\!]^*\langle\rho, \mu\rangle = \langle\rho(a), \mu\rangle = \langle\rho(b), \mu\rangle = [\![b]\!]^*\langle\rho, \mu\rangle = [\![a[b/a]]\!]^*\langle\rho, \mu\rangle.$$

Induction: Suppose that for every expression E' of depth at most k hypothesis holds, i.e., $[\![\mathsf{E}']\!]^*\langle\rho, \mu\rangle = [\![\mathsf{E}'[b/a]]\!]^*\langle\rho, \mu\rangle$, for every memory μ. Let E be an expression such that $\mathsf{depth}(\mathsf{E}) = k + 1$. We show that $[\![\mathsf{E}]\!]^*\sigma = [\![\mathsf{E}[b/a]]\!]^*\sigma$. We distinguish different possible forms of E:

- If $\mathsf{E} = \mathsf{E}_1 \oplus \mathsf{E}_2$, we have $k + 1 = \mathsf{depth}(\mathsf{E}) = 1 + \max\{\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2)\}$ (Definition 6.3), which entails $\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2) \leq k$. Therefore, inductive hypothesis holds on both E_1 and E_2. More precisely, inductive hypothesis entails

$$\begin{aligned} [\![\mathsf{E}_1]\!]^*\langle\rho, \mu\rangle &= [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle\rho', \mu\rangle = \langle w_1, \mu_1\rangle \\ [\![\mathsf{E}_2]\!]^*\langle\rho, \mu_1\rangle &= [\![\mathsf{E}_2[(a \oplus n)/a]]\!]^*\langle\rho', \mu_1\rangle = \langle w_2, \mu_2\rangle, \end{aligned}$$

  where $w_1, w_2 \in \mathbb{Z}$. Therefore:

$$\begin{aligned} [\![\mathsf{E}]\!]^*\langle\rho, \mu\rangle &= [\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\langle\rho, \mu\rangle \\ &= \langle w_1 \oplus w_2, \mu_2\rangle && \text{[By Definition 6.7]} \\ &= [\![\mathsf{E}_1[b/a] \oplus \mathsf{E}_2[b/a]]\!]^*\langle\rho, \mu\rangle && \text{[By hypothesis and Definition 6.7]} \\ &= [\![\mathsf{E}[b/a]]\!]^*\langle\rho, \mu\rangle. \end{aligned}$$

- If $\mathsf{E} = \mathsf{E}_1.f$, we have $k + 1 = \mathsf{depth}(\mathsf{E}) = 1 + \mathsf{depth}(\mathsf{E}_1)$, which entails $\mathsf{depth}(\mathsf{E}_1) = k$, and therefore, hypothesis holds on it, i.e.,

$$[\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle = [\![\mathsf{E}_1[b/a]]\!]^*\langle\rho,\mu\rangle = \langle\ell,\mu_1\rangle,$$

where $\ell \in \mathbb{L}$ and $\mu_1(\ell).\mathtt{type} \in \mathbb{K}$. We have

$$\begin{aligned}
[\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_1.f]\!]^*\langle\rho,\mu\rangle \\
&= \langle(\mu_1(\ell).\phi)(f),\mu_1\rangle && \text{[By Definition 6.7]} \\
&= [\![\mathsf{E}_1[b/a].f]\!]^*\langle\rho',\mu\rangle && \text{[By hypothesis and Definition 6.7]} \\
&= [\![\mathsf{E}[b/a]]\!]^*\langle\rho',\mu\rangle.
\end{aligned}$$

- If $\mathsf{E} = \mathsf{E}_1.\mathtt{length}$, we have $k + 1 = \mathsf{depth}(\mathsf{E}) = 1 + \mathsf{depth}(\mathsf{E}_1)$, which entails $\mathsf{depth}(\mathsf{E}_1) = k$, and therefore, hypothesis holds on it, i.e.,

$$[\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle = [\![\mathsf{E}_1[b/a]]\!]^*\langle\rho,\mu\rangle = \langle\ell,\mu_1\rangle,$$

where $\ell \in \mathbb{L}$ and $\mu_1(\ell).\mathtt{type} \in \mathbb{A}$. We have

$$\begin{aligned}
[\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_1.\mathtt{length}]\!]^*\langle\rho,\mu\rangle \\
&= \langle\mu_1(\ell).\mathtt{length},\mu_1\rangle && \text{[By Definition 6.7]} \\
&= [\![\mathsf{E}_1[b/a].\mathtt{length}]\!]^*\langle\rho',\mu\rangle && \text{[By hypothesis and Definition 6.7]} \\
&= [\![\mathsf{E}[b/a]]\!]^*\langle\rho',\mu\rangle.
\end{aligned}$$

- If $\mathsf{E} = \mathsf{E}_1[\mathsf{E}_2]$, we have $k + 1 = \mathsf{depth}(\mathsf{E}) = 1 + \max\{\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2)\}$ (Definition 6.3), which entails $\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2) \leq k$. Therefore, inductive hypothesis holds on both E_1 and E_2. More precisely, inductive hypothesis entails

$$\begin{aligned}
[\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle\rho',\mu\rangle = \langle\ell,\mu_1\rangle \\
[\![\mathsf{E}_2]\!]^*\langle\rho,\mu_1\rangle &= [\![\mathsf{E}_2[(a \oplus n)/a]]\!]^*\langle\rho',\mu_1\rangle = \langle k,\mu_2\rangle,
\end{aligned}$$

where $\ell \in \mathbb{L}$, $\mu_2(\ell).\mathtt{type} \in \mathbb{A}$ and $k \in \mathbb{Z}$. Therefore:

$$\begin{aligned}
[\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_1[\mathsf{E}_2]]\!]^*\langle\rho,\mu\rangle \\
&= \langle(\mu_2(\ell).\phi)(k),\mu_2\rangle && \text{[By Definition 6.7]} \\
&= [\![\mathsf{E}_1[b/a][\mathsf{E}_2[b/a]]]\!]^*\langle\rho,\mu\rangle && \text{[By hypothesis and Definition 6.7]} \\
&= [\![\mathsf{E}[b/a]]\!]^*\langle\rho,\mu\rangle.
\end{aligned}$$

- If $\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1,\ldots,\mathsf{E}_\pi)$, we have $k + 1 = \mathsf{depth}(\mathsf{E}) = 1 + \max_{0\leq i\leq\pi}\{\mathsf{depth}(\mathsf{E}_i)\}$, and therefore $\mathsf{depth}(\mathsf{E}_i) \leq k$, for each $0 \leq i \leq \pi$. Thus, hypothesis holds on each E_i, which entails:

$$\begin{aligned}
[\![\mathsf{E}_0]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_0[b/a]]\!]^*\langle\rho,\mu\rangle = \langle w_0,\mu_0\rangle \\
[\![\mathsf{E}_1]\!]^*\langle\rho,\mu_0\rangle &= [\![\mathsf{E}_1[b/a]]\!]^*\langle\rho,\mu_0\rangle = \langle w_1,\mu_1\rangle \\
&\cdots \\
[\![\mathsf{E}_\pi]\!]^*\langle\rho,\mu_{\pi-1}\rangle &= [\![\mathsf{E}_\pi[b/a]]\!]^*\langle\rho,\mu_{\pi-1}\rangle = \langle w_\pi,\mu_\pi\rangle.
\end{aligned}$$

Hence, for each $1 \leq i \leq \pi$, evaluation of both E_i and $\mathsf{E}_i[b/a]$ in $\langle\rho,\mu_{i-1}\rangle$ and $\langle\rho',\mu_{i-1}\rangle$ respectively gives equal result $\langle w_i,\mu_i\rangle$ and, by Definition 6.7, it implies that evaluations of both E and $\mathsf{E}[b/a]$ in $\langle\rho,\mu\rangle$ are equal and correspond to the value returned by the method m. Namely, in both cases, the execution of m is deterministic since we fixed the actual parameters (receiver $\mu_\pi(w_0)$ and parameters $w_1,\ldots,w_\pi$) and the memory (μ_π), hence in both cases it will produce the same return value. This value is enriched with the resulting memory μ' obtained from μ_π as a side-effect of m's execution.

■

6.3 Definition of Definite Expression Aliasing Analysis

The goal of this section is to define a static analysis that computes, for each program point, and each variable available at that point, a subset of expressions whose value must be equal to the value of that variable, for any possible execution of the program, every time that program point is reached. We show how it is possible to instantiate the parameters of the general parameterized framework introduced in Chapter 4 in order to obtain the desired static analysis. There are two essential things a designer should do in order to define such a static analysis:

1. Mathematically encode the property of interest, define the abstract domain and show how it can be related to the concrete one (Section 4.3);
2. Define a propagation rule for every possible arc available in the ACG, i.e., define an abstract semantics of our target language which simulates the behavior of concrete bytecode instructions with respect to the abstract domain defined above.

Subsections 6.3.1 and 6.3.2 deal with the points 1. and 2. respectively.

6.3.1 Abstract Domain Alias

The first goal of this section is to mathematically encode the property of interest. In Section 6.2 we explained when an expression E is aliased to a variable v in a state σ. This notion strictly depends on the current state of the program. We want to determine that property statically, and the most natural way for representing the fact that a variable must be aliased to some expressions is to assign to each variable available at a program point, a set of expressions that always have the same value as that variable itself. We followed this idea and formally defined the abstract domain Alias.

Definition 6.17 (Concrete and Abstract Domain). *The concrete and abstract domains over $\tau \in \mathcal{T}$ are $\mathsf{C}_\tau = \langle \wp(\Sigma_\tau), \subseteq, \cup, \cap, \Sigma_\tau, \varnothing \rangle$ and $\textsc{Alias}_\tau = \langle \mathcal{A}_\tau, \sqsubseteq, \sqcup, \sqcap, \top_\tau, \bot_\tau \rangle$, where*

- $\mathcal{A}_\tau = (\wp(\mathbb{E}_\tau))^{|\tau|}$;
- *for every* $A^1 = \langle \mathsf{A}^1_0, \ldots, \mathsf{A}^1_{|\tau|-1} \rangle \in \mathcal{A}_\tau$ *and* $A^2 = \langle \mathsf{A}^2_0, \ldots, \mathsf{A}^2_{|\tau|-1} \rangle \in \mathcal{A}_\tau$,

$$A^1 \sqsubseteq A^2 \Leftrightarrow \forall 0 \leq i < |\tau|, \mathsf{A}^1_i \supseteq \mathsf{A}^2_i;$$

- *for every* $A^1 = \langle \mathsf{A}^1_0, \ldots, \mathsf{A}^1_{|\tau|-1} \rangle \in \mathcal{A}_\tau$ *and* $A^2 = \langle \mathsf{A}^2_0, \ldots, \mathsf{A}^2_{|\tau|-1} \rangle \in \mathcal{A}_\tau$,

$$A^1 \sqcup A^2 = \langle \mathsf{A}^1_0 \cap \mathsf{A}^2_0, \ldots, \mathsf{A}^1_{|\tau|-1} \cap \mathsf{A}^2_{|\tau|-1} \rangle;$$

- *for every* $A^1 = \langle \mathsf{A}^1_0, \ldots, \mathsf{A}^1_{|\tau|-1} \rangle \in \mathcal{A}_\tau$ *and* $A^2 = \langle \mathsf{A}^2_0, \ldots, \mathsf{A}^2_{|\tau|-1} \rangle \in \mathcal{A}_\tau$,

$$A^1 \sqcap A^2 = \langle \mathsf{A}^1_0 \cup \mathsf{A}^2_0, \ldots, \mathsf{A}^1_{|\tau|-1} \cup \mathsf{A}^2_{|\tau|-1} \rangle;$$

- $\top_\tau = \varnothing^{|\tau|}$;
- $\bot_\tau = (\mathbb{E}_\tau)^{|\tau|}$.

For a fixed number $d \in \mathbb{N}$, we write $\textsc{Alias}^d_\tau$ to denote a restriction of $\textsc{Alias}_\tau$ in which all expressions's depth is at most d, i.e.,

$$\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau|-1} \rangle \in \textsc{Alias}^d_\tau \Leftrightarrow \forall 0 \leq i < |\tau|. \forall \mathsf{E} \in \mathsf{A}_i. \mathsf{depth}(\mathsf{E}) \leq d.$$

$\textsc{Alias}^d_\tau$'s top and bottom elements are denoted by $\top^d_\tau$ and $\bot^d_\tau$ respectively.

Concrete states σ corresponding to an abstract element $\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau|-1} \rangle$ must satisfy the aliasing information represented by the latter, i.e., for each $0 \leq r < |\tau|$, the value of all the expressions from the set A_r in σ must coincide with the value of v_r in σ (*definite* aliasing). Our concretization map formalizes this intuition.

Definition 6.18 (Concretization map). *Consider a type environment $\tau \in \mathcal{T}$ and an abstract state $A = \langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau|-1} \rangle \in \textsc{Alias}_\tau$. We define $\gamma_\tau : \textsc{Alias}_\tau \to \mathsf{C}_\tau$, the concretization map of our abstract domain $\textsc{Alias}_\tau$ as follows:*

$$\gamma(A) = \{\langle \rho, \mu \rangle \in \Sigma_\tau \mid \forall 0 \leq r < |\tau| . \forall \mathsf{E} \in \mathsf{A}_r . [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r)\}.$$

Example 6.19. Consider a type environment $\tau \in \mathcal{T}$ with $\mathsf{dom}(\tau) = \{l_0, \ldots, l_4\}$ and the state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ given in Fig. 6.2. We define the following abstract states from $\textsc{Alias}_\tau$:

$$\begin{array}{lc}
A^1 = \langle & \overbrace{\varnothing}^{l_0}, \overbrace{\varnothing}^{l_1}, \overbrace{\{l_3.\texttt{min}\}}^{l_2}, \overbrace{\{l_4.\texttt{head}\}}^{l_3}, \overbrace{\varnothing}^{l_4} \rangle \in \textsc{Alias}^1_\tau \\
A^2 = \langle & \varnothing, \varnothing, \{l_3.\texttt{min}, l_4.\texttt{head}.\texttt{min}\}, \{l_4.\texttt{head}\}, \varnothing \rangle \in \textsc{Alias}^2_\tau \\
A^3 = \langle & \varnothing, \{l_3\}, \{l_3.\texttt{min}, l_4.\texttt{head}.\texttt{min}\}, \varnothing, \varnothing \rangle \in \textsc{Alias}^2_\tau.
\end{array}$$

By Definition 6.17, we can state that the aliasing information contained in A^2 is more precise comparing to A^1, i.e., $A^2 \sqsubseteq A^1$. On the other hand, we cannot compare A^1 and A^3, since the approximation of the aliasing information related to l_2 of A^1 is less precise than the one of A^3, but the approximation of the aliasing information related to l_3 of A^1 is more precise than the one of A^3.

Moreover, σ satisfies the aliasing information contained in both A^1 and A^2: in Example 6.10 we have shown that in σ variable l_2 is aliased to $l_3.\texttt{min}$ and $l_4.\texttt{head}.\texttt{min}$, while variable l_3 is aliased to $l_4.\texttt{head}$. Hence,

$$\sigma \in \gamma_\tau(A^2) \subseteq \gamma_\tau(A^1).$$

On the other hand, according to A^3, l_1 is aliased to l_3, which is not the case in σ: $(\rho(l_1) = \ell_2 \neq \ell_3 = \rho(l_3)$, which entails $\sigma \notin \gamma_\tau(A^3)$. □

Requirements 4.1 and 4.2 deal with the abstract domain representing the property of interest. We will show in Section 6.4 that the $\textsc{Alias}^d_\tau$ abstract domain actually satisfies these requirements.

6.3.2 Propagation Rules

In Chapter 4 we defined the notion of abstract constraint graph, ACG, and we showed how these graphs can be constructed from the text of the program under analysis. We recall that an ACG is composed of the set of nodes, corresponding to different program bytecode instructions, and of the set of arcs which connect those nodes. Each node of an ACG created for the definite expression aliasing analysis is enriched with an element of the abstract domain Alias. That abstract element represents an approximation of the actual aliasing information available at that point. On the other hand, each arc of that ACG is enriched with a propagation rule showing how the abstract elements (i.e., approximations) available at arc's sources are propagated to its sink. In Section 4.4 we specified the

requirements that these propagation rules have to satisfy in order to guarantee the soundness of the overall analysis, but we did not give any concrete definition of any propagation rule, since they strictly depend on the property which is being analyzed, while we were dealing with a generic property in that chapter. On the contrary, in this chapter, we are interested in one particular property, i.e., expressions definitely aliased to program variables, we have shown how it can be mathematically represented in our framework, and in this subsection we show how we propagate the abstract elements approximating that property.

In the following we assume the presence of a *side-effects* approximation. Namely, we suppose that, for each method or constructor m available in the program under analysis, or in any of the libraries that program may use, there exist the following pieces of information computed statically:

- a set of *fields that might be read* during any possible execution of m;
- a set of *fields that might be updated* during any possible execution of m;
- a set of *array types of all possible arrays whose elements might be read* during any possible execution of m and
- a set of *array types of all possible arrays whose elements might be updated* during any possible execution of m.

These pieces of information can be computed statically, and our tool Julia is able to provide them. Our analysis works correctly even when these approximations are not available: we can always assume that each method or constructor might read and modify every field and elements of arrays of every possible array type. In that case the definite expression aliasing information we determine would be less precise, but still sound.

According to Definition 4.1, we distinguish between simple $(1-1)$ arcs, having one source and one sink node, and multi $(2-1)$ arcs, which have two source and one sink node. We assume for all $1-1$ arcs that τ and τ' are the static type information at and immediately after the execution of a bytecode instruction ins, respectively. Moreover, we assume that τ contains j stack elements and i local variables. We write $\mathsf{noStackElements}(\mathsf{E})$ to denote that an expression E contains no stack elements, i.e., $\mathsf{variables}(\mathsf{E}) \cap \{s_0, \ldots, s_{j-1}\} = \emptyset$. In the following we define the propagation rules related to the definite expression aliasing analysis.

Definition 6.20 (Sequential arcs). *If* ins *is a bytecode instruction, distinct from* call*, immediately followed by a bytecode instruction* ins'*, distinct from* catch*, then an* $1-1$ sequential arc *is built from* $\boxed{\mathsf{ins}}$ *to* $\boxed{\mathsf{ins}'}$*, with a propagation rule*

$$\lambda\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau|-1}\rangle.\langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{|\tau'|-1}\rangle,$$

where, for each $0 \leq r < |\tau'|$*,* A'_r *is defined by one of the following rules:*

RULE #1*: If* $\mathsf{ins} = \mathsf{const}\ x$*, then*

$$\mathsf{A}'_r = \begin{cases} \mathsf{A}_r & \text{if } r \neq |\tau| \\ \{x\} & \text{if } r = |\tau|. \end{cases}$$

RULE #2*: If* $\mathsf{ins} = \mathsf{load}\ k\ \mathsf{t}$*, then*

$$\mathsf{A}'_r = \begin{cases} \mathsf{A}_r \cup \mathsf{A}_r[s_j/l_k] & \textit{if } r \notin \{k, |\tau|\} \\ \mathsf{A}_k \cup \{s_j\} & \textit{if } r = k \\ \mathsf{A}_k \cup \{l_k\} & \textit{if } r = |\tau|. \end{cases}$$

Rule #3*: If* ins = store k t*, then*

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \textit{if } r \neq k \\ \{\mathsf{E} \in \mathsf{A}_{|\tau|-1} \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \textit{if } r = k. \end{cases}$$

Rule #4*: If* ins $\in$ {add, sub, mul, div, rem}*, then*

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \textit{if } r \neq |\tau|-2 \\ \{\mathsf{E}_1 \oplus \mathsf{E}_2 \mid \mathsf{E}_1 \in \mathsf{A}_{|\tau|-2} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_1, \mathsf{ins}) \wedge \\ \qquad \mathsf{E}_2 \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_2, \mathsf{ins})\} & \textit{if } r = |\tau|-2, \end{cases}$$

where $\oplus$ *is* $+, -, \times$, div, $\%$ *when* ins *is* add, sub, mul, div, rem *respectively.*

Rule #5*: If* ins = inc k x*, then*

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E}[l_k - x/l_k] \mid \mathsf{E} \in \mathsf{A}_r\} & \textit{if } r \neq k \\ \varnothing & \textit{if } r = k. \end{cases}$$

Rule #6*: If* ins = new κ*, then*

$$\mathsf{A}'_r = \begin{cases} \mathsf{A}_r & \textit{if } r \neq |\tau| \\ \varnothing & \textit{if } r = |\tau|. \end{cases}$$

Rule #7*: If* ins = getfield f*, then*

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \textit{if } r \neq |\tau|-1 \\ \{\mathsf{E}.f \mid \mathsf{E} \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}) \wedge \neg\mathsf{mightModify}(\mathsf{E}, \{f\})\} & \textit{if } r = |\tau|-1. \end{cases}$$

Rule #8*: If* ins = putfield f*, then*

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}.$$

Rule #9*: If* ins = arraynew t[ ]*, then*

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \textit{if } r \neq |\tau|-1 \\ \varnothing & \textit{if } r = |\tau|-1. \end{cases}$$

Rule #10*: If* ins = arraylength t[ ]*, then*

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \textit{if } r \neq |\tau|-1 \\ \{\mathsf{E}.\mathtt{length} \mid \mathsf{E} \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \textit{if } r = |\tau|-1. \end{cases}$$

Rule #11*: If* ins = arrayload t[ ]*, then*

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \textit{if } r \neq |\tau|-2 \\ \{\mathsf{E}_1[\mathsf{E}_2] \mid \mathsf{E}_1 \in \mathsf{A}_{|\tau|-2} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_1, \mathsf{ins}) \wedge \\ \qquad \mathsf{E}_2 \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_2, \mathsf{ins}) \wedge \\ \qquad [\mathsf{E}_1 \textit{ and } \mathsf{E}_2 \textit{ do not invoke any method}]\} & \textit{if } r = |\tau|-2. \end{cases}$$

RULE #12*: If* ins = arraystore t[ ]*, then*

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}.$$

RULE #13*: If* ins = dup t*, then*

$$\mathsf{A}'_r = \begin{cases} \mathsf{A}_r \cup \mathsf{A}_r[s_j/s_{j-1}] & \text{if } r < |\tau|-1 \\ \mathsf{A}_{|\tau|-1} \cup \{s_j\} & \text{if } r = |\tau|-1 \\ \mathsf{A}_{|\tau|-1} \cup \{s_{j-1}\} & \text{if } r = |\tau|. \end{cases}$$

RULE #14*: If* ins $\in$ {ifeq t, ifne t, catch, exception_is K}*, then*

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}.$$

Let us now explain, in more detail, the propagation rules introduced in Definition 6.20. The sequential arcs link an instruction ins to its immediate successor ins′ propagating, for every variable v at ins′, all those expressions E aliased to v at ins that cannot be affected by ins itself, i.e., such that ¬canBeAffected(E, ins) holds. However, some new alias expressions might be added to the initial approximation as well. We discuss the rules introduced above:

const x - in this case, a new variable ($v_{|\tau|} = s_j$ is pushed onto the operand stack, and its value is x, while everything else stays unchanged. Therefore, for each variable available in dom(τ), we just propagate its current approximation, while the approximation related to $v_{|\tau|}$ is $\{x\}$.

load k t - In this case a new variable (s_j) is pushed onto the operand stack and its value is equal to that of l_k. Therefore, for each variable $v_r \in \mathsf{dom}(\tau)$, we propagate all the alias expressions already present in A_r and by using the fact that $l_k = s_j$, we also add all those alias expression from A_r obtained by replacing all the occurrences of l_k with s_j. Obviously, s_j and l_k become alias expressions of s_j and l_k respectively.

store k t - In this case the topmost variable is popped from the operand stack (s_{j-1}) and its value is assigned to l_k. Therefore, all the alias expressions involving l_k and s_{j-1} in the initial approximations A_r, for any $r \neq k$, should be removed from the final ones (by Definition 6.11, canBeAffected(E, store k t) = *true* if and only if l_k or s_{j-1} occurs in E). On the other hand, the final approximation related to l_k contains all the alias expressions $\mathsf{E} \in \mathsf{A}_{|\tau|-1}$ belonging to the initial approximation related to s_{j-1} which are not modified by the store k t, i.e, such that ¬canBeAffected(E, store k t) holds.

add, sub, mul, div, rem - in this case two topmost stack elements (memorized in s_{j-1} and s_{j-2}) of integer type are popped from the operand stack and the result of an opportune arithmetic operations applied to these two values is pushed back onto the operand stack (memorized in s_{j-2}). Therefore, for each variable v_r from dom(τ'), except s_{j-2}, we propagate all those alias expressions belonging to their initial approximations which might not be affected by this bytecode instruction, i.e., the ones not containing any occurrence of s_{j-1} and s_{j-2}. On the other hand, the expressions definitely aliased to the new topmost stack element s_{j-2} are of form $\mathsf{E}_1 \oplus \mathsf{E}_2$, where $\oplus$ is the arithmetic operation corresponding to this bytecode instruction, while E_1 and E_2 are expressions definitely aliased to s_{j-1} and s_{j-2} before the bytecode instruction is executed and which are not affected by this bytecode instruction.

inc k x - in this case the value memorized in the local variable l_k is incremented by x. Therefore, the final approximation related to each variable except l_k is composed of all the expressions available in the initial one in which all the occurrences of l_k are replaced with $l_k - x$. In the case of l_k, we soundly assume that there is no expression aliased to that variable, after the bytecode instruction is executed.

new κ - In this case a new object is created and the location it is bound to is pushed onto the operand stack, in s_j. Therefore, for each variable, except s_j, its initial approximation is kept. On the other hand, since s_j holds a fresh location, we soundly assume there is no expressions aliased to s_j.

getfield f - In this case the location memorized in the topmost operand stack element s_{j-1} is replaced with the value of the field f of the object corresponding to the former. Therefore, for each variable, except s_{j-1}, its final approximation contains all the alias expressions from the initial one which are not modified by this bytecode instruction, i.e., the ones with no occurrence of s_{j-1} (Definition 6.11). On the other hand, the final approximation related to s_{j-1} contains the expressions $\mathsf{E}.f$ where E is aliased to s_{j-1} before the bytecode instruction is executed, it cannot be modified by the latter ($\neg$canBeAffected(E, ins), i.e., E contains no occurrences of s_{j-1}) and no evaluation of E might modify the field f ($\neg$mightModify(E, $\{f\}$)).

putfield f - In this case the value memorized in the topmost operand stack element s_{j-1} is written in the field f of the object corresponding to the location memorized in the second topmost operand stack element s_{j-2}, and both s_{j-1} and s_{j-2} are popped from the operand stack. Hence, for each variable, we propagate all the alias expressions E belonging to its initial approximation which cannot be modified by this bytecode instruction, i.e., such that there is no occurrence of s_{j-2} and s_{j-1} in E and such that no evaluation of E might read the field f (Definition 6.11).

arraynew α - In this case the topmost operand stack element containing an integer value is replaced with the fresh location bound to the new created array. The propagation is similar to the case of new κ.

arraylength α - In this case the topmost operand stack element s_{j-1} containing a reference to an array is replaced with the length of that array. Therefore, for each variable, except s_{j-1}, its final approximation contains all the alias expressions from the initial one which are not modified by this bytecode instruction, i.e., the ones in which s_{j-1} does not appear (Definition 6.11). On the other hand, the expressions aliased to s_{j-1} are of the form E.`length`, where E is an alias expression of the old topmost stack element, which is not modified by this bytecode instruction.

arrayload α - In this case the k-th element of the array corresponding to the location memorized in the second topmost operand stack element s_{j-2}, where k is the topmost operand stack element, is written onto the top of the stack. Previously, both s_{j-1} and s_{j-2} are popped from the stack. The propagation rule and its explanation are analogous to the case of arithmetic operations.

arraystore α - In this case the value memorized in the topmost operand stack element s_{j-1} is written in the k-th element of the array corresponding to the location memorized in the third topmost operand stack element s_{j-3}, where k is the integer value memorized in the second topmost operand stack element. All s_{j-1}, s_{j-2} and s_{j-3} are popped from the operand stack. The propagation rule and its explanation are analogous to the case of putfield f.

dup t - In this case, the new topmost stack element s_j is a copy of the former topmost stack element s_{j-1}, hence they are trivially aliased to each other, and all the alias expressions of s_{j-1} at ins become the alias expressions of s_j at ins′. More precisely, we let $\mathsf{A}'_{|\tau|-1} = \mathsf{A}_{|\tau|-1} \cup \{s_j\}$ and $\mathsf{A}'_{|\tau|} = \mathsf{A}_{|\tau|-1} \cup \{s_{j-1}\}$. The approximations of the aliasing expressions of all other variables are enriched by all the expressions containing s_{j-1} already present in those approximations in which occurrences of s_{j-1} are repliced by s_j: $\mathsf{A}'_r = \mathsf{A}_r \cup \mathsf{A}_r[s_j/s_{j-1}]$.

otherwise - catch and exception_is K do not modify the initial state, and therefore do not change the definite expression aliasing information available at that point, while ifne t and ifeq t just pop the topmost operand stack element, and therefore do not modify the aliasing expressions of any other variable which contains no occurrence of s_{j-1}.

Example 6.21. In Fig. 6.5 we give the ACG of the method `delayMinBy` from Fig. 4.1. Nodes **a**, **b** and **c** belong to the caller of this method and exemplify the arcs related to the call and return bytecodes. Arcs are decorated with the number of their associated propagation rules. Note that the graph for the whole program includes other nodes and arcs. Fig. 6.5 only shows the portion that is relevant for our example.

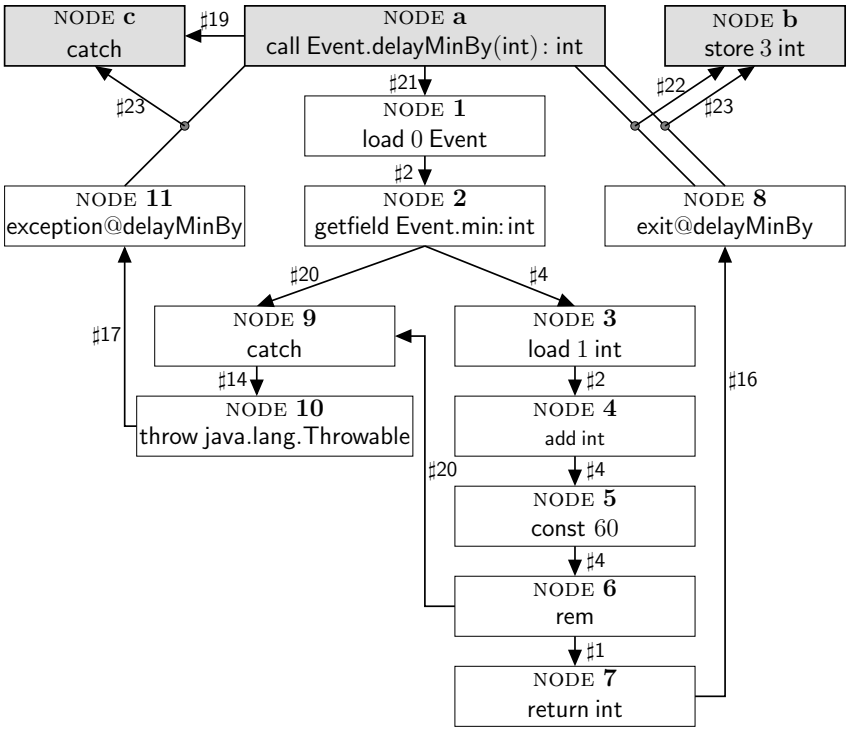


Fig. 6.5. The ACG for the method `delayMinBy` in Fig. 4.1

In the following examples for each node x we let τ_x, i_x and j_x denote the type environment, number of local variables and number of stack elements at x respectively. We let $A^x = \langle \mathsf{A}^x_0, \ldots, \mathsf{A}^x_{(i_x+j_x-1)} \rangle$ denote an approximation of the actual aliasing information

at x, where each A_r^x denotes a set of expressions definitely aliased to the local variable l_r if $0 \leq r < i_x$ or to a stack element s_{r-i_x} if $0 \leq r - i_x < j_x$. Moreover, we assume that $i_{\mathbf{a}} = 3$, $j_{\mathbf{a}} = 2$ and that the call at node **a** occurs in a context with

$$\mathsf{A}_0^{\mathbf{a}} = \varnothing, \mathsf{A}_1^{\mathbf{a}} = \varnothing, \mathsf{A}_2^{\mathbf{a}} = \{v_1.\mathsf{getFirst}(), v_3\}, \mathsf{A}_3^{\mathbf{a}} = \{v_1.\mathsf{getFirst}(), v_2\} \text{ and } \mathsf{A}_4^{\mathbf{a}} = \{15\}. \quad (6.1)$$

□

The following example illustrates an application of some of the propagation rules introduced by Definition 6.20.

Example 6.22. Consider, for instance, nodes **2**, **3**, **4** and **5** in Figure 6.5, and suppose that $i_{\mathbf{2}} = 2$ and $j_{\mathbf{2}} = 1$, i.e., at node **2** there are 2 local variables ($v_0 = l_0$ and $v_1 = l_1$) and 1 operand stack element ($v_2 = s_0$). Moreover, suppose that the variables v_0, v_1 and v_2 are respectively aliased to the following sets of expressions:

$$\mathsf{A}_0^{\mathbf{2}} = \{v_2\}, \quad \mathsf{A}_1^{\mathbf{2}} = \varnothing \quad \text{and } \mathsf{A}_2^{\mathbf{2}} = \{v_0\}.$$

Nodes **2** and **3** are linked by a sequential arc with propagation rule #7. It can be easily determined that $i_{\mathbf{3}} = 2$ and $j_{\mathbf{3}} = 1$. Note that at node **2**, by Definition 6.11, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{getfield}\ \mathtt{min})$ is equivalent to $v_2 \notin \mathsf{variables}(\mathsf{E})$, since getfield might only affect the values of the expressions containing the topmost stack element, i.e., v_2. Moreover, according to Definition 6.12,

$$\mathsf{mightModify}(v_0, \{\mathtt{min}\}) = \mathit{false}.$$

Using these facts and Definition 6.20 (Rule #7) we obtain:

$$\begin{aligned}
\mathsf{A}_0^{\mathbf{3}} &= \{\mathsf{E} \in \mathsf{A}_0^{\mathbf{2}} \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{getfield}\ \mathtt{min})\} = \{\mathsf{E} \in \{v_2\} \mid v_2 \notin \mathsf{variables}(\mathsf{E})\} = \varnothing \\
\mathsf{A}_1^{\mathbf{3}} &= \{\mathsf{E} \in \mathsf{A}_1^{\mathbf{2}} \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{getfield}\ \mathtt{min})\} = \{\mathsf{E} \in \varnothing \mid v_2 \notin \mathsf{variables}(\mathsf{E})\} = \varnothing \\
\mathsf{A}_2^{\mathbf{3}} &= \{\mathsf{E}.\mathtt{min} \mid \mathsf{E} \in \mathsf{A}_2^{\mathbf{2}} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{getfield}\ \mathtt{min}) \wedge \neg\mathsf{mightModify}(\mathsf{E}, \{\mathtt{min}\})\} \\
&= \{\mathsf{E}.\mathtt{min} \mid \mathsf{E} \in \{v_0\} \wedge v_2 \notin \mathsf{variables}(\mathsf{E}) \wedge \neg\mathsf{mightModify}(\mathsf{E}, \{\mathtt{min}\})\} = \{v_0.\mathtt{min}\}.
\end{aligned}$$

Thus,

$$\mathit{\Pi}^{\#7}(A^{\mathbf{2}}) = \langle \varnothing, \varnothing, \{v_0.\mathtt{min}\} \rangle. \quad (6.2)$$

Nodes **3** and **4** are linked by a sequential arc with propagation rule #2, and at node **4** we have $i_{\mathbf{4}} = j_{\mathbf{4}} = 2$. If we assume that $A^{\mathbf{3}} = \mathit{\Pi}^{\#7}(A^{\mathbf{2}})$ (Equation 6.2), then, by Definition 6.20 (Rule #2), we have:

$$\begin{aligned}
\mathsf{A}_0^{\mathbf{4}} &= \mathsf{A}_0^{\mathbf{3}} \cup \mathsf{A}_0^{\mathbf{3}}[v_3/v_1] = \varnothing \\
\mathsf{A}_1^{\mathbf{4}} &= \mathsf{A}_1^{\mathbf{3}} \cup \{v_3\} = \varnothing \cup \{v_3\} = \{v_3\} \\
\mathsf{A}_2^{\mathbf{4}} &= \mathsf{A}_2^{\mathbf{3}} \cup \mathsf{A}_2^{\mathbf{3}}[v_3/v_1] = \{v_0.\mathtt{min}\} \cup \{v_0.\mathtt{min}\}[v_3/v_1] = \{v_0.\mathtt{min}\} \\
\mathsf{A}_3^{\mathbf{4}} &= \mathsf{A}_1^{\mathbf{3}} \cup \{v_1\} = \varnothing \cup \{v_1\} = \{v_1\}
\end{aligned}$$

Thus,

$$\mathit{\Pi}^{\#2}(A^{\mathbf{3}}) = \langle \varnothing, \{v_3\}, \{v_0.\mathtt{min}\}, \{v_1\} \rangle. \quad (6.3)$$

Nodes **4** and **5** are linked by a sequential arc with propagation rule #4, and at node **5** we have $i_{\mathbf{5}} = 2$ and $j_{\mathbf{5}} = 1$. Note that at node **4**, by Definition 6.11, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{add})$ holds if and only if $\mathsf{variables}(\mathsf{E}) \cap \{v_2, v_3\} = \varnothing$, since add might only affect the values of

the expressions containing two topmost stack elements, i.e., v_2 and v_3 (v_3 is removed from the stack, and the sum of the values held in v_2 and v_3 is written back in v_2). If we assume that $A^{\mathbf{4}} = \Pi^{\#2}(A^{\mathbf{3}})$ (Equation 6.3), then, by Definition 6.20 (RULE #4), we have:

$$\begin{aligned}
\mathsf{A}_0^{\mathbf{5}} &= \{\mathsf{E} \in \mathsf{A}_0^{\mathbf{4}} \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{add})\} = \{\mathsf{E} \in \varnothing \mid v_2, v_3 \notin \mathsf{variables}(\mathsf{E})\} = \varnothing \\
\mathsf{A}_1^{\mathbf{5}} &= \{\mathsf{E} \in \mathsf{A}_1^{\mathbf{4}} \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{add})\} = \{\mathsf{E} \in \{v_3\} \mid v_2, v_3 \notin \mathsf{variables}(\mathsf{E})\} = \varnothing \\
\mathsf{A}_2^{\mathbf{5}} &= \{\mathsf{E}_1 + \mathsf{E}_2 \mid \mathsf{E}_1 \in \mathsf{A}_2^{\mathbf{4}} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_1, \mathsf{add}) \wedge \mathsf{E}_2 \in \mathsf{A}_3^{\mathbf{4}} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_2, \mathsf{add})\} \\
&= \{\mathsf{E}_1 + \mathsf{E}_2 \mid \mathsf{E}_1 \in \{v_0.\mathtt{min}\} \wedge \mathsf{variables}(\mathsf{E}_1) \cap \{v_2, v_3\} = \varnothing \\
&\qquad\qquad \wedge \mathsf{E}_2 \in \{v_1\} \wedge \mathsf{variables}(\mathsf{E}_2) \cap \{v_2, v_3\} = \varnothing\} \\
&= \{v_0.\mathtt{min} + v_1\}
\end{aligned}$$

Thus, $A^{\mathbf{5}} = \langle \varnothing, \varnothing, \{v_0.\mathtt{min} + v_1\} \rangle$. □

Definition 6.23 (Final arcs). *For each* return t *and* throw κ *occurring in a method or constructor* m*, there are* $1-1$ final arcs *from* $\boxed{\mathsf{return\ t}}$ *to* $\boxed{\mathsf{exit@}m}$ *and from* $\boxed{\mathsf{throw}\ \kappa}$ *to* $\boxed{\mathsf{exception@}m}$*, respectively, with a propagation rule*

$$\lambda\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau|-1} \rangle . \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{|\tau'|-1} \rangle,$$

where, for each $0 \leq r < |\tau'|$*,* A'_r *is defined by one of the following rules:*

RULE #15*: If* ins = return *void, then*

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\}.$$

RULE #16*: If* ins = return t*, then*

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\} & \text{if } r \neq i \\ \{\mathsf{E} \in \mathsf{A}_{|\tau|-1} \mid \mathsf{noStackElements}(\mathsf{E})\} & \text{if } r = i. \end{cases}$$

RULE #17*: If* ins = throw κ*, then*

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\} & \text{if } r \neq i \\ \varnothing & \text{if } r = i, \end{cases}$$

where noStackElements(E) *is true if and only if* variables(E) $\cap \{s_0, \ldots, s_{j-1}\} = \varnothing$*, i.e., if* E *contains no operand stack elements.*

The **final arcs** introduced in Definition 6.23 feed nodes $\boxed{\mathsf{exit@}m}$ and $\boxed{\mathsf{exception@}m}$ for each method or constructor m. They propagate, for each local variable l_k available at $\boxed{\mathsf{exit@}m}$ (respectively $\boxed{\mathsf{exception@}m}$), all those expressions aliased to l_k at a $\boxed{\mathsf{return}}$ (respectively $\boxed{\mathsf{throw}}$) in which no stack variable occurs. In the case of return t, with t ≠ void, the alias expressions of the only stack element at $\boxed{\mathsf{exit@}m}$ (i.e., $v_i = s_0$) are alias expressions of the topmost stack element at $\boxed{\mathsf{return\ t}}$ (s_{j-1}) with no stack elements. In the case of throw κ, we conservatively assume that no expression is aliased to the only stack element at $\boxed{\mathsf{exception@}m}$ ($v_i = s_0$).

Example 6.24. Consider nodes **7** and **8** in Fig. 6.5, which are linked by a final arc with propagation rule #16. It can be easily determined that at node **7**, $i_{\mathbf{7}} = 2$ and $j_{\mathbf{7}} = 1$, therefore, the only stack element there is $v_2 = s_0$, and noStackElements(E) holds if and

only if $v_2 \notin \mathsf{variables(E)}$. Moreover, at node **8** we have $i_{\mathbf{8}} = 2$ and $j_{\mathbf{8}} = 1$. If we assume that $A^{\mathbf{7}} = \langle \varnothing, \varnothing, \{(v_0.\mathtt{min} + v_1)\%60\}\rangle$, then, by Definition 6.23 (Rule #16), we have:

$$\begin{array}{l}
\mathsf{A}_0^{\mathbf{8}} = \{\mathsf{E} \in \mathsf{A}_0^{\mathbf{5}} \mid \mathsf{noStackElements(E)}\} = \{\mathsf{E} \in \varnothing \mid \mathsf{noStackElements(E)}\} = \varnothing \\
\mathsf{A}_1^{\mathbf{8}} = \{\mathsf{E} \in \mathsf{A}_1^{\mathbf{5}} \mid \mathsf{noStackElements(E)}\} = \{\mathsf{E} \in \varnothing \mid \mathsf{noStackElements(E)}\} = \varnothing \\
\mathsf{A}_2^{\mathbf{8}} = \{\mathsf{E} \in \mathsf{A}_2^{\mathbf{5}} \mid \mathsf{noStackElements(E)}\} = \{\mathsf{E} \in \{v_0.\mathtt{min} + v_1\} \mid v_2 \notin \mathsf{variables(E)}\} = \{v_0.\mathtt{min} + v_1\}
\end{array}$$

Thus,

$$\Pi^{\#16}(A^{\mathbf{7}}) = \langle \varnothing, \varnothing, \{v_0.\mathtt{min} + v_1\}\rangle. \tag{6.4}$$

□

Definition 6.25 (Exceptional arcs). *For each* ins *throwing an exception, immediately followed by a* catch*, an arc is built from* $\boxed{\mathsf{ins}}$ *to* $\boxed{\mathsf{catch}}$*, with a propagation rule*

$$\lambda\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau|-1}\rangle.\langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{|\tau'|-1}\rangle,$$

where, for each $0 \le r < |\tau'|$, A'_r *is defined by the following rules:*

Rule #18*: If* ins = throw κ*, then:*
Rule #19*: If* ins = call $m_1 \ldots m_n$*, then:*
Rule #20*: If* ins *is one of the following bytecode instructions:* div, rem, new κ, getfield f, putfield f, arraynew α, arraylength α, arrayload α *or* arraystore α*, then:*

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements(E)}\} & \text{if } r \neq i \\ \varnothing & \text{if } r = i, \end{cases}$$

where noStackElements(E) *is true if and only if* variables(E) $\cap \{s_0, \ldots, s_{j-1}\} = \varnothing$*, i.e., if* E *contains no operand stack elements.*

The exceptional arcs link every instruction that might throw an exception to the catch at the beginning of their exception handler(s). They propagate alias expressions of local variables analogously to the final arcs. For the only stack element ($v_i = s_0$), holding the thrown exception, there is no alias expression ($\mathsf{A}_i = \varnothing$).

Example 6.26. Consider nodes **2** and **9** in Fig. 6.5, which are linked by an exceptional arc with propagation rule #20. Recall that at node **2**, $i_{\mathbf{5}} = 2$ and $j_{\mathbf{2}} = 1$, therefore, the only stack element there is $v_2 = s_0$ and noStackElements(E) holds if and only if $v_2 \notin \mathsf{variables(E)}$. By Definition 6.25 and hypotheses about $\mathsf{A}_0^{\mathbf{2}}$, $\mathsf{A}_1^{\mathbf{2}}$ and $\mathsf{A}_2^{\mathbf{2}}$ given in Example 6.22 we obtain:

$$\begin{array}{l}
\mathsf{A}_0^{\mathbf{7}} = \{\mathsf{E} \in \mathsf{A}_0^{\mathbf{2}} \mid \mathsf{noStackElements(E)}\} = \{\mathsf{E} \in \varnothing \mid \mathsf{noStackElements(E)}\} = \varnothing \\
\mathsf{A}_1^{\mathbf{7}} = \{\mathsf{E} \in \mathsf{A}_1^{\mathbf{2}} \mid \mathsf{noStackElements(E)}\} = \{\mathsf{E} \in \varnothing \mid \mathsf{noStackElements(E)}\} = \varnothing \\
\mathsf{A}_2^{\mathbf{6}} = \varnothing
\end{array}$$

Thus, $\Pi^{\#20}(A^{\mathbf{2}}) = \langle \varnothing, \varnothing, \varnothing\rangle$. □

Definition 6.27 (Parameter passing arcs). *For each* call $m_1 \ldots m_q$ *with* π *parameters (including the implicit parameter* this*), for each* $1 \le w \le q$ *we build an* $1-1$ parameter parssing arc *from* $\boxed{\mathsf{call}\ m_1 \ldots m_q}$ *to the node corresponding to the first bytecode of* m_w*, with the propagation rule* Rule #21*:*

$$\lambda\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau|-1}\rangle.\langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{\pi-1}\rangle,$$

where, for each $0 \le r < \pi$, $\mathsf{A}'_r = \varnothing$.

In the following we give some auxiliary definitions, which are necessary for the definition of the propagation rules for return value and side-effects arcs. We start with a map noParameters(), which specifies whether there exists an actual argument of a method call among the variables appearing in an expression.

Definition 6.28 (noParameters)**.** *Consider a type environment $\tau \in \mathcal{T}$ related to a program point with a method call* call $m_1 \dots m_n$*, and suppose that this method has π actual arguments (including the implicit parameter* `this`*). For every expression* $\mathsf{E} \in \mathbb{E}_\tau$*, we define a map* noParameters : $\mathbb{E}_\tau \to \{true, false\}$ *as:*

$$\mathsf{noParameters}(\mathsf{E}) = \mathsf{variables}(\mathsf{E}) \cap \{v_{|\tau|-\pi}, \dots, v_{|\tau|-1}\} = \emptyset$$

(we recall that variables $v_{|\tau|-\pi}, \dots, v_{|\tau|-1}$ correspond to π topmost operand stack elements).

The following definition specifies when the executions of a method are safe for an alias expression available at the point that method is invoked.

Definition 6.29 (safeExecution)**.** *Consider a type environment $\tau \in \mathcal{T}$ related to a program point with a method call* ins_C = call $m_1 \dots m_n$*, and suppose that this method has π actual arguments (including the implicit parameter* `this`*). For every expression* $\mathsf{E} \in \mathbb{E}_\tau$*, we define a map* $\mathsf{safeExecution}(\cdot, \mathsf{ins}_C) : \mathbb{E}_\tau \to \{true, false\}$ *as:*

$$\mathsf{safeExecution}(\mathsf{E}, \mathsf{ins}_C) = \mathsf{noParameters}(\mathsf{E}) \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}_C).$$

Namely, we say that *execution of* ins_C *is safe for an expression* E, if all possible executions of all the dynamic targets m_i of ins_C definitely do not affect E (i.e., $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}_C)$ holds) and if no actual parameter of ins_C appears in E (i.e., noParameters(E) holds). The former requires that every field that might be read by E must not be modified by any execution of any dynamic target m_i of ins_C, and that no execution of any dynamic target m_i of ins_C might write into an array whose elements might be read by E (Definition 6.11). The latter is required since the actual parameters of ins_C disappear from the operand stack after ins_C is executed.

Let us now characterize when the executions of a method are safe for an alias expression composed of different sub-expressions aliased to the actual arguments of the method call.

Definition 6.30 (safeAlias)**.** *Consider a type environment $\tau \in \mathcal{T}$ related to a program point with a method call* ins_C = call $m_1 \dots m_n$*, and suppose that this method has π actual arguments (including the implicit parameter* `this`*). For every expression* $\mathsf{E} \in \mathbb{E}_\tau$ *such that there exists a sub-expression* E_i *of* E *for each actual argument $v_{|\tau|-\pi+i}$ of* ins_C*, and each approximation $A = \langle \mathsf{A}_0, \dots, \mathsf{A}_{|\tau|-1} \rangle \in \textsc{Alias}_\tau$, we define a map* $\mathsf{safeAlias}(\cdot, A, \mathsf{ins}_C) : \mathbb{E}_\tau \to \{true, false\}$ *as:*

$$\mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_C) = \textstyle\bigwedge_{i=0}^{\pi-1}(\mathsf{E}_i \in \mathsf{A}_{|\tau|-\pi+i}) \wedge \bigwedge_{i=0}^{\pi-1} \mathsf{safeExecution}(\mathsf{E}_i, \mathsf{ins}_c) \wedge$$
$$[\text{no evaluation of } \mathsf{E} \text{ might modify any field from } \mathsf{fields}(\mathsf{E})$$
$$\text{or any array element of type t if } \mathsf{E} \text{ also might read}$$
$$\text{an array element of type t}' \text{ where t}' \in \mathsf{compatible}(\mathsf{t})].$$

Hence, an *alias expression* E *composed of sub-expressions* $\mathsf{E}_0, \dots, \mathsf{E}_{\pi-1}$ *is safe w.r.t.* $A = \langle \mathsf{A}_0, \dots, \mathsf{A}_{|\tau|-1} \rangle$ *and* ins (i.e., $\mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins})$ holds) if the following conditions hold:

- according to A, for each $0 \leq i < \pi$, the actual parameter $v_{|\tau|-\pi+i}$, is aliased to E_i;
- for each $0 \leq i < \pi$, ins_C is safe for E_i, i.e., $\mathsf{safeExecution}(\mathsf{E}_i, \mathsf{ins}_C)$ holds;
- no field and no array element might be both read and modified during all possible evaluations of E.

Finally, we specify when an alias expression of the returned value of a method available at the non-exceptional end of that method is safe at that point.

Definition 6.31 (safeReturn)**.** *Consider a type environment $\tau \in \mathcal{T}$ related to a non-exceptional end of a method m, and suppose that this method has π formal arguments (including the implicit parameter* `this`*). For every expression $\mathsf{R} \in \mathbb{E}_\tau$, we define a map $\mathsf{safeRetrun}(\cdot, m) : \mathbb{E}_\tau \to \{true, false\}$ as:*

$$\mathsf{safeReturn}(\mathsf{R}, m) = \mathsf{variables}(\mathsf{R}) \subseteq \{l_0, \ldots, l_{\pi-1}\} \wedge \forall l_k \in \mathsf{variables}(\mathsf{R}),\ l_k \textit{ is not modified by } m$$

(we recall that the formal arguments of a callee are held in the local variables $l_0, \ldots, l_{\pi-1}$).

We say that an *alias expression* R *of a return value at a non-exceptional end of a calee m is safe at that point* (i.e., $\mathsf{safeReturn}(\mathsf{R}, m)$ holds) if only local variables holding the formal arguments of m ($l_0, \ldots, l_{\pi-1}$) appear in R and none of them might be modified by m. The latter condition requires that for each $l_k \in \mathsf{variables}(\mathsf{R})$, no $\mathsf{store}\ k\ \mathsf{t}$ nor $\mathsf{inc}\ k\ x$ occurs in m.

We can finally define the propagation rules for the return value and side-effects arcs of the ACGs for the definite expression aliasing analysis.

Definition 6.32 (Return value arcs). *For each $\mathsf{ins}_C = \mathsf{call}\ m_1 \ldots m_q$ to a method with π actual arguments (including the implicit parameter* `this`*) returning a value of type $\mathsf{t} \neq \mathsf{void}$, and each subsequent bytecode instruction ins_N distinct from* catch*, we build, for each $1 \leq w \leq q$, a $\mathsf{2{-}1}$* return value arc *from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit@}m_w}$ (2 sources, in that order) to $\boxed{\mathsf{ins}_N}$. Suppose that the static type information at $\boxed{\mathsf{ins}_C}$, $\boxed{\mathsf{exit@}m_w}$ and $\boxed{\mathsf{ins}_N}$ are τ_C, τ_E and τ_N, respectively. Given $A = \langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau_C|-\pi}, \ldots, \mathsf{A}_{|\tau_C|-1} \rangle \in \textsc{Alias}_{\tau_C}$ and $R = \langle \mathsf{R}_0, \ldots, \mathsf{R}_{|\tau_E|-1} \rangle \in \textsc{Alias}_{\tau_E}$, the propagation rule of these arcs is defined as:*

$$\lambda A, R.\langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{|\tau_C|-\pi} \rangle,$$

where for each $0 \leq r \leq |\tau_C| - \pi$, A'_r is defined by the Rule #22*:*

$$\mathsf{A}'_r = \begin{cases} \mathsf{A}_r & \textit{if } r \neq |\tau_C|-\pi \\ \{\mathsf{E} = \mathsf{R}[\mathsf{E}_0, \ldots, \mathsf{E}_{\pi-1}/l_0, \ldots, l_{\pi-1}] \mid \mathsf{R} \in \mathsf{R}_{|\tau_E|-1} \wedge \mathsf{safeReturn}(\mathsf{R}, m_w) \wedge \mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_C)\} & \\ \cup\ \{\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_{\pi-1}) \mid \mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_C)\} & \textit{if } r = |\tau_C|-\pi. \end{cases}$$

Definition 6.33 (Side-effects arcs). *For each $\mathsf{ins}_C = \mathsf{call}\ m_1 \ldots m_q$ to a method with π actual arguments (including the implicit parameter* `this`*), and each subsequent bytecode instruction ins_N, we build, for each $1 \leq w \leq q$, a $\mathsf{2{-}1}$* side-effects arc *from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit@}m_w}$ (2 sources, in that order) to $\boxed{\mathsf{ins}_N}$, if ins_N is not a* catch *and a $\mathsf{2{-}1}$* side-effects arc *from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exception@}m_w}$ (2 sources, in that order) to $\boxed{\mathsf{catch}}$. Suppose that the static type information at $\boxed{\mathsf{ins}_C}$, $\boxed{\mathsf{exit@}m_w}$ (or $\boxed{\mathsf{exception@}m_w}$) and $\boxed{\mathsf{ins}_N}$ are τ_C, τ_E and τ_N respectively. The propagation rule of these arcs is defined as:*

$$\lambda\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau_C|-\pi}, \ldots, \mathsf{A}_{|\tau_C|-1}\rangle, \langle \mathsf{R}_0, \ldots, \mathsf{R}_{|\tau_E|-1}\rangle.\langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{|\tau_N|-1}\rangle,$$

where for each $0 \leq r < |\tau_N|$, A'_r *is defined by the* RULE #23*:*

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{safeExecution}(\mathsf{E}, \mathsf{ins}_C)\} & \text{if } r \neq |\tau_C|-\pi \\ \mathbb{E}_{\tau_N} & \text{if } r = |\tau_C|-\pi. \end{cases}$$

There exists a return value arc for each dynamic target m_w of a call ins_C returning a value. RULE #22) considers $\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau_C|-1}\rangle$ and $\langle \mathsf{R}_0, \ldots, \mathsf{R}_{|\tau_E|-1}\rangle$, approximations at $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit@}m_w}$, and builds the alias expressions related to the returned value $v_{|\tau_C|-\pi} = v_{|\tau_N|-1}$ at the node corresponding to the instruction which follows the call, $\boxed{\mathsf{ins}_N}$. An alias expression $\mathsf{R} \in \mathsf{R}_{|\tau_E|-1}$ of the computed value $v_{|\tau_E|-1}$ at $\boxed{\mathsf{exit@}m_w}$ can be turned into an alias expression of $v_{|\tau_C|-\pi}$ at $\boxed{\mathsf{ins}_N}$ if

1. R is safe at m_w and
2. every occurrence of a formal parameter l_k in R is replaced by an alias expression $\mathsf{E}_k \in \mathsf{A}_{|\tau_C|-\pi+k}$ of the corresponding actual parameter $v_{|\tau_C|-\pi+k}$ at $\boxed{\mathsf{ins}_C}$, which is safe w.r.t. ins_C.

Moreover, $\mathsf{E} = \mathsf{E}_0.m_w(\mathsf{E}_1, \ldots, \mathsf{E}_{\pi-1})$ can be an alias of $v_{|\tau_C|-\pi}$ at $\boxed{\mathsf{ins}_N}$ if it is safe w.r.t. ins_C.

The side-effects arcs consider the alias expressions E of the variables v_r different from the actual parameters $(v_{|\tau_C|-\pi}, \ldots, v_{|\tau_C|-1})$ of the method at $\boxed{\mathsf{ins}_C}$ and insert them among the alias expressions of v_r also at $\boxed{\mathsf{ins}_N}$ if they are safe w.r.t. ins_C.

Example 6.34. Nodes **a** and **b** are linked to node **b** through a return value and a side-effects arc with propagation rules #22 and #23. We recall that $i_{\mathbf{a}} = 3$, $j_{\mathbf{a}} = \pi = 2$ (Example 6.21), $i_{\mathbf{6}} = 2$, $j_{\mathbf{6}} = 1$ (Example 6.24) and the static type of the returned value of `delayMinBy` is void. Therefore, $i_{\mathbf{b}}$ and $j_{\mathbf{b}}$ have to be 3 and 1 respectively. Let ins_C denote the call to `delayMinBy` at node **a** and let us first consider the return value arc and let us assume that $A^{\mathbf{a}} = \langle \mathsf{A}^{\mathbf{a}}_0, \mathsf{A}^{\mathbf{a}}_1, \mathsf{A}^{\mathbf{a}}_2, \mathsf{A}^{\mathbf{a}}_3, \mathsf{A}^{\mathbf{a}}_4\rangle$ (Equation 6.1) and $A^{\mathbf{8}} = \Pi^{\#16}(A^{\mathbf{7}})$ (Equation 6.4). Application of the propagation rule #22 (Definition 6.32), on the pair $(A^{\mathbf{a}}, A^{\mathbf{8}})$ gives: $\mathsf{A}^{\mathbf{b}}_r = \mathsf{A}^{\mathbf{a}}_r$ for $0 \leq r \leq 2$ and

$$\begin{aligned} \mathsf{A}^{\mathbf{b}}_3 = \{\mathsf{E} = \mathsf{R}[\mathsf{E}_0, \mathsf{E}_1/l_0, l_1] \mid \mathsf{R} \in \mathsf{A}^{\mathbf{8}}_2 \wedge \mathsf{safeReturn}(\mathsf{R}, \texttt{delayMinBy}) \wedge \mathsf{safeAlias}(\mathsf{E}, A^{\mathbf{a}}, \mathsf{ins}_C)\} \\ \cup \{\mathsf{E} = \mathsf{E}_0.\texttt{delayMinBy}(\mathsf{E}_1) \mid \mathsf{safeAlias}(\mathsf{E}, A^{\mathbf{a}}, \mathsf{ins}_C)\}. \end{aligned} \tag{6.5}$$

Let us consider the alias expression $\mathsf{R} = (v_0.\texttt{min} + v_1)\%60$, which is aliased to v_2 at node **8**, i.e.,

$$\mathsf{R} \in \mathsf{A}^{\mathbf{8}}_2. \tag{6.6}$$

It is clear that only formal parameters of `delayMinBy` (l_0 and l_1) appear in R. Moreover, it is possible to statically determine that `delayMinBy` does not modify l_0 and l_1 since that method contains no store nor inc instructions (Fig. 3.3). Thus,

$$\mathsf{safeReturn}(\mathsf{R}, \texttt{delayMinBy}) = \mathit{true}. \tag{6.7}$$

It is worth noting that the formal parameters l_0 and l_1 (node **1**) correspond to the actual parameters $v_3 = s_0$ and $v_4 = s_1$ (node **a**) of `delayMinBy`. By (6.5), an alias expression E of $v_3 = s_0$ at node **b** (holding the returned value of `delayMinBy`) can be obtained by

substituting all occurrences of l_0 and l_1 in R by alias expressions E_0 and E_1 of s_0 and s_1 at node **a** respectively, only if E is safe w.r.t. ins_C, i.e., if $\mathsf{safeAlias}(\mathsf{E}, \mathsf{A}^{\mathbf{a}}, \mathsf{ins}_C)$ holds. By hypotheses introduced in Example 6.21, the alias expressions of s_0 at **a** are v_2 and v_1.`getFirst`(), while the only alias expression of s_1 at **a** is 15. These expressions contain no actual parameter of the call at node **a**. Let us show that $\mathsf{E} = \mathsf{R}[v_1.\mathsf{getFirst}(), 15/l_0, l_1] = (v_1.\texttt{getFirst}().\texttt{min} + 15)\%60$ is safe w.r.t. ins_C. First of all we have:

$$v_1.\texttt{getFirst}() \in \mathsf{A}_3^{\mathbf{a}} \text{ and } 15 \in \mathsf{A}_4^{\mathbf{a}}. \tag{6.8}$$

It is clear that no execution of `delayMin` might modify the evaluation of 15, since the latter is a constant. Thus,

$$\mathsf{safeExecution}(15, \mathsf{ins}_C) = \mathit{true}. \tag{6.9}$$

On the other hand, `delayMinBy` does not modify any field (Fig. 4.1), and therefore, it never modifies any field that might be read during any evaluation of v_1.getFirst(), which implies that $\neg\mathsf{canBeAffected}(v_1.\mathsf{getFirst}(), \mathsf{ins}_C)$ holds, and therefore

$$\mathsf{safeExecution}(v_1.\mathsf{getFirst}(), \mathsf{ins}_C) = \mathit{true}. \tag{6.10}$$

Finally, `getFirst` reads no array element and, by Definition 6.12, modifies no fields:

$$\begin{aligned}
&\mathsf{mightModify}(v_1.\texttt{getFirst}().\texttt{min} + 15, \mathsf{fields}(v_1.\texttt{getFirst}().\texttt{min} + 15))\\
&= \mathsf{mightModify}(v_1.\texttt{getFirst}().\texttt{min}, F) \vee \mathsf{mightModify}(15, F)\\
&= \mathsf{mightModify}(v_1.\texttt{getFirst}(), F) \vee \mathit{false}\\
&= \mathit{false},
\end{aligned} \tag{6.11}$$

where $F = \{\texttt{List.head:Object}, \texttt{Event.min:int}\}$. From Equations 6.9, 6.10 and 6.11 we obtain $\mathsf{safeAlias}(\mathsf{E}, \mathsf{A}^{\mathbf{a}}, \mathsf{ins}_C) = \mathit{true}$, which, together with (6.6) and (6.7) implies that E is an alias expression of v_3 at node **b**. We can similarly show that also the alias expression $\mathsf{R}[v_2, 15/l_0, l_1] = (v_2.\texttt{min} + 15)\%60$ is an alias expression of v_3 at **b**. It can be easily shown that also v_1.`getFirst`().`delayMinBy`(15) and v_2.`delayMinBy`(15) are safe w.r.t. ins_C. Namely,

- $v_1.\texttt{getFirst}() \in \mathsf{A}_3^{\mathbf{a}}$ and $15 \in \mathsf{A}_4^{\mathbf{a}}$;
- $\mathsf{safeAlias}(v_1.\texttt{getFirst}(), A^{\mathbf{a}}, \mathsf{ins}_C)$ holds (see (6.10);
- $\mathsf{safeAlias}(15, A^{\mathbf{a}}, \mathsf{ins}_C)$ holds (see (6.9);
- `getFirst` reads no array elements and by (6.6) modifies no fields that might be read by v_1.`getFirst`() and 15.

Hence, $\mathsf{safeAlias}(v_1.\texttt{getFirst}.\texttt{delayMinBy}(15), A^{\mathbf{a}}, \mathsf{ins}_C)$ holds. We can similarly show that also $\mathsf{safeAlias}(v_2.\texttt{min} + 15, A^{\mathbf{a}}, \mathsf{ins}_C)$ holds. Therefore, Rule #22 gives rise to the following aliasing information:

$$\begin{aligned}
\mathsf{A}_0^{\mathbf{b}'} &= \mathsf{A}_1^{\mathbf{b}'} = \varnothing\\
\mathsf{A}_2^{\mathbf{b}'} &= \{v_3, v_1.\mathsf{getFirst}()\}\\
\mathsf{A}_3^{\mathbf{b}'} &= \{(v_1.\texttt{getFirst}().\texttt{min}+15)\%60, v_1.\texttt{getFirst}().\texttt{delayMinBy}(15),\\
&\qquad (v_2.\texttt{min}+15)\%60, v_2.\texttt{delayMinBy}(15)\}.
\end{aligned}$$

On the other hand, rule #23 states that an alias expression E of any local variable l_k at node **b** is an alias expressions of the same variable at node **a** if E is safe w.r.t.

$\Pi^{\#21}(\mathsf{A}^{\mathbf{a}}) \sqsubseteq \mathsf{A}^{\mathbf{1}}$	$\Pi^{\#4}(\mathsf{A}^{\mathbf{4}}) \sqsubseteq \mathsf{A}^{\mathbf{5}}$	$\Pi^{\#17}(\mathsf{A}^{\mathbf{8}}) \sqsubseteq \mathsf{A}^{\mathbf{9}}$
$\Pi^{\#2}(\mathsf{A}^{\mathbf{1}}) \sqsubseteq \mathsf{A}^{\mathbf{2}}$	$\Pi^{\#16}(\mathsf{A}^{\mathbf{5}}) \sqsubseteq \mathsf{A}^{\mathbf{6}}$	$\Pi^{\#22}(\mathsf{A}^{\mathbf{a}}, \mathsf{A}^{\mathbf{6}}) \sqcup \Pi^{\#23}(\mathsf{A}^{\mathbf{a}}, \mathsf{A}^{\mathbf{6}}) \sqsubseteq \mathsf{A}^{\mathbf{b}}$
$\Pi^{\#7}(\mathsf{A}^{\mathbf{2}}) \sqsubseteq \mathsf{A}^{\mathbf{3}}$	$\Pi^{\#20}(\mathsf{A}^{\mathbf{2}}) \sqsubseteq \mathsf{A}^{\mathbf{7}}$	$\Pi^{\#19}(\mathsf{A}^{\mathbf{a}}) \sqcup \Pi^{\#23}(\mathsf{A}^{\mathbf{a}}, \mathsf{A}^{\mathbf{9}}) \sqsubseteq \mathsf{A}^{\mathbf{c}}$
$\Pi^{\#2}(\mathsf{A}^{\mathbf{3}}) \sqsubseteq \mathsf{A}^{\mathbf{4}}$	$\Pi^{\#14}(\mathsf{A}^{\mathbf{7}}) \sqsubseteq \mathsf{A}^{\mathbf{8}}$	

Fig. 6.6. Constraints generated from the ACG from Fig. 6.5

the executions of `delayMinBy` (i.e., if $\mathsf{safeExecution}(\mathsf{E}, \mathsf{ins}_C)$), while any expression can be an alias of v_3 at node **b**. It is obvious that ins_C is not safe for $v_3 \in \mathsf{A}_2^{\mathbf{a}}$ (since $\mathsf{noParameters}(v_3) = \mathit{false}$), while it is safe for $v_1.\mathsf{getFirst}()$, like we have already shown above (see (6.10)). Therefore, rule #23 gives rise to the following aliasing information:

$$\begin{aligned}\mathsf{A}_0^{\mathbf{b}''} &= \mathsf{A}_1^{\mathbf{b}''} = \varnothing\\ \mathsf{A}_2^{\mathbf{b}''} &= \{v_1.\mathsf{getFirst}()\}\\ \mathsf{A}_3^{\mathbf{b}''} &= \mathbb{E}_{\tau_{\mathbf{b}}}\end{aligned}$$

Hence, there are two arcs leading to node **b**, and bringing aliasing approximations $A^{\mathbf{b}'}$ and $A^{\mathbf{b}''}$, which permit us to compute the aliasing information at node **b** as

$$\begin{aligned}A^{\mathbf{b}} &= A^{\mathbf{b}'} \sqcup A^{\mathbf{b}''}\\ &= \langle \mathsf{A}_0^{\mathbf{b}'} \cap \mathsf{A}_0^{\mathbf{b}''}, \mathsf{A}_1^{\mathbf{b}'} \cap \mathsf{A}_1^{\mathbf{b}''}, \mathsf{A}_2^{\mathbf{b}'} \cap \mathsf{A}_2^{\mathbf{b}''}, \mathsf{A}_3^{\mathbf{b}'} \cap \mathsf{A}_3^{\mathbf{b}''} \rangle\\ &= \langle \varnothing, \varnothing, \mathsf{A}_2^{\mathbf{b}''}, \mathsf{A}_3^{\mathbf{b}'} \rangle.\end{aligned}$$

□

At this point we can formally define our definite expression aliasing analysis in the general framework of constraint-based analyses.

Definition 6.35 (Definite Expression Aliasing Analysis). Definite Expression Aliasing Analysis *is a* system of constraints *extracted from the* ACG *whose nodes are enriched with elements of the abstract domain* $\textsc{Alias}_\tau^d$*, where* τ *is the type environment corresponding to the node,* $d \in \mathbb{N}$ *is the fixed expression depth, and whose arcs are enriched with the propagation rules* #1 − #23 *introduced in Definitions 6.20 - 6.33. The extraction of constraints and the generation of the system of constraints concerning an* ACG *is explained in Section 4.5.*

Example 6.36. In Fig. 6.6 we provide the constraints extracted from the ACG introduced in Fig. 6.5. □

6.4 Soundness of the Definite Expression Aliasing Analysis

The goal of this section is to prove that there exists a solution to the system of constraints extracted from the ACG for the definite expression aliasing analysis that we defined in the previous section, and that this solution is sound. Since we follow the constraint-based approach defined in Chapter 4, if we prove that the requirements provided in the latter (Requirements 4.1- 4.11) hold, then the results obtained in Sections 4.5 and 4.6 guarantee

the existence of the least solution of the system of constraints mentioned above, as well as its soundness. The following subsections show that the instantiation of the general parameterized framework for constraint-based statyc analyses of Java bytecode program that we present in this section, i.e., the abstract domain Alias and the propagation rules introduced in Definitions 6.20, 6.23, 6.25, 6.27, 6.32 and 6.33 satisfy those requirements.

6.4.1 ACC Condition

This requirement is one of the conditions which guarantee the existence and the uniqueness of the least solution of our analyses, like Theorem 4.11 has shown. The following lemma shows that the abstract domain $\textsc{Alias}_\tau^d$, and therefore our definite expression aliasing analysis satisfy it.

Lemma 6.37. *The abstract domain* Alias *satisfies Requirement 4.1. More precisely, given a type environment $\tau \in \mathcal{T}$ and an integer $d \in \mathbb{N}$, every ascending chain of elements in* $\textsc{Alias}_\tau^d$ *eventually stabilizes.*

Proof. We recall that $\textsc{Alias}_\tau^d = \langle \mathcal{A}_\tau, \sqsubseteq, \bot_\tau^d, \top_\tau^d, \sqcup, \sqcap \rangle$, where

- $\mathcal{A}_\tau = (\wp(\mathbb{E}_\tau^d))^{|\tau|}$, where $\mathbb{E}_\tau^d = \{\mathsf{E} \in \mathbb{E}_\tau \mid \mathsf{depth}(\mathsf{E}) \leq d\}$;
- for every $A^1 = \langle \mathsf{A}_0^1, \ldots, \mathsf{A}_{|\tau|-1}^1 \rangle \in \mathcal{A}_\tau$ and $A^2 = \langle \mathsf{A}_0^2, \ldots, \mathsf{A}_{|\tau|-1}^2 \rangle \in \mathcal{A}_\tau$,

$$A^1 \sqsubseteq A^2 \Leftrightarrow \forall 0 \leq i < |\tau|, \mathsf{A}_i^1 \supseteq \mathsf{A}_i^2;$$

- for every $A^1 = \langle \mathsf{A}_0^1, \ldots, \mathsf{A}_{|\tau|-1}^1 \rangle \in \mathcal{A}_\tau$ and $A^2 = \langle \mathsf{A}_0^2, \ldots, \mathsf{A}_{|\tau|-1}^2 \rangle \in \mathcal{A}_\tau$,

$$A^1 \sqcup A^2 = \langle \mathsf{A}_0^1 \cap \mathsf{A}_0^2, \ldots, \mathsf{A}_{|\tau|-1}^1 \cap \mathsf{A}_{|\tau|-1}^2 \rangle;$$

- for every $A^1 = \langle \mathsf{A}_0^1, \ldots, \mathsf{A}_{|\tau|-1}^1 \rangle \in \mathcal{A}_\tau$ and $A^2 = \langle \mathsf{A}_0^2, \ldots, \mathsf{A}_{|\tau|-1}^2 \rangle \in \mathcal{A}_\tau$,

$$A^1 \sqcap A^2 = \langle \mathsf{A}_0^1 \cup \mathsf{A}_0^2, \ldots, \mathsf{A}_{|\tau|-1}^1 \cup \mathsf{A}_{|\tau|-1}^2 \rangle;$$

- $\top_\tau^d = \varnothing^{|\tau|}$;
- $\bot_\tau^d = (\mathbb{E}_\tau^d)^{|\tau|}$.

By Definition 6.17, $\bot_\tau^d$, i.e., the bottom element of $\textsc{Alias}_\tau^d$ is finite since it might contain only expressions whose depth is at most d, and variables in $\mathsf{dom}(\tau)$, field and method names are finite. Moreover, when $A^1 \sqsubseteq A^2$, it means that for each variable $v_r \in \mathsf{dom}(\tau)$, A_r^2 (the approximation A^2 assigns to v_r) is included in A_r^1 (the approximation A^1 assigns to v_r), i.e., $\mathsf{A}_r^2 \subseteq \mathsf{A}_r^1$. It means that greater elements in an ascending chain assign less alias expressions to each variable. It is worth noting that the least and the greatest elements an ascending chain might have are respectively $\bot_\tau^d$ (which is finite) and $\top_\tau^d = \varnothing^{|\tau|}$, which implies that this ascending chain eventually stabilizes, i.e., $\textsc{Alias}_\tau^d$ satisfies the ACC condition. Hence, Requirement 4.1 is satisfied. ∎

6.4.2 Galois Connection

The next step is to show that the concretization map γ that we introduced in Definition 6.18 gives rise to a Galois connection (Chapter 2). This result would imply the satisfiability of Requirement 4.2. We start this proof by showing that our γ map is co-additive.

Lemma 6.38. *For any type environment $\tau \in \mathcal{T}$, the function γ_τ is co-additive, i.e.,*

$$\gamma_\tau(\bigsqcap_{i\geq 0} A^i) = \bigcap_{i\geq 0} \gamma_\tau(A^i).$$

Proof. Consider a family of abstract elements $\{\langle \mathsf{A}_0^i, \ldots, \mathsf{A}_{n-1}^i\rangle\}_i$, where $n = |\tau|$. Then,

$$\begin{aligned}
\gamma_\tau(\bigsqcap_i \langle \mathsf{A}_0^i, \ldots, \mathsf{A}_{n-1}^i\rangle) &= \gamma_\tau(\langle \bigcup_i \mathsf{A}_0^i, \ldots \bigcup_i \mathsf{A}_{n-1}^1\rangle) \\
&= \{\sigma \in \Sigma_\tau \mid \forall 0 \leq r < n. \forall \mathsf{E} \in \bigcup_i \mathsf{A}_r^i. [\![\mathsf{E}]\!]\sigma = \rho(v_r)\} \\
&= \{\sigma \in \Sigma_\tau \mid \forall 0 \leq r < n. \forall i. \forall \mathsf{E} \in \mathsf{A}_r^i. [\![\mathsf{E}]\!]\sigma = \rho(v_r)\} \\
&= \{\sigma \in \Sigma_\tau \mid \forall i. \forall 0 \leq r < n. \forall \mathsf{E} \in \mathsf{A}_r^i. [\![\mathsf{E}]\!]\sigma = \rho(v_r)\} \\
&= \bigcap_i \{\sigma \in \Sigma_\tau \mid \forall 0 \leq r < n. \forall \mathsf{E} \in \mathsf{A}_r^i. [\![\mathsf{E}]\!]\sigma = \rho(v_r)\} \\
&= \bigcap_i \gamma_\tau(\langle \mathsf{A}_0^i \ldots \mathsf{A}_{n-1}^i\rangle).
\end{aligned}$$

■

We can now show that the map γ actually gives rise to a Galois connection.

Lemma 6.39. *The abstract domain* Alias *satisfies Requirement 4.2. More precisely, given a type environment $\tau \in \mathcal{T}$, and the function $\gamma_\tau : \textsc{Alias}_\tau \to \mathsf{C}_\tau$, there exists a function $\alpha_\tau : \mathsf{C}_\tau \to \textsc{Alias}_\tau$ such that $\langle \mathsf{C}_\tau, \alpha_\tau, \gamma_\tau, \textsc{Alias}_\tau\rangle$ is a Galois connection.*

Proof. Both C_τ and $\textsc{Alias}_\tau$ are complete lattices. Moreover, Lemma 6.38 shows that γ_τ is co-additive and therefore, according to the results mentioned in Chapter 2 (subsection Galois Connection), there exists the unique map α_τ, determined as:

$$\forall C \in \mathsf{C}_\tau. \alpha(C) = \bigsqcap \{A \in \textsc{Alias}_\tau | C \subseteq \gamma_\tau(A)\},$$

such that $\langle \mathsf{C}_\tau, \alpha_\tau, \gamma_\tau, \textsc{Alias}_\tau\rangle$ is a Galois connection. Therefore, Requirement 4.2 is satisfied and $\textsc{Alias}_\tau$ is actually an abstract domain, in the sense of abstract interpretation. ■

6.4.3 Monotonicity of the Propagation Rules

Another important condition necessary for the proof of existence of the least solution of our analysis is the monotonicity of the propagation rules. These rules represent an abstract semantics of bytecode instructions. We enunciate the following lemma without a proof, since it is straightforward.

Lemma 6.40. *The propagation rules* Rule #1 - Rule #23 *satisfy Requirement 4.3, i.e., they are monotonic with respect to $\sqsubseteq$.*

Proof. This proof is straightforward from the definition of the propagation rules. ■

6.4.4 Sequential Arcs

This subsection is dedicated to Requirement 4.4, which states that in the case of the propagation rules of the sequential arcs, only non-exceptional concrete states belonging to the concretization of a correct approximation of the property of interest before a bytecode instruction is executed, are correctly propagated by the corresponding rule. That is because the sequential arcs simulate only those bytecode instructions which are defined on non-exceptional concrete states, and undefined on the exceptional ones. Let us show that this property actually holds.

Lemma 6.41. *The propagation rules* Rule #1 - Rule #14 *introduced by Definition 6.20 satisfy Requirement 4.4. More precisely, let us consider a sequential arc from a bytecode* ins *and its propagation rule Π. Assume that* ins *has static type information τ at its beginning and τ' immediately after its non-exceptional execution. Then, for every $A \in \text{Alias}_\tau$ we have:*

$$ins(\gamma_\tau(A)) \cap \Xi_{\tau'} \subseteq \gamma_{\tau'}(\Pi(A))$$

(we recall that ins is the semantics of ins*, see Fig. 3.6).*

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contains i local variables $L = \{l_0, \ldots, l_{i-1}\}$ and j stack elements $S = \{s_0, \ldots, s_{j-1}\}$. For ease of representation, we let $\mathsf{dom}(\tau) = \{v_0, \ldots, v_{n-1}\}$, where $n = |\tau|$, $v_r = l_r$ for $0 \leq r < i$ and $v_r = s_{r-i}$ for $i \leq r < n$, like we did in Definition 3.7. Moreover, let $\mathsf{dom}(\tau') = L' \cup S'$, where L' and S' are the local variables and stack elements of $\mathsf{dom}(\tau')$, and let $n' = |\tau'|$.

We choose an arbitrary abstract element $A = \langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle \in \text{Alias}_\tau$, an arbitrary state $\sigma' = \langle \rho', \mu' \rangle \in ins(\gamma_\tau(A)) \cap \Xi_{\tau'}$, and we show that $\sigma' \in \gamma_{\tau'}(\Pi(A))$, i.e., (Definition 6.18) that

$$\text{for each } 0 \leq r < n' \text{ and every } \mathsf{E} \in \mathsf{A}_r, [\![\mathsf{E}]\!]\sigma' = \rho'(v_r). \tag{6.12}$$

Note that, by the choice of σ', there exists $\sigma = \langle \rho, \mu \rangle \in \gamma_\tau(A)$ such that $\sigma' = ins(\sigma)$ and such that, for each $0 \leq r < n$ and every $\mathsf{E} \in \mathsf{A}_r$, $[\![\mathsf{E}]\!]\sigma = \rho(v_r)$.

$\underline{\mathsf{ins} = \mathsf{const}\ x}$. We have $L' = L$, $S' = S \cup \{s_j\}$. Moreover, for every $v \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\underline{\rho'(v) = \rho(v)}$, while $\rho'(s_j) = x$ and $\mu' = \mu$. According to Rule #1 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_n \rangle$, where $\mathsf{A}'_r = \mathsf{A}_r$ for $r \neq n$ and $\mathsf{A}'_n = \{x\}$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $0 \leq r < n$, then $\mathsf{A}'_r = \mathsf{A}_r$, and therefore $\mathsf{E} \in \mathsf{A}_r$. By hypothesis,

  $$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

  In this case, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for each $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. Finally, by Corollary 6.15, we have:

  $$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$

- if $r = n$, then $\mathsf{A}'_n = \{x\}$ and $\mathsf{E} = x$. Therefore, $[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = x = \rho'(v_n)$.

$\underline{\mathsf{ins} = \mathsf{load}\ k\ \mathsf{t}}$. We have $L' = L$, $S' = S \cup \{s_j\}$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\underline{\rho'(v) = \rho(v)}$, while $\rho'(s_j) = \rho(l_k)$. According to Rule #2 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_n \rangle$, where $\mathsf{A}'_r = \mathsf{A}_r \cup \mathsf{A}_r[s_j/l_k]$ for $r \notin \{k, n\}$, $\mathsf{A}'_k = \mathsf{A}_k \cup \{s_j\}$ and $\mathsf{A}_n = \mathsf{A}_k \cup \{l_k\}$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $r \notin \{k, n\}$, then $\mathsf{A}'_r = \mathsf{A}_r \cup \mathsf{A}_r[s_j/l_k]$. If $\mathsf{E} \in \mathsf{A}_r$, then, by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho(v_r) = \rho'(v_r).$$

  In this case, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have:

$$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}]\!]\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho'(v_r).$$

  Otherwise, if $\mathsf{E} \in \mathsf{A}_r[s_j/l_k]$, then there exists $\mathsf{E}_1 \in \mathsf{A}_r$ such that $\mathsf{E} = \mathsf{E}_1[s_j/l_k]$. Note that, by hypothesis,

$$[\![\mathsf{E}_1]\!]\langle\rho, \mu\rangle = \rho(v_r) = \rho'(v_r).$$

  Since $\rho'(s_j) = \rho'(l_k)$, we have, by Lemma 6.16:

$$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}_1[s_j/l_k]]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}_1]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}_1]\!]\langle\rho', \mu\rangle.$$

  On the other hand, $\mathsf{E}_1 \in \mathsf{A}_r$ entails $s_j \notin \mathsf{variables}(\mathsf{E}_1)$, and therefore for every $v \in \mathsf{variables}(\mathsf{E}_1)$, $\rho'(v) = \rho(v)$. By Corollary 6.15 we have:

$$[\![\mathsf{E}_1]\!]\langle\rho', \mu\rangle = [\![\mathsf{E}_1]\!]\langle\rho, \mu\rangle = \rho'(v_r).$$

  Thus, $[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}_1]\!]\langle\rho', \mu\rangle = \rho'(v_r)$.
- if $r = k$, then $v_k = l_k$ and $\mathsf{A}'_k = \mathsf{A}_k \cup \{s_j\}$. If $\mathsf{E} \in \mathsf{A}_k$, then by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho(v_k) = \rho(l_k) = \rho'(s_j) = \rho'(v_k).$$

  Moreover, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have

$$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}]\!]\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho'(v_k).$$

  Otherwise, $\mathsf{E} = s_j$, and we have

$$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = \rho'(s_j) = \rho'(l_k) = \rho'(v_k).$$

- if $r = n$, then $v_n = s_j$ and $\mathsf{A}'_n = \mathsf{A}_k \cup \{l_k\}$. If $\mathsf{E} \in \mathsf{A}_k$, then by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho(v_k) = \rho'(v_n).$$

  Moreover,$s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have:

$$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}]\!]\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho'(v_n).$$

  Otherwise, $\mathsf{E} = l_k$, and we have

$$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = \rho'(l_k) = \rho'(s_j) = \rho'(v_n).$$

$\mathsf{ins} = \mathsf{store}\ k\ \mathsf{t}$. We have $L' = L$, $S' = S \setminus \{s_{j-1}\}$, $\mu' = \mu$, and for every $v \in \mathsf{dom}(\tau') \setminus \overline{\{l_k\}}$, $\rho'(v) = \rho(v)$, while $\rho'(l_k) = \rho(s_{j-1})$. According to Rule #5 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{n-2}\rangle$, where $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}$ for $r \neq k$ and $\mathsf{A}'_k = \{\mathsf{E} \in \mathsf{A}_k \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}$. According to Definition 6.11, for every $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ holds if $l_k, s_{j-1} \notin \mathsf{variables}(\mathsf{E})$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $r \neq k$, then $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid l_k, s_{j-1} \notin \mathsf{variables(E)}\} \subseteq \mathsf{A}_r$. Since $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ we have, by hypothesis,
$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$
Moreover, $l_k, s_{j-1} \notin \mathsf{variables(E)}$, and therefore, for every $v \in \mathsf{variables(E)}$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have:
$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$
- if $r = k$, then $\mathsf{A}'_k = \{\mathsf{E} \in \mathsf{A}_{n-1} \mid l_k, s_{j-1} \notin \mathsf{variables(E)}\} \subseteq \mathsf{A}_{n-1}$. Since $\mathsf{E} \in \mathsf{A}'_{n-1} \subseteq \mathsf{A}_{n-1}$ we have, by hypothesis,
$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_{n-1}) = \rho(s_{j-1}) = \rho'(l_k) = \rho'(v_k).$$
Moreover, $l_k, s_{j-1} \notin \mathsf{variables(E)}$, and therefore, for every $v \in \mathsf{variables(E)}$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have:
$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_k).$$

$\underline{\mathsf{ins} \in \{\mathsf{add}, \mathsf{sub}, \mathsf{mul}, \mathsf{div}, \mathsf{rem}\}}$. We have $L' = L$, $S' = S \setminus \{s_{j-1}\}$, $\mu' = \mu$, and for every $v \in \mathsf{dom}(\tau') \setminus \{s_{j-2}\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_{j-2}) = \rho(s_{j-2}) \oplus \rho(s_{j-1})$, where $\oplus$ is the arithmetic operation corresponding to ins. According to Rule #4 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{n-2}\rangle$, where

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected(E, ins)}\} & \text{if } r \neq |\tau|-2 \\ \{\mathsf{E}_1 \oplus \mathsf{E}_2 \mid \mathsf{E}_1 \in \mathsf{A}_{|\tau|-2} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_1, \mathsf{ins}) \wedge \\ \qquad \mathsf{E}_2 \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_2, \mathsf{ins})\} & \text{if } r = |\tau|-2, \end{cases}$$

According to Definition 6.11, for every expresion $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected(E, ins)}$ holds if $s_{j-2}, s_{j-1} \notin \mathsf{variables(E)}$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $r < n-2$, then $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid s_{j-2}, s_{j-1} \notin \mathsf{variables(E)}\} \subseteq \mathsf{A}_r$. Since $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ we have, by hypothesis,
$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$
Moreover, $s_{j-2}, s_{j-1} \notin \mathsf{variables(E)}$, and therefore, for every $v \in \mathsf{variables(E)}$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have:
$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$
- if $r = n-2$, then $\mathsf{E} = \mathsf{E}_1 \oplus \mathsf{E}_2$, where $\mathsf{E}_1 \in \mathsf{A}_{n-2}$, $\mathsf{E}_2 \in \mathsf{A}_{n-1}$, $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E}_1)$ and $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E}_2)\}$. Since $\mathsf{E}_1 \in \mathsf{A}_{n-2}$ and $\mathsf{E}_2 \in \mathsf{A}_{n-1}$ we have, by hypothesis,
$$[\![\mathsf{E}_1]\!]\langle\rho,\mu\rangle = \rho(v_{n-2}) = \rho(s_{j-2})$$
$$[\![\mathsf{E}_2]\!]\langle\rho,\mu\rangle = \rho(v_{n-1}) = \rho(s_{j-1}).$$
Moreover, for $h \in \{1,2\}$, $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E}_h)$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E}_h)$, $\rho'(v) = \rho(v)$. Hence, by Corollary 6.15,
$$[\![\mathsf{E}_1]\!]^*\langle\rho',\mu\rangle = [\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle = \langle\rho(s_{j-2}), \mu_1\rangle$$

and

$$[\![\mathsf{E}_2]\!]^*\langle\rho',\mu_1\rangle = [\![\mathsf{E}_2]\!]^*\langle\rho,\mu_1\rangle = \langle\rho(s_{j-1}),\mu_2\rangle.$$

Hence,

$$[\![\mathsf{E}]\!]^*\langle\rho',\mu'\rangle = [\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\langle\rho',\mu'\rangle = [\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\langle\rho',\mu\rangle = \langle\rho(s_{j-2}) \oplus \rho(s_{j-1}),\mu_2\rangle,$$

i.e.,

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = \rho(s_{j-2}) \oplus \rho(s_{j-1}) = \rho'(s_{j-2}) = \rho'(v_{n-2}).$$

$\underline{\mathsf{ins} = \mathsf{inc}\ k\ x}$. We have $L' = L$, $S' = S$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau') \setminus \{l_k\}$, $\underline{\rho'(v) = \rho(v)}$, while $\rho'(l_k) = \rho(l_k) + x$. According to Rule #5 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{n-1}\rangle$, where $\mathsf{A}'_r = \{\mathsf{E}[l_k - x/l_k] \mid \mathsf{E} \in \mathsf{A}_r\}$ for $r \neq k$, and $\mathsf{A}'_k = \emptyset$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $r \neq k$, then $\mathsf{A}'_r = \{\mathsf{E}[l_k - x/l_k] \mid \mathsf{E} \in \mathsf{A}_r\}$. If $l_k \notin \mathsf{variables}(\mathsf{E})$, then, $\mathsf{E}[l_k - x/l_k] = \mathsf{E} \in \mathsf{A}_r$ and, by hypothesis,

  $$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

  Moreover, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$ and, by Corollary 6.15, we have:

  $$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

  Otherwise, $l_k \in \mathsf{variables}(\mathsf{E})$, and we have that for any $v \in \mathsf{variables}(\mathsf{E}) \setminus \{l_k\}$, $\rho'(v) = \rho(v)$. Hence, by Lemma 6.14,

  $$[\![\mathsf{E}[l_k - x/l_k]]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle.$$

  By hypothesis,

  $$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r),$$

  and therefore

  $$[\![\mathsf{E}[l_k - x/l_k]]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}[l_k - x/l_k]]\!]\langle\rho',\mu\rangle = \rho'(v_r).$$

- if $r = k$, then $\mathsf{A}'_k = \emptyset$, and therefore $\forall \mathsf{E} \in \mathsf{A}'_k.[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = \rho'(v_k)$ trivially holds.

$\underline{\mathsf{ins} = \mathsf{new}\ \kappa}$. We have $L' = L$, $S' = S \cup \{s_j\}$. For every $v \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\rho'(v) = \rho(v)$, while $\underline{\rho'(s_j) = \ell \in \mathbb{L}}$, where ℓ is a fresh location, i.e., a location not reachable from any other location and which does not reach any other location. Moreover, $\mu' = \mu[\ell \mapsto o]$, where o is a new object of class κ. It is worth noting that, under these circumstances,

$$\forall \mathsf{E} \in \mathbb{E}_{\tau'}.[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle. \tag{6.13}$$

According to Rule #6 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_n\rangle$, where $\mathsf{A}'_r = \mathsf{A}_r$ for $r \neq n$, while $\mathsf{A}'_n = \emptyset$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $0 \leq r < n$, then $\mathsf{A}'_r = \mathsf{A}_r$, and therefore $\mathsf{E} \in \mathsf{A}_r$. By hypothesis,

  $$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

In this case, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have

$$[\![\mathsf{E}]\!]\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho'(v_r).$$

Hence, the latter and (6.13) entail

$$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}]\!]\langle\rho', \mu\rangle = \rho'(v_r).$$

- if $r = n$, then $\mathsf{A}'_n = \emptyset$, and therefore $\forall \mathsf{E} \in \mathsf{A}'_n.[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = \rho'(v_n)$ trivially holds.

$\overline{\mathsf{ins} = \mathsf{getfield}\ f}$. We have $L' = L$, $S' = S$, $\mu' = \mu$, and for every $v \in \mathsf{dom}(\tau') \setminus \{s_{j-1}\}$, $\overline{\rho'(v) = \rho(v)}$, while $\rho'(s_{j-1}) = (\mu\rho(s_{j-1}).\phi)(f)$. According to Rule #7 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{n-1}\rangle$, where for any $r \neq n-1$,

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\},$$

while

$$\mathsf{A}'_{n-1} = \{\mathsf{E}.f \mid \mathsf{E} \in \mathsf{A}_{n-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}) \wedge \neg\mathsf{mightModify}(\mathsf{E}, \{\kappa.f\colon \mathsf{t}\})\}.$$

According to Definition 6.11, for any $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ holds if $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $r \neq n-1$, then $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ and we have, by hypothesis,

  $$[\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho(v_r) = \rho'(v_r).$$

  Moreover, $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have

  $$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}]\!]\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho'(v_r).$$

- if $r = n-1$, then since $\mathsf{E} \in \mathsf{A}'_{n-1}$, there exists $\mathsf{E}_1 \in \mathsf{A}_{n-1}$ such that $\mathsf{E} = \mathsf{E}_1.f$. By hypothesis,

  $$[\![\mathsf{E}_1]\!]\langle\rho, \mu\rangle = \rho(v_{n-1}) = \rho(s_{j-1}).$$

  In addition, $s_{j-1} \notin \mathsf{variables}(\mathsf{E}_1)$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E}_1)$, $\rho'(v) = \rho(v)$. Hence, by Corollary 6.15, we have:

  $$[\![\mathsf{E}_1]\!]^*\langle\rho', \mu\rangle = [\![\mathsf{E}_1]\!]^*\langle\rho, \mu\rangle = \langle\rho(s_{j-1}), \mu_1\rangle.$$

  Hence,

  $$[\![\mathsf{E}_1]\!]^*\langle\rho', \mu'\rangle = [\![\mathsf{E}_1]\!]^*\langle\rho', \mu\rangle = \langle\rho(s_{j-1}), \mu_1\rangle.$$

  Moreover, $\neg\mathsf{mightModify}(\mathsf{E}_1, \{f\})$ guarantees that no evaluation of E_1 might modify the field f. In particular, evaluation of E_1 in $\langle\rho, \mu\rangle$ does not modify f, and therefore its value before E_1's evaluation, $(\mu\rho(s_{j-1}).\phi)(f)$, is equal to its value after E_1's evaluation, $(\mu_1\rho(s_{j-1}).\phi)(f)$. Hence,

  $$\begin{aligned}[\![\mathsf{E}]\!]^*\langle\rho', \mu'\rangle &= [\![\mathsf{E}_1.f]\!]^*\langle\rho', \mu'\rangle \\ &= \langle(\mu_1\rho(s_{j-1}).\phi)(f), \mu_1\rangle \\ &= \langle(\mu\rho(s_{j-1}).\phi)(f), \mu_1\rangle \\ &= \langle\rho'(s_{j-1}), \mu_1\rangle,\end{aligned}$$

  which implies $[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = \rho'(s_{j-1}) = \rho'(v_r)$.

$\underline{\mathsf{ins} = \mathsf{putfield}\ f}$. We have $L' = L$, $S' = S \setminus \{s_{j-2}, s_{j-1}\}$, $\mu' = \mu[(\mu\rho(s_{j-2}).\phi)(f) \mapsto \rho(s_{j-1})]$ and $\rho' = \rho$. According to Rule #8 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{n-3}\rangle$, where for each $0 \leq r < n-2$,

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}.$$

According to Definition 6.11, for every expresion $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ holds if $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})$ and if no evaluation of E might read a field f. Consider an expression $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$, for an arbitrary $0 \leq r < n-2$. By hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho(v_r).$$

Moreover, $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})$, hence for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho(v) = \rho'(v)$ and, by Corollary 6.15:

$$[\![\mathsf{E}]\!]\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho(v_r) = \rho'(v_r).$$

Since no evaluation of E, and in particular its evaluation in $\langle\rho', \mu\rangle$, might read any field f, E's value $[\![\mathsf{E}]\!]\langle\rho', \mu\rangle$ does not depend on a value of any field f in that state. In particular, $[\![\mathsf{E}]\!]\langle\rho', \mu\rangle$ does not depend on the value of the field f of the object memorized in location $\rho(s_{j-1})$, $(\mu\rho(s_{j-1}).\phi)(f)$. Since $\mu' = \mu[(\mu\rho(s_{j-2}).\phi)(f) \mapsto \rho(s_{j-1})]$, i.e., the only difference between memories μ and μ' is exactly the value of the field mentioned above, we conclude that E's values in $\langle\rho', \mu'\rangle$ and in $\langle\rho', \mu\rangle$ are equal, i.e.,

$$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}]\!]\langle\rho', \mu\rangle = \rho'(v_r).$$

$\underline{\mathsf{ins} = \mathsf{arraynew}\ \alpha}$. We have $L' = L$, $S' = S$. For every $v \in \mathsf{dom}(\tau') \setminus \{s_{j-1}\}$, $\rho'(v) = \rho(v)$, $\underline{\text{while } \rho'(s_{j-1}) = \ell \in \mathbb{L}}$, where ℓ is a fresh location, i.e., a location not reachable from any other location and which does not reach any other location. Moreover, $\mu' = \mu[\ell \mapsto a]$, where o is a new aray of array type α. It is worth noting that, under these circumstances,

$$\forall \mathsf{E} \in \mathbb{E}_{\tau'}.[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}]\!]\langle\rho', \mu\rangle. \tag{6.14}$$

According to Rule #9 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_n\rangle$, where $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}$ for $r \neq n-1$, while $\mathsf{A}'_{n-1} = \emptyset$. According to Definition 6.11, for every expresion $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ holds if $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $0 \leq r < n$, then $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid s_{j-1} \notin \mathsf{variables}(\mathsf{E})\} \subseteq \mathsf{A}_r$. Since $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ we have, by hypothesis,
$$[\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho(v_r) = \rho'(v_r).$$
Moreover, $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By (6.14 and Corollary 6.15, we have:
$$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}]\!]\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho'(v_r).$$
- if $r = n-1$, then $\mathsf{A}'_n = \emptyset$, and therefore $\forall \mathsf{E} \in \mathsf{A}'_n.[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = \rho'(v_n)$ trivially holds.

$\underline{\mathsf{ins} = \mathsf{arraylength}\ \alpha}$. We have $L' = L$, $S' = S$, $\mu' = \mu$, and for every $v \in \mathsf{dom}(\tau') \setminus \{s_{j-1}\}$, $\underline{\rho'(v) = \rho(v)}$, while $\rho'(s_{j-1}) = \mu\rho(s_{j-1})$.`length`. According to Rule #10 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{n-1}\rangle$, where for any $r \neq n-1$,

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\},$$

while

$$\mathsf{A}'_{n-1} = \{\mathsf{E}.\texttt{length} \mid \mathsf{E} \in \mathsf{A}_{n-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}.$$

According to Definition 6.11, for any $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ holds if $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $r \neq n-1$, then $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ and we have, by hypothesis,

  $$[\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho(v_r) = \rho'(v_r).$$

  Moreover, $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have

  $$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}]\!]\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho'(v_r).$$

- if $r = n-1$, then since $\mathsf{E} \in \mathsf{A}'_{n-1}$, there exists $\mathsf{E}_1 \in \mathsf{A}_{n-1}$ such that $\mathsf{E} = \mathsf{E}_1.\texttt{length}$. By hypothesis,

  $$[\![\mathsf{E}_1]\!]\langle\rho, \mu\rangle = \rho(v_{n-1}) = \rho(s_{j-1}).$$

  In addition, $s_{j-1} \notin \mathsf{variables}(\mathsf{E}_1)$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E}_1)$, $\rho'(v) = \rho(v)$. Hence, by Corollary 6.15, we have:

  $$[\![\mathsf{E}_1]\!]^*\langle\rho', \mu'\rangle = [\![\mathsf{E}_1]\!]^*\langle\rho', \mu\rangle = [\![\mathsf{E}_1]\!]^*\langle\rho, \mu\rangle = \langle\rho(s_{j-1}), \mu_1\rangle.$$

  Moreover, no evaluation of any expression might modify the length of already existing array, i.e., the length of the array $\mu\rho(s_{j-1})$ is equal before and after E_1's evaluation in $\langle\rho, \mu\rangle$: $\mu\rho(s_{j-1}).\texttt{length} = \mu_1\rho(s_{j-1}).\texttt{length}$. Hence,

  $$\begin{aligned}[\![\mathsf{E}]\!]^*\langle\rho', \mu'\rangle &= [\![\mathsf{E}_1.\texttt{length}]\!]^*\langle\rho', \mu'\rangle \\ &= \langle\mu_1\rho(s_{j-1}).\texttt{length}, \mu_1\rangle \\ &= \langle\mu\rho(s_{j-1}).\texttt{length}, \mu_1\rangle \\ &= \langle\rho'(s_{j-1}), \mu_1\rangle,\end{aligned}$$

  which implies $[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = \rho'(s_{j-1}) = \rho'(v_r)$.

$\overline{\mathsf{ins} = \mathsf{arrayload}\ \alpha}$. We have $L' = L$, $S' = S \setminus \{s_{j-1}\}$, $\mu' = \mu$, and for every $v \in \mathsf{dom}(\tau') \setminus \overline{\{s_{j-2}\}}$, $\overline{\rho'(v) = \rho(v)}$, while $\rho'(s_{j-2}) = (\mu\rho(s_{j-2}).\phi)(\rho(s_{j-1}))$. According to Rule #11 of Definition 6.20, $\Pi(\langle\mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle\mathsf{A}'_0, \ldots, \mathsf{A}'_{n-2}\rangle$, where

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \text{if } r \neq |\tau|-2 \\ \{\mathsf{E}_1[\mathsf{E}_2] \mid \mathsf{E}_1 \in \mathsf{A}_{|\tau|-2} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_1, \mathsf{ins}) \wedge \\ \qquad \mathsf{E}_2 \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_2, \mathsf{ins}) \wedge \\ \qquad [\mathsf{E}_1 \text{ and } \mathsf{E}_2 \text{ do not invoke any method}] & \text{if } r = |\tau|-2. \end{cases}$$

According to Definition 6.11, for every expresion $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ holds if $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $r < n-2$, then $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid s_{j-2}, s_{j-1} \notin \mathsf{variables(E)}\} \subseteq \mathsf{A}_r$. Since $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ we have, by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

  Moreover, $s_{j-2}, s_{j-1} \notin \mathsf{variables(E)}$, and therefore, for every $v \in \mathsf{variables(E)}$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have:

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

- if $r = n-2$, then $\mathsf{E} = \mathsf{E}_1[\mathsf{E}_2]$, where $\mathsf{E}_1 \in \mathsf{A}_{n-2}$, $\mathsf{E}_2 \in \mathsf{A}_{n-1}$, $s_{j-2}, s_{j-1} \notin \mathsf{variables(E}_1\mathsf{)}$, $s_{j-2}, s_{j-1} \notin \mathsf{variables(E}_2\mathsf{)}\}$ and E_1 and E_2 do not invoke any method. Since $\mathsf{E}_1 \in \mathsf{A}_{n-2}$ and $\mathsf{E}_2 \in \mathsf{A}_{n-1}$ we have, by hypothesis,

$$[\![\mathsf{E}_1]\!]\langle\rho,\mu\rangle = \rho(v_{n-2}) = \rho(s_{j-2})$$
$$[\![\mathsf{E}_2]\!]\langle\rho,\mu\rangle = \rho(v_{n-1}) = \rho(s_{j-1}).$$

  Since $s_{j-2}, s_{j-1} \notin \mathsf{variables(E}_1\mathsf{)}$, for every $v \in \mathsf{variables(E}_1\mathsf{)}$, $\rho'(v) = \rho(v)$, and, by Corollary 6.15, we have:

$$[\![\mathsf{E}_1]\!]^*\langle\rho',\mu'\rangle = [\![\mathsf{E}_1]\!]^*\langle\rho',\mu\rangle = [\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle = \langle\rho(s_{j-2}),\mu\rangle. \tag{6.15}$$

  Similarly, since $s_{j-2}, s_{j-1} \notin \mathsf{variables(E}_2\mathsf{)}$, for every $v \in \mathsf{variables(E}_2\mathsf{)}$, $\rho'(v) = \rho(v)$, and, by Corollary 6.15, we have:

$$[\![\mathsf{E}_2]\!]^*\langle\rho',\mu'\rangle = [\![\mathsf{E}_2]\!]^*\langle\rho',\mu\rangle = [\![\mathsf{E}_2]\!]^*\langle\rho,\mu\rangle = \langle\rho(s_{j-1}),\mu\rangle. \tag{6.16}$$

  It is worth noting that since both E_1 and E_2 do not invoke any method, their evaluation in any state $\langle\rho_1,\mu_1\rangle$ do not modify the memory μ_1, hence the resulting memory when E_1 and E_2 are evaluated in $\langle\rho,\mu\rangle$ is μ (Equations 6.15 and 6.16). Hence, Equations 6.15 and 6.16 entail

$$[\![\mathsf{E}]\!]^*\langle\rho',\mu'\rangle = [\![\mathsf{E}_1[\mathsf{E}_2]]\!]^*\langle\rho',\mu'\rangle = \langle(\mu\rho(s_{j-2}).\phi)(\rho(s_{j-1})),\mu\rangle = \langle\rho'(s_{j-2}),\mu\rangle,$$

  i.e.,

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = \rho'(s_{j-2}) = \rho'(v_{n-2}).$$

ins = arraystore t[ ]. We have $L' = L$, $S' = S \setminus \{s_{j-3}, s_{j-2}, s_{j-1}\}$, $\rho' = \rho$ and

$$\mu' = \mu[(\mu\rho(s_{j-3}).\phi)(\rho(s_{j-2})) \mapsto \rho(s_{j-1})].$$

According to Rule #12 of Definition 6.20, $\Pi(\langle\mathsf{A}_0,\ldots,\mathsf{A}_{n-1}\rangle) = \langle\mathsf{A}'_0,\ldots,\mathsf{A}'_{n-4}\rangle$, where for each $0 \le r < n-3$,

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected(E, ins)}\}.$$

According to Definition 6.11, for every expresion $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected(E, ins)}$ holds if $s_{j-3}, s_{j-2}, s_{j-1} \notin \mathsf{variables(E)}$ and if there is no evaluation of E which might read an element of an array of type t′[ ] where t′ $\in$ compatible(t). Consider an expression $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$, for an arbitrary $0 \le r < n-3$. By hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, $s_{j-3}, s_{j-2}, s_{j-1} \notin \mathsf{variables(E)}$, hence for every $v \in \mathsf{variables(E)}$, $\rho(v) = \rho'(v)$ and, by Corollary 6.15:

$$[\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r). \tag{6.17}$$

No evaluation of E might read an element of an array of type $\mathsf{t}'[\,]$, where $\mathsf{t}' \in \mathsf{compatible}(\mathsf{t})$. Therefore, E's value in any state definitely does not depend on any array element whose type is compatible with t and, consequentially, $\mathsf{arraystore}\ \mathsf{t}[\,]$ never affects its value. Hence,

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle. \tag{6.18}$$

Hence, Equations 6.17 and 6.18 entail:

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = \rho'(v_r).$$

$\mathsf{ins} = \mathsf{dup}\,\mathsf{t}$. We have $L' = L$, $S' = S \cup \{s_j\}$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau') \setminus \overline{\{s_j\}}$, $\rho'(v) = \rho(v)$, while $\rho'(s_j) = \rho(s_{j-1})$. According to Rule #13 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_n\rangle$, where $\mathsf{A}'_r = \mathsf{A}_r \cup \mathsf{A}_r[s_j/s_{j-1}]$ for $r < n-1$, $\mathsf{A}'_{n-1} = \mathsf{A}_{n-1} \cup \{s_j\}$ and $\mathsf{A}_n = \mathsf{A}_{n-1} \cup \{s_{j-1}\}$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $0 \le r < n-1$, then $\mathsf{A}'_r = \mathsf{A}_r \cup \mathsf{A}_r[s_j/s_{j-1}]$. If $\mathsf{E} \in \mathsf{A}_r$, then, by hypothesis,

  $$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

  In this case, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have:

  $$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

  Otherwise, if $\mathsf{E} \in \mathsf{A}_r[s_j/s_{j-1}]$, then there exists $\mathsf{E}_1 \in \mathsf{A}_r$ such that $\mathsf{E} = \mathsf{E}_1[s_j/s_{j-1}]$. Note that, by hypothesis,

  $$[\![\mathsf{E}_1]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

  Since $\rho'(s_j) = \rho'(s_{j-1})$, we have, by Lemma 6.16,

  $$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}_1[s_j/s_{j-1}]]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}_1]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}_1]\!]\langle\rho',\mu\rangle.$$

  On the other hand, $\mathsf{E}_1 \in \mathsf{A}_r$ entails $s_j \notin \mathsf{variables}(\mathsf{E}_1)$, and therefore for every $v \in \mathsf{variables}(\mathsf{E}_1)$, $\rho'(v) = \rho(v)$. By Corollary 6.15 we have

  $$[\![\mathsf{E}_1]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}_1]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

  Thus, $[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}_1]\!]\langle\rho',\mu\rangle = \rho'(v_r)$.
- if $r = n-1$, then $v_r = s_{j-1}$ and $\mathsf{A}'_r = \mathsf{A}_{n-1} \cup \{s_j\}$. If $\mathsf{E} \in \mathsf{A}_{n-1}$, then by hypothesis,

  $$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

  Moreover, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have:

  $$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

  Otherwise, $\mathsf{E} = s_j$, and we have $[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = \rho'(s_j) = \rho'(s_{j-1}) = \rho'(v_r)$.

- if $r = n$, then $v_r = s_j$ and $\mathsf{A}'_r = \mathsf{A}_{n-1} \cup \{s_{j-1}\}$. If $\mathsf{E} \in \mathsf{A}_{n-1}$, then by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_{n-1}) = \rho'(v_n).$$

  Moreover, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 6.15, we have

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_n).$$

  Otherwise, $\mathsf{E} = s_{j-1}$, and $[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = \rho'(s_{j-1}) = \rho'(s_j) = \rho'(v_n)$.

$\mathsf{ins} \in \{\mathsf{ifne\ t}, \mathsf{ifeq\ t}, \mathsf{catch}, \mathsf{exception\_is}\ K\}$. We have $L' = L$, $S' = S$ when $\mathsf{ins} \in \overline{\{\mathsf{catch}, \mathsf{exception\_is}\ K\}}$, and $S' = S \setminus \{s_{j-2}, s_{j-1}\}$ otherwise. Moreover, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau')$, $\rho'(v) = \rho(v)$. According to Rule #14 of Definition 6.20, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{n'-1}\rangle$, where $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}$ for each $0 \leq r < n'$. According to Definition 6.11, for any $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ always holds when $\mathsf{ins} \in \{\mathsf{catch}, \mathsf{exception\_is}\ K\}$, while $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ holds if $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})$ when $\mathsf{ins} \in \{\mathsf{ifne\ t}, \mathsf{ifeq\ t}\}$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ for an arbitrary $0 \leq r < n'$. By hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ entails $\mathsf{variables}(\mathsf{E}) \subseteq \mathsf{dom}(\tau')$, and therefore for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. Then, by Corollary 6.15,

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

■

6.4.5 Final Arcs

This subsection is dedicated to Requirement 4.5, which states that in the case of the propagation rules of the final arcs, both exceptional and non-exceptional concrete states belonging to the concretization of a correct approximation of the property of interest before a bytecode instruction is executed, are correctly propagated by the corresponding rule. It means that the propagation rules of the final arcs must soundly approximate the concrete behavior of each final bytecode instruction (return t, return void, throw κ) of a method or a constructor belonging to the program under analysis. Let us show that this property actually holds.

Lemma 6.42. *The propagation rules* Rule #1 - Rule #14 *introduced by Definition 6.23 satisfy Requirement 4.5. More precisely, let us consider a final arc from a bytecode* ins *and its propagation rule Π. 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*, its exceptional execution if* ins *is a* throw κ*). Then, for every $A \in \textsc{Alias}_\tau$ we have:*

$$\mathit{ins}(\gamma_\tau(A)) \subseteq \gamma_{\tau'}(\Pi(A))$$

(we recall that ins is the semantics of ins*, see Fig. 3.6).*

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contains i local variables $L = \{l_0, \ldots, l_{i-1}\}$ and j stack elements $S = \{s_0, \ldots, s_{j-1}\}$. For ease of representation, we let $\mathsf{dom}(\tau) = \{v_0, \ldots, v_{n-1}\}$, where $n = |\tau|$, $v_r = l_r$ for $0 \leq r < i$ and $v_r = s_{r-i}$ for $i \leq r < n$, like we did in Definition 3.7. Moreover, let $\mathsf{dom}(\tau') = L' \cup S'$, where L' and S' are the local and stack variables of $\mathsf{dom}(\tau')$, and let $n' = |\tau'|$.

We choose an arbitrary abstract element $A = \langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle \in \textsc{Alias}_\tau$, an arbitrary state $\sigma' = \langle \rho', \mu' \rangle \in \mathit{ins}(\gamma_\tau(A))$, and we show that $\sigma' \in \gamma_{\tau'}(\mathit{\Pi}(A))$, i.e., (Definition 6.18) that

$$\text{for each } 0 \leq r < n' \text{ and every } \mathsf{E} \in \mathsf{A}_r, [\![\mathsf{E}]\!]\sigma' = [\![v_r]\!]\sigma'.$$

Note that, by the choice of σ', there exists $\sigma = \langle \rho, \mu \rangle \in \gamma_\tau(A)$ such that $\sigma' = \mathit{ins}(\sigma)$ and such that, for each $0 \leq r < n$ and every $\mathsf{E} \in \mathsf{A}_r$, $[\![\mathsf{E}]\!]\sigma = [\![v_r]\!]\sigma = \rho(v_r)$.

$\underline{\mathsf{ins} = \mathsf{return\ void}}$. We have $L' = L$, $S' = \emptyset$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau')$, $\rho'(v) = \rho(v)$. According to Rule #15 of Definition 6.23, $\mathit{\Pi}(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{i-1} \rangle$, where for each $0 \leq r < i$, $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\}$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ for an arbitrary $0 \leq r < i$. By hypothesis,

$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, $\mathsf{noStackElements}(\mathsf{E})$ entails $\mathsf{variables}(\mathsf{E}) \subseteq \mathsf{dom}(\tau')$, and therefore for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. Therefore, by Corollary 6.15, we have:

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$

$\underline{\mathsf{ins} = \mathsf{return\ t}}$. 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 $\rho'(s_0) = \rho(s_{j-1})$. According to Rule #16 of Definition 6.23, $\mathit{\Pi}(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_i \rangle$, where $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\}$ for $r \neq i$ and $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_{n-1} \mid \mathsf{noStackElements}(\mathsf{E})\}$ for $r = i$. Note that for any $\mathsf{E} \in \mathbb{E}_\tau$, $\mathsf{noStackElements}(\mathsf{E})$ entails $\mathsf{variables}(\mathsf{E}) \subseteq \mathsf{dom}(\tau')$, thus for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. Then, by Corollary 6.15,

$$[\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle. \tag{6.19}$$

Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- If $r \neq i$ then $\mathsf{E} \in \mathsf{A}_r$ and, by hypothesis,

  $$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

  By (6.19), we have

  $$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = \rho'(v_r).$$

- If $r = i$ then $\mathsf{E} \in \mathsf{A}_{n-1}$ and, by hypothesis,

  $$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_{n-1}) = \rho(s_{j-1}) = \rho'(s_0) = \rho'(v_i).$$

  By (6.19), we have

  $$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = \rho'(v_i).$$

$\underline{\mathsf{ins} = \mathsf{throw}\ \kappa}$. We have $L' = L$, $S' = \{s_0\}$ and for every $v \in L'$, $\rho'(v) = \rho(v)$. If $\rho(s_{j-1}) \neq \mathtt{null}$, $\rho'(s_0) = \rho(s_{j-1})$ and $\mu' = \mu$. Otherwise, $\rho'(s_0) = \ell$ where ℓ is fresh and $\mu' = \mu[\ell \mapsto npe]$, where *npe* is a new object of class `NullPointerException` containing only fresh locations. It is worth noting that, under these circumstances, for every $\mathsf{E} \in \mathbb{E}_{\tau'}$, $[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}]\!]\langle\rho', \mu\rangle$. According to Rule #17 of Definition 6.23, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_i\rangle$, where $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\}$ for $r \neq i$ and $\mathsf{A}'_r = \varnothing$ for $r = i$. Consider an expression $\mathsf{E} \in \mathsf{A}_r$. For $r \neq i$, the proof is analogous to the proof of the corresponding case for $\mathsf{return\ t}$. If $r = i$, then $\mathsf{A}'_i = \varnothing$, and $\forall \mathsf{E} \in \mathsf{A}'_i . [\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = \rho'(v_i)$ trivially holds. ■

6.4.6 Exceptional Arcs

This subsection is dedicated to Requirements 4.6 and 4.7. The latter states that in the case of the propagation rules of the exceptional arcs, the exceptional concrete states belonging to the concretization of a correct approximation of the property of interest before a bytecode instruction is executed, are correctly propagated by the corresponding rule. It means that the propagation rules of the exceptional arcs simulating the exceptional executions of the bytecode instructions which can throw an exception have to be sound. Let us show that this property actually holds.

Lemma 6.43. *The propagation rules* Rule #18 *and* Rule #20 *introduced by Definition 6.25 satisfy Requirement 4.6. More precisely, let us consider an exceptional arc from a bytecode* ins *distinct from* call *and its propagation rule Π. Assume that* ins *has static type information τ at its beginning and τ' immediately after its exceptional execution. Then, for every $A \in \textsc{Alias}_\tau$ we have:*

$$ins(\gamma_\tau(A)) \cap \underline{\Xi}_{\tau'} \subseteq \gamma_{\tau'}(\Pi(A))$$

(we recall that ins is the semantics of ins*, see Fig. 3.6).*

Proof. The proof is analogous to the proof of Lemma 6.42 when $\mathsf{ins} = \mathsf{throw}\ \kappa$. ■

On the other hand, Requirement 4.7 deals with one particular case of the exceptional arcs: when a method is invoked on a `null` receiver. In that case we require that the exceptional states launched by the method are included in the approximation of the property of interest after the call to that method. Let us show that our propagation rules satisfy this requirement.

Lemma 6.44. *The propagation rule* Rule #19 *introduced by Definition 6.25 satisfies Requirement 4.7. More precisely, consider an exceptional arc from a method invocation $\mathsf{ins}_C = \mathsf{call}\ m_1 \ldots m_n$ and its propagation rule Π, and let π be the number of its actual arguments (*this *included). Then, for each $1 \leq w \leq q$, and every $\sigma = \langle\langle \mathsf{l} \parallel v_{\pi-1} :: \ldots :: v_1 :: \mathtt{null} : \mathsf{s}\rangle, \mu\rangle \in \gamma_\tau(A)$ (σ assigns `null` to the receiver of ins_C right before it is executed), where $A \in \textsc{Alias}_\tau$ is an arbitrary abstract element, we have:*

$$\underline{\langle\langle \mathsf{l} \parallel \ell\rangle, \mu[\ell \mapsto npe]\rangle} \subseteq \gamma_{\tau'}(\Pi(A)),$$

where ℓ is a fresh location, and npe is a new instance of `NullPointerException`.

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contain local variables L and $j \geq \pi$ operand stack elements $S = \{s_0, \ldots, s_{j-\pi}, \ldots, s_{j-1}\}$, where π is the number of parameters of method m_w (including this). Consider an arbitrary abstract element $A \in \textsc{Alias}_\tau$ and a state $\sigma = \langle \rho, \mu \rangle = \langle \langle \mathsf{l} \parallel v_{\pi-1} :: \ldots :: v_1 :: \mathtt{null} : \mathsf{s} \rangle, \mu \rangle \in \gamma_\tau(A)$. Then, by Rule 3 from Fig. 3.7, we have that $\mathsf{dom}(\tau') = L \cup \{s_0\}$, and the resulting state $\sigma' = \langle \rho', \mu' \rangle$ is such that for each $a \in \mathsf{dom}(\tau') \setminus \{s_0\}$, $\rho'(a) = \rho(a)$, $\rho(s_0) = \ell$, where ℓ is a fresh location and $\mu' = \mu[\ell \mapsto npe]$, where npe is a new instance of `NullPointerException`. Hence, $\sigma' = \langle \langle \mathsf{l} \parallel \ell \rangle, \mu[\ell \mapsto npe] \rangle$. Moreover, according to Rule #19,

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\} & \text{if } r \neq i \\ \varnothing & \text{if } r = i. \end{cases}$$

We must prove that $\sigma' \in \gamma_{\tau'}(\Pi(A))$, i.e., (Definition 6.18) that

$$\text{for each } 0 \leq r \leq i \text{ and every } \mathsf{E} \in \mathsf{A}_r, [\![\mathsf{E}]\!]\sigma' = [\![v_r]\!]\sigma'.$$

Let $\mathsf{E}_r \in \mathsf{A}'_r$, for an arbitrary $0 \leq r \leq i$. We distinguish the following cases:

- If $r \neq i$ then $\mathsf{E} \in \mathsf{A}_r$ and, by hypothesis,

  $$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

  Moreover, $\mathsf{noStackElements}(\mathsf{E})$ entails $\mathsf{variables}(\mathsf{E}) \subseteq \mathsf{dom}(\tau')$, and therefore for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. Then, by Corollary 6.15, we have:

  $$[\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$

  The only difference between μ and μ' is object o associated to a fresh location, which is not reachable from any other location. Therefore, μ and μ' behave like they were the same memory. Hence, for every $\mathsf{E} \in \mathbb{E}_{\tau'}$,

  $$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = \rho'(v_r).$$

- If $r = i$ then $\mathsf{A}'_n = \varnothing$, and therefore $\forall \mathsf{E} \in \mathsf{A}'_n . [\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = \rho'(v_n)$ trivially holds.

Therefore, $\langle \langle \mathsf{l} \parallel \ell \rangle, \mu[\ell \mapsto npe] \rangle = \sigma' \in \gamma_{\tau'}(\Pi(A))$. ■

6.4.7 Parameter Passing Arcs

This subsection is dedicated to Requirement 4.8 which states that the propagation rules of the parameter passing arcs are sound. Namely, this rule soundly approximates the behavior of the *makescope* function. Let us show that this property holds.

Lemma 6.45. *The propagation rule* Rule #21 *introduced by Definition 6.27 satisfies Requirement 4.8. More precisely, let us consider a parameter passing arc from a method invocation* $\mathsf{ins}_C = \mathsf{call}\ m_1 \ldots m_n$ *to the first bytecode of a callee* m_w*, for some* $w \in [1..k]$*, and its propagation rule* Π*. Assume that* ins_C *has static type information* τ *at its beginning and that* τ' *is the static type information at the beginning of* m_w*. Then, for every* $A \in \textsc{Alias}_\tau$ *we have:*

$$(\mathit{makescope}\ m_w)(\gamma_\tau(A)) \subseteq \gamma_{\tau'}(\Pi(A))$$

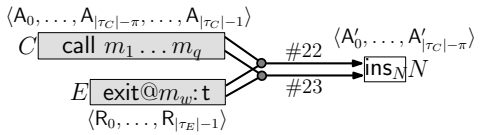


Fig. 6.7. Arcs going into the node corresponding to ins_N.

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contains local variables L and $j \geq \pi$ stack elements $S = \{s_0, \ldots, s_{j-\pi}, \ldots, s_{j-1}\}$, where π is the number of parameters of method m_i (including `this`). Then, $\mathsf{dom}(\tau') = \{l_0, \ldots, l_{\pi-1}\}$. We choose an arbitrary abstract element $A = \langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle \in \textsc{Alias}_\tau$, an arbitrary state $\sigma' = \langle \rho', \mu' \rangle \in (\mathit{makescope}\ m_i)(\gamma_\tau(A))$, and we show that $\sigma' \in \gamma_{\tau'}(\Pi(A))$ i.e., (Definition 6.18) that

$$\text{for each } 0 \leq r < n' \text{ and every } \mathsf{E} \in \mathsf{A}_r, [\![\mathsf{E}]\!]\sigma' = [\![v_r]\!]\sigma'.$$

Note that, by the choice of σ', there exists $\sigma = \langle \rho, \mu \rangle \in \gamma_\tau(A)$ such that $\sigma' = \mathit{ins}(\sigma)$ and such that, for each $0 \leq r < n$ and every $\mathsf{E} \in \mathsf{A}_r$, $[\![\mathsf{E}]\!]\sigma = [\![v_r]\!]\sigma = \rho(v_r)$.

According to Rule #21, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{\pi-1} \rangle$, where for each $0 \leq r < \pi$, $\mathsf{A}'_r = \varnothing$. Then, for each r, $\forall \mathsf{E} \in \mathsf{A}_r = \varnothing'.[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = \rho'(v_r)$ trivially holds. ■

6.4.8 Return and Side-Effects Arcs at Non-Exceptional Ends

In this subsection we show that our definite expression aliasing analysis satisfies Requirements 4.9 and 4.10. The following lemmas deal with the return values and side-effects of the non-exceptional executions of methods. Namely, in the case of a non-void method, the propagation rule of the return value arc enriches the resulting approximation of the definite aliasing information immediately after the call to that method by adding all those aliasing expressions that the returned value might correspond to. On the other hand, that method might modify the initial memory from which the method has been executed. These modifications must be captured by the propagation rules of the side-effects arcs. The approximation of the property of interest after the call to the method is, therefore, determined as the join ($\sqcup$) of the approximations obtained from the propagation rules of the return value and the side-effects arcs, and it is sound, like Lemma 6.46 shows. Lemma 6.47 handles the case of a void method, and therefore only the corresponding side-effects arc is considered there.

Lemma 6.46. *The propagation rules* Rule #22 *and* Rule #23 *introduced by Definitions 6.32 and 6.33 satisfy Requirement 4.9. More precisely, the propagation rules for the return value arcs and side-effects arcs are sound at a* non-void *method return. Namely, let* $w \in [1..n]$ *and consider a return value and a side-effect arc from nodes* $\mathsf{C} = \boxed{\mathsf{call}\ m_1 \ldots m_n}$ *and* $\mathsf{E} = \boxed{\mathsf{exit@}\, m_w}$ *to a node* $\mathsf{Q} = \boxed{\mathsf{ins}_q}$ *and their propagation rules* $\Pi^{\#22}$ *and* $\Pi^{\#23}$*, respectively. We depict this situation in Fig. 6.7. Let* τ_c*,* τ_q *and* τ_e *be the static type information at* C*,* Q *and* E*, respectively, and let* d *be 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* $A \in \textsc{Alias}_{\tau_c}$ *and* $R \in \textsc{Alias}_{\tau_e}$*, we have:*

$$d((\mathit{makescope}\ m_w)(\gamma_{\tau_c}(A)) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}(\Pi^{\#22}(A, R) \sqcup \Pi^{\#23}(A, R)).$$

Proof. In the following we assume: $\mathsf{dom}(\tau_h) = L_h \cup S_h$, where h can be C, E or N; $\mathsf{dom}(\tau_h) = \{v_0, \ldots, v_{|\tau_h|-1}\}$, where $v_r = l_r$ when $0 \le r < |L_h|$ and $v_r = s_{r-|L_h|}$ when $|L_h| \le r < |\tau_a|$; π is the number of parameters of method m, $|\tau_C| - \pi \ge |L_C|$, $|\tau_N| = |\tau_C| - \pi + 1$, $L_N = L_C$ and $S_E = S_N = \{s_0\}$.

Consider two abstract elements at C and E: $A = \langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau_C|-\pi}, \ldots, \mathsf{A}_{|\tau_C|-1} \rangle \in \mathsf{A}_{\tau_C}$ and $R = \langle \mathsf{R}_0, \ldots, \mathsf{R}_{|\tau_E|-1} \rangle \in \mathsf{A}_{\tau_E}$, two concrete states corresponding to these abstract elements $\sigma_C = \langle \rho_C, \mu_C \rangle \in \gamma_{\tau_C}(A)$ and $\sigma_E = \langle \rho_E, \mu_E \rangle \in \gamma_{\tau_E}(R)$ and state $\sigma_N = \langle \rho_N, \mu_N \rangle = d((\mathit{makescope}\ m_w)(\sigma_C)) \cap \Xi_{\tau_N}$. These states have to satisfy the following conditions imposed by Definition 3.19:

1. for every $0 \le r < |\tau_C| - \pi$, $\rho_N(v_r) = \rho_C(v_r)$;
2. $\rho_N(v_{|\tau_C|-\pi}) = \rho_E(v_{|\tau_E|-1})$;
3. $\mu_N = \mu_E$.

Moreover, since $\sigma_C \in \gamma_{\tau_C}(A)$, and $\sigma_E \in \gamma_{\tau_E}(A)$, by Definition 6.18, the following condition holds:

$$\forall 0 \le r < |\tau_h| . \forall \mathsf{E} \in \mathsf{A}_r . [\![\mathsf{E}]\!]\sigma_h = \rho_h(v_r), \tag{6.20}$$

where $h \in \{C, E\}$. Let us show that $\sigma_N \in \gamma_{\tau_N}(A')$, where $A' = \Pi^{\#22}(A, R) \sqcup \Pi^{\#23}(A, R)$, i.e., that Equation 6.20 also holds for $h = N$. By Definitions 6.17 and 4.1, we have $\Pi^{\#22}(A, R) \sqcup \Pi^{\#23}(A, R) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{|\tau_C|-\pi} \rangle$, where A'_r are defined as follows:

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{safeExecution}(\mathsf{E}, \mathsf{ins}_c)\} & \text{if } r < |\tau_C| - \pi \\ \overbrace{\{\mathsf{E} = \mathsf{R}[\mathsf{E}_0, \ldots, \mathsf{E}_{\pi-1}/l_0, \ldots, l_{\pi-1}] \mid \mathsf{R} \in \mathsf{R}_{|\tau_E|-1} \wedge \mathsf{safeReturn}(\mathsf{R}, m_w) \wedge \mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_c)\}}^{X} & \\ \cup \underbrace{\{\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_{\pi-1}) \mid \mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_c)\}}_{Y} & \text{if } r = |\tau_C| - \pi \end{cases}$$

where maps $\mathsf{noParameters}$, $\mathsf{safeExecution}$, $\mathsf{safeAlias}$ and $\mathsf{safeReturn}$ are introduced in Definitions 6.28, 6.29, 6.30 and 6.31 respectively.

Let $\mathsf{E} \in \mathsf{A}_r$, for an arbitrary $0 \le r \le |\tau_C| - \pi$ and let us show that $[\![\mathsf{E}]\!]\sigma_N = \rho_N(v_r)$. We distinguish the following cases:

- if $r \ne |\tau_C| - \pi$, then $\mathsf{E} \in \mathsf{A}_r$ and $\mathsf{safeExecution}(\mathsf{E}, \mathsf{ins}_C)$ hold. It entails:
  1. $\mathsf{E} \in \mathsf{A}_r$, and therefore, by hypothesis (6.20),
  $$[\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle = \rho_C(v_r);$$
  2. $\mathsf{noParameters}(\mathsf{E})$ holds, and therefore $\mathsf{variables}(\mathsf{E}) \subseteq \{v_0, \ldots, v_{|\tau_C|-\pi-1}\} \subseteq \mathsf{dom}(\tau_N)$, which entails
  $$\mathsf{E} \in \mathbb{E}_{\tau_N};$$
  3. $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}_C)$, i.e., execution of ins_C cannot affect evaluations of E, and therefore
  $$[\![\mathsf{E}]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle.$$

  Therefore,
  $$[\![\mathsf{E}]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle = \rho_C(v_r).$$
- if $r = |\tau_C| - \pi$, we distinguish two cases. In the first case, $\mathsf{E} \in X$, and therefore it has the following form: $\mathsf{E} = \mathsf{R}[\mathsf{E}_0, \ldots, \mathsf{E}_{\pi-1}/l_0, \ldots, l_{\pi-1}]$, where $\mathsf{R} \in \mathsf{R}_{|\tau_E|-1}$, $\mathsf{safeReturn}(\mathsf{R}, m_w)$ and $\mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_C)$ hold. According to Definition 6.30, the latter entails:

1. for each $0 \leq k < \pi$, $\mathsf{noParameters}(\mathsf{E}_k)$ holds, thus $\mathsf{variables}(\mathsf{E}_k) \subseteq \{v_0, \ldots, v_{|\tau_C|-\pi-1}\} \subseteq \mathsf{dom}(\tau_N)$, which entails
$$\mathsf{E}_k \in \mathbb{E}_{\tau_N};$$
2. for each $0 \leq k < \pi$, $\mathsf{E}_k \in \mathsf{A}_{|\tau_C|-\pi+k}$, and therefore, by hypothesis (6.20),
$$[\![\mathsf{E}_k]\!]\langle \rho_C, \mu_C \rangle = \rho_C(v_{|\tau_C|-\pi+k});$$
3. for each $0 \leq k < \pi$, $\neg\mathsf{canBeAffected}(\mathsf{E}_k, \mathsf{ins}_C)$ holds, i.e., execution of ins_C cannot affect evaluations of E_k, and therefore
$$[\![\mathsf{E}_k]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}_k]\!]\langle \rho_C, \mu_C \rangle;$$
4.
$$\begin{array}{c}\text{no evaluation of } \mathsf{E} \text{ (hence of } \mathsf{E}_0, \ldots, \mathsf{E}_{\pi-1}, \mathsf{R}) \text{ might modify}\\ \text{any field that might be read by } \mathsf{E} \text{ (hence by } \mathsf{E}_0, \ldots, \mathsf{E}_{\pi-1}, \mathsf{R})\\ \text{or any element of an array of type } \mathsf{t}[\,] \text{ when } \mathsf{E} \text{ (hence } \mathsf{E}_0, \ldots, \mathsf{E}_{\pi-1}, \mathsf{R})\\ \text{might read an element of an array of type } \mathsf{t}'[\,] \text{ where } \mathsf{t}' \in \mathsf{compatible}(\mathsf{t}).\end{array} \tag{6.21}$$

   In other words, any evaluation of one of these expressions does not affect the value of any other of these expressions.

Since $\mathsf{R} \in \mathsf{R}_{|\tau_E|-1}$, by hypothesis (6.20) and by Definition 3.19,
$$[\![R]\!]\langle \rho_E, \mu_E \rangle = \rho_E(v_{|\tau_E|-1}) = \rho_N(v_{|\tau_C|-\pi}).$$
Moreover, according to Definition 6.31, $\mathsf{safeReturn}(\mathsf{R}, m_w)$ entails that $\mathsf{variables}(\mathsf{R}) \subseteq \{l_0, \ldots, l_{\pi-1}\}$, i.e., every variable occurring in R corresponds to a formal parameter of m_w, and for each $l_k \in \mathsf{variables}(\mathsf{R})$, l_k is not modified by m_w, and therefore it has the same value at the beginning and at the end of execution of m_w. Since formal parameter l_k at E corresponds to the actual parameter $v_{|\tau_C|+k}$ at C, we obtain that for every $l_k \in \mathsf{variables}(\mathsf{R})$,
$$\rho_E(l_k) = \rho_C(v_{|\tau_C|+k}).$$
Evaluation of a variable l_k occurring in R in $\langle \rho_E, \mu_E \rangle$ gives
$$[\![l_k]\!]^*\langle \rho_E, \mu_E \rangle = \langle \rho_E(l_k), \mu_E \rangle = \langle \rho_E(l_k), \mu_N \rangle.$$
It is worth noting that the resulting memory does not change. On the other hand, evaluation of an alias expression $\mathsf{E}_k \in \mathsf{A}_{|\tau_C|-\pi+k}$ of $v_{|\tau_C|+k}$ at C in $\langle \rho_N, \mu_N \rangle$ might update the memory:
$$\begin{array}{lll}[\![\mathsf{E}_k]\!]^*\langle \rho_N, \mu_N \rangle &= \langle [\![\mathsf{E}_k]\!]\langle \rho_C, \mu_C \rangle, \mu'_N \rangle & [\text{since } \neg\mathsf{canBeAffected}(\mathsf{E}_k, \mathsf{ins}_C) \text{ holds}]\\ &= \langle \rho_C(s_{p+k}), \mu'_N \rangle & [\text{by hypothesis (6.20)}]\\ &= \langle \rho_E(l_k), \mu'_N \rangle & [\text{since } \mathsf{safeReturn}(\mathsf{R}, m_w) \text{ holds}]\end{array}$$
Therefore, E_k in $\langle \rho_N, \mu_N \rangle$ and l_k in $\langle \rho_E, \mu_E \rangle$ have the same value, but the resulting memory might be different. Nevertheless, (6.21) guarantees that of any $\mathsf{E}_0, \ldots, \mathsf{E}_{\pi-1}, \mathsf{R}$ in $\langle \rho_N, \mu'_N \rangle$ produces the same value which would be obtained if that expression were evaluated in $\langle \rho_N, \mu_N \rangle$, since μ'_N and μ_N might differ only on the fields and array elements that are not read by these expressions. Therefore, the results of these evaluations

are $\rho_E(l_0), \ldots, \rho_E(l_{\pi-1})$ respectively and evaluation of $\mathsf{E} = \mathsf{R}[\mathsf{E}_0, \ldots, \mathsf{E}_{\pi-1}/l_0, \ldots, l_{\pi-1}]$ in $\langle \rho_N, \mu_N \rangle$ gives the same value as the evaluation of R in $\langle \rho_E, \mu_E \rangle$. That value is

$$\rho_E(v_{|\tau_E|-1}) = \rho_N(v_{|\tau_C|-\pi}).$$

In the second case, $\mathsf{E} \in Y$, and has the following form $\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_{\pi-1})$, where $\mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_C)$ holds. The latter entails:

1. for each $0 \le k < \pi$, $\mathsf{noParameters}(\mathsf{E}_k)$ holds, thus $\mathsf{variables}(\mathsf{E}_k) \subseteq \{v_0, \ldots, v_{|\tau_C|-\pi-1}\} \subseteq \mathsf{dom}(\tau_N)$, which entails $\mathsf{E}_k \in \mathbb{E}_{\tau_N}$. Hence, $\mathsf{E} \in \mathbb{E}_{\tau_N}$;
2. for each $0 \le k < \pi$, $\mathsf{E}_k \in \mathsf{A}_{|\tau_C|-\pi+k}$, and therefore, by hypothesis (6.20),
$$[\![\mathsf{E}_k]\!]\langle \rho_C, \mu_C \rangle = \rho_C(v_{|\tau_C|-\pi+k});$$
3. for each $0 \le k < \pi$, $\neg\mathsf{canBeAffected}(\mathsf{E}_k, \mathsf{ins}_C)$, i.e., execution of ins_C cannot affect evaluations of E_k, and therefore
$$[\![\mathsf{E}_k]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}_k]\!]\langle \rho_C, \mu_C \rangle;$$
4. Analogously to (6.21), any evaluation of one of these expressions does not affect the value of any other of these expressions;

Statements 2, 3 and 4 above take us to the following conclusions:

$$\left.\begin{array}{ll} [\![v_{|\tau_C|-\pi}]\!]^*\langle \rho_C, \mu_C \rangle & = \langle \rho_C(v_{|\tau_C|-\pi}), \mu_C \rangle \\ [\![\mathsf{E}_0]\!]^*\langle \rho_C, \mu_C \rangle & = \langle \rho_C(v_{|\tau_C|-\pi}), \mu_C^0 \rangle \\ [\![\mathsf{E}_0]\!]^*\langle \rho_N, \mu_N \rangle & = \langle \rho_C(v_{|\tau_C|-\pi}), \mu_N^0 \rangle \end{array}\right\} \begin{array}{c} \mu_C, \mu_C^0 \text{ and } \mu_N^0 \text{ agree on} \\ \text{all fields and all array elements} \\ \text{that might be read by } \mathsf{E}_0, \ldots, \mathsf{E}_{\pi-1}, m \end{array}$$

$$\ldots$$

$$\left.\begin{array}{ll} [\![v_{|\tau_C|-1}]\!]^*\langle \rho_C, \mu_C \rangle & = \langle \rho_C(v_{|\tau_C|-1}), \mu_C \rangle \\ [\![\mathsf{E}_{\pi-1}]\!]^*\langle \rho_C, \mu_C^{\pi-2} \rangle & = \langle \rho_C(v_{|\tau_C|-1}), \mu_C^{\pi-1} \rangle \\ [\![\mathsf{E}_{\pi-1}]\!]^*\langle \rho_N, \mu_N^{\pi-2} \rangle & = \langle \rho_C(v_{|\tau_C|-1}), \mu_N^{\pi-1} \rangle \end{array}\right\} \begin{array}{c} \mu_C, \mu_C^{\pi-1} \text{ and } \mu_N^{\pi-1} \text{ agree on} \\ \text{all fields and all array elements} \\ \text{that might be read by } \mathsf{E}_0, \ldots, \mathsf{E}_{\pi-1}, m \end{array}$$

These facts imply that the evaluations of $v_{|\tau_C|-\pi}.m(v_{|\tau_C|-\pi+1}, \ldots, v_{|\tau_C|-1})$ in $\langle \rho_C, \mu_C \rangle$ and of $\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_{\pi-1})$ in $\langle \rho_N, \mu_N \rangle$ give the same value: namely, we execute method m of the object $\mu_C(\rho_C(v_{|\tau_C|-\pi})) = \mu_N^{\pi-1}(\rho_C(v_{|\tau_C|-\pi}))$ with parameters $\rho_C(v_{|\tau_C|-\pi+1}), \ldots, \rho_C(v_{|\tau_C|-1})$ on memories μ_C and $\mu_N^{\pi-1}$ which agree on all the fields and all the array elements that might be read by m. Therefore, they will return the same value, and that value is, by Definition 3.19, memorized in $\rho_E(v_{|\tau_E|-1}) = \rho_N(v_{|\tau_C|-\pi})$. Hence,

$$[\![\mathsf{E}]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle = \rho_N(v_{|\tau_C|-\pi}).$$

■

Lemma 6.47. *The propagation rule* Rule #23 *introduced by Definition 6.33 satisfies Requirement 4.10. More precisely, the propagation rule for the side-effects arcs is sound at a* void *method return. Namely, let* $w \in [1..n]$ *and consider a side-effect arc from nodes* $\mathsf{C} = \boxed{\mathsf{call}\ m_1 \ldots m_n}$ *and* $\mathsf{E} = \boxed{\mathsf{exit@}\, m_w}$ *to a node* $\mathsf{Q} = \boxed{\mathsf{ins}_q}$ *and its propagation rule* $\Pi^{\#23}$*. We depict this situation in Fig. 6.7, where the return value arc with the propagation rule* #22 *is omitted. Let* τ_c, τ_q *and* τ_e *be the static type information at* C, Q *and* E*, respectively, and let* d *be 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* $A \in \text{Alias}_{\tau_c}$ *and* $R \in \text{Alias}_{\tau_e}$*, we have:*

$$d((\mathit{makescope}\ m_w)(\gamma_{\tau_c}(A)) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}(\Pi^{\#23}(A, R)).$$

The proof of this lemma is analogous to the case $r \ne |\tau_C| - \pi$ of the proof of Lemma 6.46.

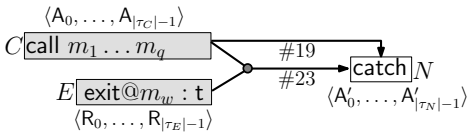


Fig. 6.8. Arcs going into the node corresponding to catch.

6.4.9 Side-Effects and Exceptional Arcs at Exceptional Ends

In this section we show that our definite expression aliasing analysis satisfies also Requirement 4.11. The following lemma deals with the exceptional executions of the methods. Namely, the approximation of the definite aliasing information at the catch which captures the exceptional states of the method we are interested in, has to be affected by all possible modifications of the initial memory due to the side-effects of the method. This is the task of the propagation rules of the side-effects arcs. On the other hand, the final approximation of the definite expression aliasing property at the point of interest (catch) has to be affected by the exceptions launched by the method when it is invoked on a `null` object too. Like in the previous case, the approximation of the definite expression aliasing information is determined as the join ($\sqcup$) of the two approximations mentioned above, and Lemma 6.48 shows that it is correct.

Lemma 6.48. *The propagation rules* Rule #19 *and* Rule #23 *introduced by Definitions 6.25 and 6.33 satisfy Requirement 4.11. More precisely, the propagation rules for the exceptional arcs of the* call *and side-effects arcs are sound when a method throws an exception. Namely, given nodes* Q = catch, C = call $m_1 \dots m_n$ *and* E = exception@m_w, *for a suitable* $w \in [1..n]$, *consider an exceptional arc from* C *to* Q *and a side-effect arc from* C *and* E *to* Q, *with their propagation rules* $\Pi^{\#19}$ *and* $\Pi^{\#23}$, *respectively. We depict this situation in Fig. 6.8. Let* τ_c, τ_q *and* τ_e *be the static type information at* C, Q *and* E, *respectively, and let* d *be 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* $A \in \textsc{Alias}_{\tau_c}$ *and* $R \in \textsc{Alias}_{\tau_e}$, *we have:*

$$d((\textit{makescope } m_w)(\gamma_{\tau_c}(A)) \cap \underline{\Xi}_{\tau_q} \subseteq \gamma_{\tau_q}(\Pi^{\#19}(A) \sqcup \Pi^{\#23}(A, R)).$$

Proof. In the following we assume: $\mathsf{dom}(\tau_h) = L_h \cup S_h$, where h can be C, E or N; $\mathsf{dom}(\tau_h) = \{v_0, \dots, v_{|\tau_h|-1}\}$, where $v_r = l_r$ when $0 \le r < |L_h|$ and $v_r = s_{r-|L_h|}$ when $|L_h| \le r < |\tau_a|$; π is the number of parameters of method m, $|\tau_C| - \pi \ge |L_C|$, $|\tau_N| = |\tau_C| - \pi + 1$, $L_N = L_C$ and $S_E = S_N = \{s_0\}$.

Consider two abstract elements at C and E: $A = \langle \mathsf{A}_0, \dots, \mathsf{A}_{|\tau_C|-\pi}, \dots, \mathsf{A}_{|\tau_C|-1} \rangle \in \mathsf{A}_{\tau_C}$ and $R = \langle \mathsf{R}_0, \dots, \mathsf{R}_{|\tau_E|-1} \rangle \in \mathsf{A}_{\tau_E}$, two concrete states corresponding to these abstract elements $\sigma_C = \langle \rho_C, \mu_C \rangle \in \gamma_{\tau_C}(A)$ and $\sigma_E = \langle \rho_E, \mu_E \rangle \in \gamma_{\tau_E}(R)$ and a state $\sigma_N = \langle \rho_N, \mu_N \rangle = d((\textit{makescope } m_w)(\sigma_C)) \cap \underline{\Xi}_{\tau_N}$. These states have to satisfy the following conditions imposed by Definition 3.19:

1. for every $0 \le r < |\tau_C| - \pi$, $\rho_N(v_r) = \rho_C(v_r)$;
2. $\rho_N(v_{|\tau_C|-\pi}) = \rho_E(v_{|\tau_E|-1})$;
3. $\mu_N = \mu_E$.

Moreover, since $\sigma_C \in \gamma_{\tau_C}(A)$, and $\sigma_E \in \gamma_{\tau_E}(A)$, by Definition 6.18, the following condition holds:

$$\forall 0 \leq r < |\tau_h| . \forall \mathsf{E} \in \mathsf{A}_r . [\![\mathsf{E}]\!]\sigma_h = \rho_h(v_r), \tag{6.22}$$

where $h \in \{C, E\}$. Let us show that $\sigma_N \in \gamma_{\tau_N}(A')$, where $A' = \Pi^{\#19}(A) \sqcup \Pi^{\#23}(A, R)$, i.e., that Equation 6.22 also holds for $h = N$. By Definitions 6.17 and 4.1, we have $\Pi^{\#19}(A) \sqcup \Pi^{\#23}(A, R) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{|\tau_N|-\pi} \rangle$, where A'_r are defined as follows:

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E}) \wedge \neg \mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}_C)\} & \text{if } r < |\tau_C| - \pi \\ \varnothing & \text{if } r = |\tau_C| - \pi \end{cases}$$

where $\mathsf{noStackElements}(\mathsf{E})$ is true if and only if $\mathsf{variables}(\mathsf{E}) \cap S_C = \varnothing$, i.e., if E contains no stack elements.

Let $\mathsf{E} \in \mathsf{A}_r$, for an arbitrary $0 \leq r \leq |\tau_C| - \pi$ and let us show that $[\![\mathsf{E}]\!]\sigma_N = \rho_N(v_r)$. We distinguish the following cases:

- if $r < |\tau_C| - \pi$, then E satisfies the following conditions:
  1. $\mathsf{noStackElements}(\mathsf{E})$ holds, and therefore $\mathsf{variables}(\mathsf{E}) \subseteq L_C = L_N \subseteq \mathsf{dom}(\tau_N)$, which entails $\mathsf{E} \in \mathbb{E}_{\tau_N}$;
  2. $\mathsf{E} \in \mathsf{A}_r$, and therefore, by hypothesis (6.22),
  $$[\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle = \rho_C(v_r);$$
  3. $\neg \mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}_C)$, i.e., execution of ins_C cannot affect evaluations of E, and therefore
  $$[\![\mathsf{E}]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle.$$

  Therefore,
  $$[\![\mathsf{E}]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle = \rho_C(v_r).$$
- if $r = |\tau_C| - \pi$, then $\mathsf{A}'_r = \varnothing$ and therefore $\forall \mathsf{E} \in \mathsf{A}'_N . [\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = \rho'(v_r)$ trivially holds.

■

6.4.10 Conclusion

In this section we have shown that all the requirements provided in Chapter 4 are satisfied by the abstract domain Alias introduced in Section 6.3.1 and by the propagation rules introduced in Section 6.3.2. This fact allows us to assert the following two results.

Theorem 6.49. *There exists the least solution to the aliasing analysis introduced in this chapter.*

Proof. This proof directly follows from the results obtained in Section 4.5, where instead of a generic abstract domain

$$\mathsf{A}_{\tau_k} = \langle \mathcal{A}_{\tau_k}, \sqsubseteq, \sqcup, \sqcap, \top_{\tau_k}, \bot_{\tau_k} \rangle,$$

for a type environment τ_k corresponding to a node k, we use the abstract domain Alias_{τ_k} defined in this chapter. Theorem 4.11 shows that when Requirements 4.1 and 4.3 are satisfied, then there exists the least solution to the system of constraints constructed in Section 4.5, representing the actual static analysis of interest. Lemmas 6.37 and 6.40 show that Requirements 4.1 and 4.3 are satisfied by the instantiation of the framework concerning the definite expression aliasing analysis, hence hypotheses of Theorem 4.11 are satisfied and we can use its results. ■

Node n	Solution of aliasing approximation
a	$\langle \emptyset, \emptyset, \{v_1.\mathsf{getFirst}(), v_3\}, \{v_1.\mathsf{getFirst}(), v_2\}, \{15\}\rangle$
1	$\langle \emptyset, \emptyset\rangle$
2	$\langle \{v_2\}, \emptyset, \{v_0\}\rangle$
3	$\langle \emptyset, \emptyset, \{v_0.\mathtt{min}\}\rangle$
4	$\langle \emptyset, \{v_3\}, \{v_0.\mathtt{min}\}\{v_1\}\rangle$
5	$\langle \emptyset, \emptyset, \{v_0.\mathtt{min} + v_1\}\rangle$
6	$\langle \emptyset, \emptyset, \{v_0.\mathtt{min} + v_1\}, \{60\}\rangle$
7	$\langle \emptyset, \emptyset, \{(v_0.\mathtt{min} + v_1)\%60\}\rangle$
8	$\langle \emptyset, \emptyset, \{(v_0.\mathtt{min} + v_1)\%60\}\rangle$
9	
10	$\langle \emptyset, \emptyset, \emptyset\rangle$
11	
c	$\langle \emptyset, \emptyset, \{v_1.\mathtt{getFirst}()\}, \emptyset\rangle$
b	$\langle \emptyset, \emptyset, \{v_1.\mathtt{getFirst}()\}, \left\{ \begin{array}{c} (v_1.\mathtt{getFirst}().\mathtt{min}{+}15)\%60, (v_2.\mathtt{min}{+}15)\%60, \\ v_1.\mathtt{getFirst}().\mathtt{delayMinBy}(15), v_2.\mathtt{delayMinBy}(15) \end{array} \right\}\rangle$

Fig. 6.9. The solution of the constraint system from Fig. 6.6

Example 6.50. In Figure 6.9 we give the least solution to the system of constraints introduced in Example 6.36 (Fig. 6.6) and concerning the ACG from Fig. 6.5. □

Finally, we can state that our definite expression aliasing analysis is sound, i.e., at each program point the set of aliasing expressions obtained by our analysis for each variable represents an approximation of the actual aliasing expression information available at that point for that variable.

Theorem 6.51. *Let* $\langle b_{first(\mathtt{main})} \parallel \xi\rangle \Rightarrow^* \langle \boxed{\begin{array}{c}\mathsf{ins}\\ rest\end{array}}^{\rightarrow b_1}_{\rightarrow b_m} \parallel \sigma\rangle :: a$ *be the execution of our operational semantics, from the block* $b_{first(\mathtt{main})}$ *starting with the first bytecode instruction of method* `main`*,* ins_0*, and an initial state* $\xi \in \Sigma_{\tau_0}$ *(containing no aliasing information), to a bytecode instruction* ins *and assume that this execution leads to a state* $\sigma \in \Sigma_\tau$*, where* τ_0 *and* τ *are the static type information at* ins_0 *and* ins*, respectively. Moreover, let* $A_0 = \top_{\tau_0} \in \textsc{Alias}_{\tau_0}$*, and let* $A \in \textsc{Alias}_\tau$ *be the aliasing approximation at* ins*, as computed by our definite expression aliasing analysis starting from* A_0*. Then,* $\sigma \in \gamma_\tau(A)$ *holds.*

Proof. This proof directly follows from Theorem 4.14 and the fact that the requirements provided in Chapter 4 are satisfied, which has been shown in this section (Lemmas 6.37-6.48). ■

Therefore, the definte expression aliasing analysis instantiated in the parameterized general framework for constraint-based static analyses of Java bytecode programs is sound and has the least solution.

6.5 Implementation and Experimental Evaluation

We have implemented our definite expression aliasing analysis inside the Julia analyzer for Java bytecode (`http://www.juliasoft.com`) and we have analyzed some real-life

benchmark programs. These benchmarks are reported in Figure 6.10. Some benchmarks are Android applications, that we analyze after being exported in Java bytecode format from the Eclipse IDE used for development. Hence we do not currently analyze their Dalvik bytecode. The benchmarks are analyzed together with most of the libraries that they use. In particular, in table 6.10 we report the libraries included in the analysis of each benchmark. Of course, the standard Java library (`java.*`) and the Android library (`android.*`, for Android benchmarks only) are always included and we do not report it in the figure. The fact that a library is included does not mean that *all* its code is analyzed: only the portion that is actually used by the benchmark is analyzed, and this is extracted through a traditional class analysis for object-oriented code [67]. The Android benchmarks are Mileage, OpenSudoku, Solitaire and TiltMazes[1]; ChimeTimer, Dazzle, OnWatch and Tricorder[2]; TxWthr[3]; VoiceRecognition, CubeLiveWallpaper, AccelerometerPlayActivity, SkeletonApp, AbdTest, Snake, BackupRestore, SoftKeyboard, MultiResolution, LunarLander, TestAppv2, TicTacToe, Spinner, TippyTipper, JetBoy, SampleSyncAdapter, NotePad, HoneycombGallery, Real3D, GestureBuilder, BluetoothChat, SearchableDictionary, ContactManager, Home and Wiktionary, that are all sample programs from the Android 3.1 distribution by Google[4]. The Java programs are JFlex, a lexical analyzers generator[5]; Plume, a library by Michael D. Ernst[6]; Nti, a non-termination analyzer by Étienne Payet[7]; Lisimplex, a numerical simplex implementation by Ricardo Gobbo[8]; avrora, an AVR simulation and analysis framework[9]; luindex, an indexer of documents, h2, a database benchmark, and sunflow, a ray tracer, from the DaCapo benchmark suite[10]; hadoop-common, a software for distributed computing[11]; and our julia analyzer itself[12].

Experiments have been performed on a Linux quad-core Intel Xeon machine running at 3.10GHz, with 12 gigabytes of RAM.

6.5.1 Results w.r.t. the expression aliasing analysis

By only considering the analysis introduced in this article, Fig. 6.10 shows that it is quite fast and scales to large software with a cost in time that has never exploded in our experiments. All analyses could be completed with less than one gigabyte of RAM. In the same figure, the last column on the right reports the average size of the set of expression aliases for each variable. We do not consider the tautological and trivial alias of a variable with itself. That column shows that that size is small, which possibly accounts for the lack of computational explosion during the analysis. This is important, since a formal study of the worst-case complexity of our analysis leads to an exponential cost, in theory, as shown below.

[1] `http://f-droid.org/repository/browse/`
[2] `http://moonblink.googlecode.com/svn/trunk/`
[3] `http://typoweather.googlecode.com/svn/trunk/`
[4] `http://developer.android.com/tools/samples/index.html`
[5] `http://jflex.de`
[6] `http://code.google.com/p/plume-lib`
[7] `http://personnel.univ-reunion.fr/epayet/Research/NTI/NTI.html`
[8] `http://sourceforge.net/projects/lisimplex`
[9] `http://compilers.cs.ucla.edu/avrora/`
[10] `http://www.dacapobench.org`
[11] `http://hadoop.apache.org`
[12] `http://www.juliasoft.com`

ID	NAME	LIBRARIES	LINES	METHODS	RUNTIME	SIZE
1	nti		13915	1653	0.51	0.17
2	lisimplex		25564	2729	1.28	0.15
3	avrora		38165	5006	3.29	0.12
4	JFlex		41365	4286	1.96	0.28
5	plume		44028	4646	2.63	0.12
6	VoiceRecognition		44974	5094	2.40	0.03
7	CubeLiveWallpaper		45891	5197	2.22	0.27
8	AccelerometerPlayActivity		47913	5394	2.34	0.19
9	SkeletonApp		57399	6371	3.49	0.32
10	AbdTest		58020	6402	6.14	0.14
11	Snake		58606	6473	3.11	0.26
12	BackupRestore		58706	6471	3.23	0.24
13	SoftKeyboard		58819	6535	4.31	0.35
14	MultiResolution		58917	6542	3.03	0.86
15	LunarLander		59122	6519	3.15	0.2
16	TestAppv2		59889	6587	3.14	0.35
17	TicTacToe		59943	6657	3.13	0.37
18	Spinner		61912	6759	3.32	0.42
19	luindex	lucene-core, lucene-demos	62050	6409	3.18	0.18
20	Solitaire		63507	6988	3.46	0.32
21	TippyTipper		65310	7322	3.36	0.89
22	JetBoy		65874	7189	3.73	0.22
23	SampleSyncAdapter		66646	7348	3.47	0.08
24	NotePad		67066	7372	3.56	0.28
25	sunflow	janino	72061	9130	6.18	0.18
26	HoneycombGallery		72352	7879	5.31	0.32
27	Real3D		75001	8179	4.37	0.15
28	TxWthr		75434	8232	43.45	0.35
29	Dazzle	hermitandroid	78344	8681	40.34	0.28
30	GestureBuilder		85213	9093	5.06	0.51
31	BluetoothChat		85290	9119	5.07	0.45
32	SearchableDictionary		88034	9392	5.24	0.24
33	ContactManager		88110	9465	5.35	0.48
34	Home		88256	9489	5.22	0.3
35	TiltMazes		90419	9641	5.35	0.39
36	ChimeTimer	hermitandroid	90465	9743	6.09	0.26
37	Mileage		104647	11188	6.07	0.16
38	Tricorder	hermitandroid	105475	11140	6.57	0.34
39	julia	bcel	106117	12495	15.42	0.29
40	Wiktionary		109140	11762	7.54	0.27
41	OnWatch	hermitandroid	114391	11928	7.01	0.26
42	hadoop-common	org.w3c.*, javax.security.* guava, protobuf-java, jets3t	118706	14812	5.91	0.21
43	OpenSudoku		120164	13002	6.59	0.22
44	h2	junit3, derbyTesting, tpcc	183398	17276	14.99	0.17

Fig. 6.10. The benchmarks that we have analyzed and the cost and precision of their expression aliasing analysis. For each benchmark we report the name, the number of non-comment non-blank source code lines that get analyzed, the number of methods that get analyzed, the time (in seconds) required for our expression aliasing analysis and the average size of the approximation of each variable (number of alias expressions per variables). The latter does not include the tautological aliasing of a variable with itself, which is trivial and irrelevant.

6.5.2 Theoretical Computational Complexity

A worst-case complexity for our definite aliasing analysis can be estimated as follows. The number N of alias expressions (Definition 6.1) over a maximal number v of variables in scope, the number n of constants (those used in the program text), the number f of field names and the number m of method names are finite as soon as we fix a maximal height k for the alias expressions themselves. More precisely, N is polynomial in v, n, f and m and exponential in k. Note that we consider here the maximal number v of variables in scope at each given program point, which is usually small, and not the total number of program variables, that can be very large instead. The approximation of each program variable is a set of alias expressions; hence there are 2^N alternatives for each variables. There are at most v such sets at each program point, one for each variable in scope there. Hence, the possible approximations at a given program point are $v \cdot 2^N$. If l is the length of the program, that is, the number of its program points, we end up with a set of possible approximations bounded from above by $l \cdot v \cdot 2^N$. That is, our expression aliasing analysis might require up to $l \cdot v \cdot 2^N$ iterations until stabilization and its computational cost is, hence, exponential in v, n, f and m and double exponential in k. In our implementation, we have fixed $k = 4$.

It is interesting to observe that this exponential blow-up does not arise in practice. First of all, many alias expressions cannot be generated because they would not type-check. Moreover, on average, each variable is approximated by very small alias sets (Fig. 6.10): in most cases, those sets are empty or singletons. For this reason, the theoretically bad computational result that we have just shown is not reflected in the actual cost of the analysis.

6.5.3 Implementation Optimizations

An abstract Java class `Alias` is used to represent the alias expressions of Definition 6.1. Constants and variables are concrete and non-recursive subclasses of `Alias`. Other alias expressions are concrete and recursive subclasses of `Alias`; for instance, the alias expression $\mathsf{E}.f$ is represented by a subclass `FieldOf` of `Alias` that refers to the field f and to another alias expression, that is E. The reduction of the memory footprint of this representation for alias expressions is possible and important, since identical or at least similar alias expressions are often used at different program points. Namely, our representation of alias expressions is *interned*, that is, we never generate two Java objects that stand for the same alias expression. This allows a maximal sharing of data structures and reduces the memory cost of our representation. We achieve this internment through a map that binds each alias expression e to the unique representative of all alias expressions equal to e. Since Julia is a parallel analyzer, race conditions must be avoided in the access to that map. To that purpose, we use a `java.util.concurrent.ConcurrentHashMap` and its handy `putIfAbsent()` method for checking the presence and putting new alias expressions in the table, atomically. Beyond the reduction of the memory cost, internment has the advantage that equality of alias expressions can be tested by faster == tests rather than `equals()` calls.

Sets of alias expressions are hence sets of unique objects. There are many such sets during the analysis, for each variable in scope at each given program point. Also in this case, a compact representation is needed. We have achieved this by using bitsets, represented through arrays of `long`s (each `long` contains 64 bits). A fixed enumeration of

the alias expressions relates a given bit position to the expression that it represents. The enumeration is built on demand as soon as new alias expressions are created by the analysis. For each bit set to 1, the alias expression in that enumeration position is assumed to belong to the bitset.

6.5.4 Benefits for other analyses

Section 6.5.1 has shown that the aliasing expression analysis can be computed in a few seconds also for large applications. It remains to show that its results are useful for other, subsequent analyses, that exploit the availability of definite aliasing information. To that purpose, we have used our analysis to support Julia's nullness and termination analyses. In particular, we use our analysis at the then branch of each comparison `if (v!=null)` to infer that the definite aliases of `v` are non-`null` there, and at each assignment `w.f=exp` to infer that expressions `E.f` are non-`null` when `exp` is non-`null` and when `E` is a definite alias of `w` whose evaluation does not read nor write `f`. Moreover, we use it to infer symbolic upper or lower bounds of variables whenever we have a comparison such as `x<y`: all definite alias expressions of `y` (respectively `x`) are upper (respectively lower) bounds for `x` (respectively `y`). This is important for the termination analysis of Julia.

Note that our nullness and termination analyses are sound, that is, there are no false negatives (as long as reflection is not used or only used in a limited way: for instance, inflation of XML views in Android is supported in Julia [70]); but there are false positives of course. We have identified actual bugs in the benchmarks, among the places where Julia signals a possible warning. However, we cannot check by hand hundreds of warnings, on third-party code. Nevertheless, the nullness analysis of Julia is currently the most precise available on the market [85] and scales to very large software, as our experiments here show. The termination analysis of Julia scales almost in the same way, as shown below, and we have never seen any other report of a termination analysis that scales to that size of programs. Nevertheless, this article is not about nullness nor termination nor their precision. We use those analyses only to support the thesis that definite aliasing analysis is useful to support other analyses.

Figure 6.11 reports the times for nullness and termination analysis. These are total times, that is, they include everything: from the parsing of the `.jar` files, to ours and other supporting analyses, to the presentation of the warnings to the user. In that figure, we have copied the times for the expression aliasing analysis alone, to highlight the fact that they are only a small fraction of the total times for nullness and termination analysis. When the expression aliasing analysis is turned off, times are in general smaller, also because there is fewer information to exploit for nullness and termination proofs. For instance, when that information is missing, it is sometimes the case that a symbolic upper bound for a loop variable is missed, which results in the immediate failure of the termination proof but in coarser results. The termination proof for h2 went into out of memory, so we do not report times for it in the figure.

Figures 6.12 and 6.13 report the precision of the same nullness and termination analyses. They show how fewer warnings are issued by Julia after those analyses, if our aliasing expression analysis is turned on. Figure 6.12 shows that the number of warnings for nullness analysis is in general halved; sometimes, there are no more warnings, thanks to the support of the expression aliasing analysis (as for benchmarks 6, 14 and 17); only in three cases there are no benefits (benchmarks 9, 11 and 16). The situation is similar for termination analysis 6.13, but the gain in precision is less evident here.

ID	EXP	NULLNESS		TERMINATION	
	ALIASING	WITH	WITHOUT	WITH	WITHOUT
1	0.51	12.64	11.73	19.66	13.96
2	1.28	27.81	24.81	17.13	15.37
3	3.29	78.50	62.93	115.21	93.90
4	1.96	54.83	48.75	79.61	63.34
5	2.63	74.98	78.15	86.21	73.63
6	2.40	50.88	45.61	27.60	24.92
7	2.22	51.21	47.25	29.76	26.86
8	2.34	57.95	54.69	31.61	31.82
9	3.49	73.89	68.12	47.01	40.88
10	6.14	138.66	119.75	52.81	76.14
11	3.11	87.92	86.48	51.80	42.70
12	3.23	77.11	72.82	49.51	45.76
13	4.31	71.08	66.16	42.05	37.72
14	3.03	78.25	71.93	47.67	43.65
15	3.15	94.59	83.16	51.00	47.77
16	3.14	75.35	70.58	49.91	45.71
17	3.13	80.64	72.09	50.52	46.76
18	3.32	91.22	85.76	49.58	48.84
19	3.18	125.16	112.97	354.19	248.65
20	3.46	107.98	100.43	96.55	89.63
21	3.36	96.42	91.39	54.63	53.76
22	3.73	99.40	92.58	56.91	51.60
23	3.47	98.28	92.58	62.96	55.58
24	3.56	98.20	92.88	56.94	51.24
25	6.18	296.79	269.18	816.06	904.79
26	5.31	109.28	105.20	61.11	59.40
27	4.37	121.69	113.95	72.17	61.36
28	43.45	166.83	109.68	95.87	65.18
29	40.34	188.86	130.80	105.50	69.22
30	5.06	155.79	138.68	80.66	76.33
31	5.07	183.82	161.70	82.77	74.16
32	5.24	167.97	146.02	85.87	76.55
33	5.35	182.31	178.20	86.33	78.64
34	5.22	191.19	170.41	87.26	75.08
35	5.35	167.06	163.04	88.13	79.55
36	6.09	188.15	173.07	88.42	84.29
37	6.07	254.84	257.70	136.21	121.86
38	6.57	247.31	233.59	132.87	114.89
39	15.42	617.00	600.70	1090.51	1630.15
40	7.54	258.14	242.33	126.09	104.66
41	7.01	307.38	277.47	151.55	136.46
42	5.91	623.97	638.86	638.47	561.45
43	6.59	341.38	322.75	164.98	148.97
44	14.99	1017.42	1007.89	N.A.	N.A.

Fig. 6.11. Total time (in seconds) for the nullness and termination analysis of our benchmarks, each computed in two versions: with our expression aliasing analysis and without our expression aliasing analysis. We also report the cost of just the expression aliasing analysis, for comparison. The cost of nullness and termination include everything: from the parsing of the `jar` files to the presentation of the warnings to the user. Hence they also include the times of the expression aliasing analysis. The termination analysis of benchmark 44 (`h2`) could not be completed because of an out of memory during the termination proof (while our expression aliasing analysis could be completed in 14.99 seconds and less than one gigabyte of RAM).

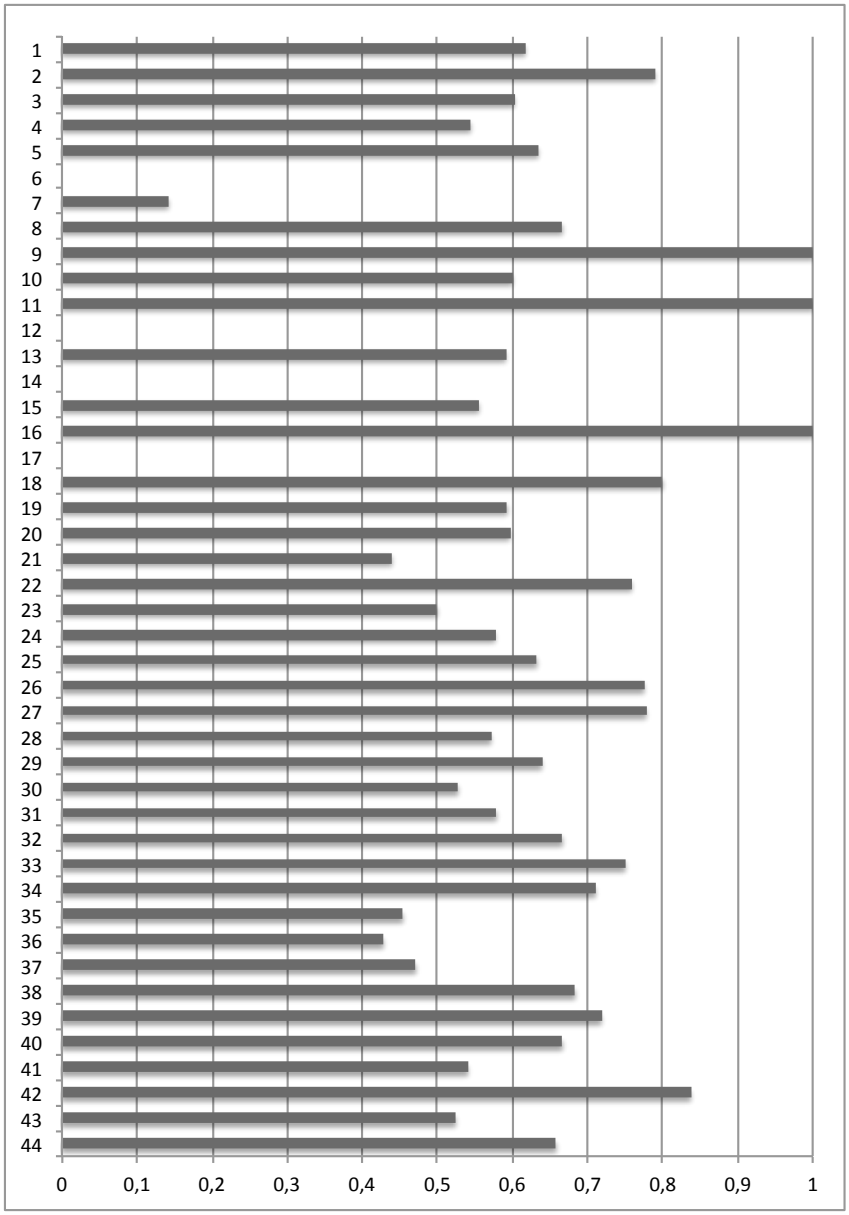


Fig. 6.12. The gain in precision due to our expression aliasing analysis, for the nullness analyses. For each benchmark, it shows how much is gained by the use of the expression aliasing information. For instance, the number of warnings for benchmark 1 (`nti`), using the aliasing information, is only 61% of the number of warnings for the same benchmark when no aliasing information is used.

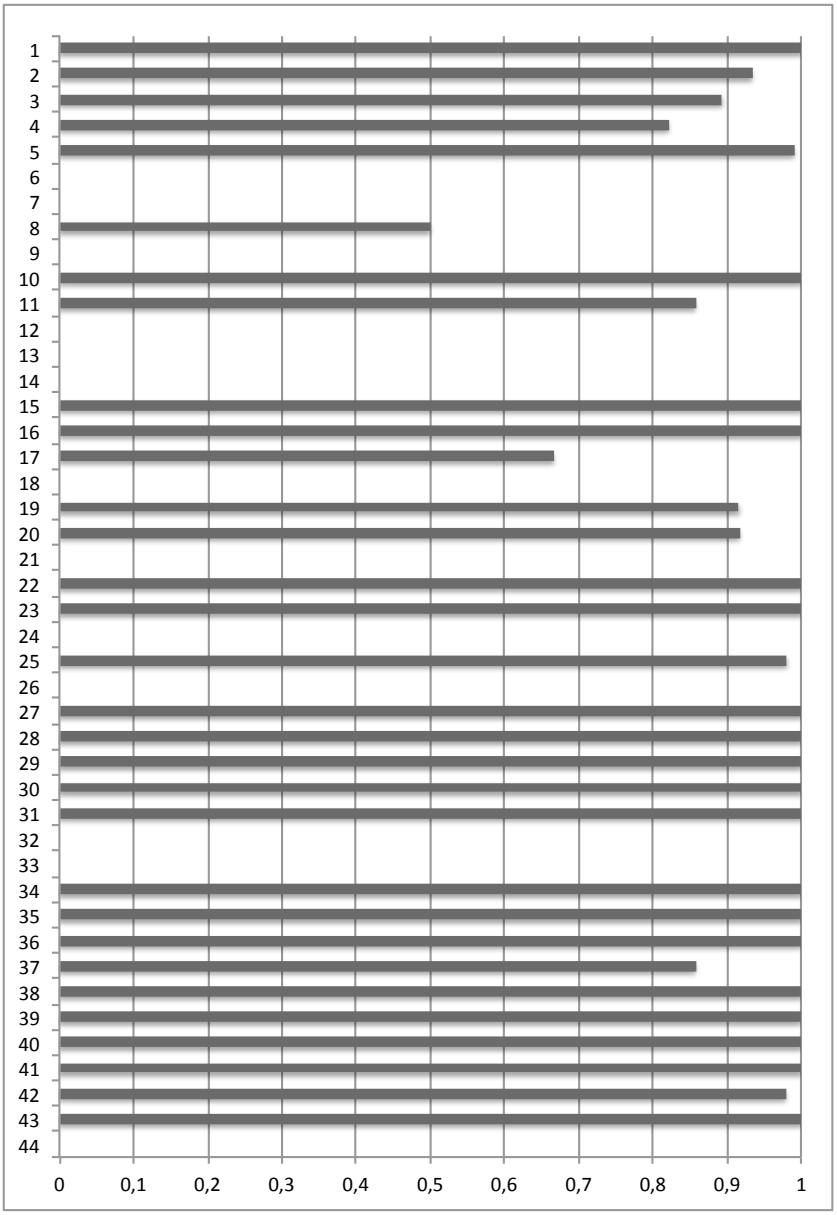


Fig. 6.13. The gain in precision due to our expression aliasing analysis, for the termination analyses. For each benchmark, it shows how much is gained by the use of the expression aliasing information. For instance, the number of warnings for benchmark 4 (`JFlex`), using the aliasing information, is only 82% of the number of warnings for the same benchmark when no aliasing information is used.

7

Related Work

The present thesis introduces a parameterized formal framework for static analyses of Java bytecode programs. The framework allows one to statically capture memory-related properties that the program under analysis has at run-time. Static analyses formalized inside the framework are constraint-based abstract interpretations of a concrete operational semantics of the target Java bytecode-like language formally defined in this thesis. In order to formalize a novel static analysis inside of this framework, the designer has to instantiate the parameters of the framework which are:

1. an abstract domain representing a property of interest and
2. a set of propagation rules representing a simulation of each normal and exceptional execution of each bytecode instruction of the target language.

When these parameters are instantiated, the framework automatically extracts from a program text a large system of constraints, representing the actual definition of the novel static analysis and introduces a set of requirements that the instantiation of the parameters has to satisfy. When this is the case, the thesis formally proves that the novel static analysis is sound and that there exists the least solution of the extracted system of constraints. Therefore, the designer does not have to prove the soundness of the overall static analysis, but only that the provided instantiation satisfies framework's requirements. This process is depicted in Fig. 1.3.

This thesis introduces the theoretical bases of the framework mentioned above, but also introduces its implementation inside of the Julia static analyzer. Chapters 5 and 6 introduce two practical instantiations of framework's parameters which give rise to two novel static analyses of heap-related properties of Java bytecode programs, which have also been implemented by Julia. Julia implements the framework like an abstract Java class dealing with graph construction, system of constraints extraction and its solution. On the other hand, data concerning the actual property of interest and propagation rules are left undefined, and they depend on different extensions of the abstract Java class which representing novel static analyses. Although the framework has been implemented inside Julia, its implementation is not one of the contributions of this thesis. On the other hand, implementations of the static analyses mentioned above are actual contributions of the thesis and their experimental evaluation is shown in the corresponding chapters.

The idea of abstract interpretation-based static analysis is not new, and since the paper introducing that technique [32] a lot of static analyses for both numerical and heap-related

properties as well as a lot of static analysis tools based on abstract interpretation have been presented. The goal of this thesis is not to provide another abstract interpretation-based static analysis, but to introduce a general technique for automatic definition of new heap-related static analysis of Java bytecode programs. The structure of the presented framework allows to deal with side-effects of non pure methods as well as with exceptional program executions. Moreover, one of the main contributions of this thesis is the soundness of the static analyses defined inside the framework, and this is a very important result.

Constraint-based approaches have been widely used for different types of program analyses. For instance, Gulwani et al. [44] introduce a constraint-based approach for discovering invariants involving linear inequalities, and generate weakest preconditions and strongest postconditons over the abstraction of linear arithmetic. Their constraints are Boolean combinations of quadratic inequalities over integer variables, and therefore their approach (although constraint-based) and goals are completely different from the ones of this thesis. Some other constraint-based techniques, successfully applied to the problem of discovering linear arithmetic invariants or inductive loop invariants for the verification of assertions, are for example [25, 29, 81].

Another very frequent application of constraints is for the problem of type inference [15, 19, 67, 89]. Usually, these techniques assume programs completely untyped, and then extract the constraints from the program text, represent them in the form of a directed graph, which is then solved by different algorithms similar to transitive closure. The abstract constraint graphs introduced in this thesis are similar to the trace graphs introduced by Palsberg and Schwartzbach in [67]. Recently, Pearce introduced a novel constraint-based formulation of flow typing, which is capable of extracting recursive types from constraints [71].

Nielson et al. [58, Chapter 3] presents a technique for constraint-based analysis of a simple functional language. Namely, the authors define a control flow analysis by developing a syntax directed specification of the problem followed by its constraint-based formulation and finally they show how these constraints can be solved. An overview of constraint-based program analyses is provided by Aiken in [14]. He underlines that this kind of analyses can be divided in two phases: constraint generation and constraint resolution. The former extracts from the program text a set of constraints related to the relevant information, while the latter solves these constraints. The author introduces set constraints, a widely used type of constraints, and shows how some classical problems such as standard dataflow equations, simple type inference and monomorphic closure analysis can be represented as particular instances of set constraint problems. Nevertheless, he does not consider any particular programming language. Moreover, the main goal of [14] is to propose different ways of solving the constraints. This thesis follows a similar direction, since the process of designing novel static analyses is divided in two phases: construction of the abstract constraint graph a nd its solution. On the other hand, this thesis provides a general parameterized formal framework for interprocedural static analyses of Java bytecode, and its analyses handle side-effects of methods as well as exceptional executions. To the best of our knowledge, ours is the first framework of this type.

Chapter 5 introduces the possible reachability analysis of program variables. Namely, a variable v of reference type reaches another variable w of reference type if there exists a sequence of fields $\mathtt{f}_1, \ldots, \mathtt{f}_n$ such that $v.\mathtt{f}_1.\ldots.\mathtt{f}_n = w$. Chapter 5 introduces a static analysis which for each program point p of a program infers a set of ordered pairs of

variables $\langle v, w \rangle$ of reference type such that there exists a program execution in which v might reach w at point p. This analysis belongs to the well-known group of *pointer analyses*, that improve the overall precision of other static analyses of programs. Plenty of works consider *pointer analyses*: in [46], more than 75 papers are surveyed. Different properties of pointers can be considered, which gives rise to distinct pointer analyses: *alias*, *sharing*, *points-to*, *escape* and *shape* analyses.

Possible (definitive) alias analysis discovers the pairs of variables that might (must) point to the same memory location. This information reveals more details comparing to the reachability analysis described in this thesis. Namely, if two variables are aliased, they are also reachable one from the other, but the opposite is in general false. Sharing analysis [82] 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. In this case, the reachability analysis gives a more detailed information about those variables. In particular, if a variable is reachable from another one, they must also share, but the opposite is in general false.

Points-to analysis computes the objects that a pointer variable my refer to at run-time. Usually, points-to analysis performs a conservative approximation of the heap, which is then used for calculating a points-to information for the whole program. Many works deal with this analysis, either by providing a formal framework or by introducing an efficient tool [45, 49–51, 75, 80, 83]. The `jpaul` tool [79] of [80] implements a pointer analysis which constructs, at each program point, a points-to graph describing how local variables and object fields point to objects. The authors explain how to use its results to perform program optimization (stack-allocation of local objects) and identify pure methods (i.e., without side-effects). The points-to graphs are precise approximations of the run-time heap memory and can be used to over-approximate the reachability information. They are much more concrete than the reachability property itself.

The goal of shape analysis is to determine the *shape invariants* describing the program's data structures [24, 27, 37, 77, 78]. Shape analyses are quite concrete and hence capture aliasing and points-to information, as well as some more accurate properties of data structures such as cyclicity or acyclicity. These properties are often encoded as first-order formulas and theorem proving is used to determine their validity. Shape analyses also contain a very precise approximation of the run-time heap memory, from which reachability can be extracted. For example, the list of instrumentation predicates introduced in [78] can be enriched with

$$\varphi_{r_x}(v) = \exists s_1, \ldots, s_k \in Sel. \exists v_1, \ldots, v_k \in Var. x(v_1) \wedge \bigwedge_{i=1}^{k-1} s_i(v_i, v_{i+1}) \wedge v_k = v,$$

whose meaning is that a pointer variable x reaches a location bound to v along some arbitrary fields. In order to verify whether a pointer variable x reaches a pointer variable y, it is necessary to check the satisfiability of the formula: $\exists v \in V. y(v) \wedge \varphi_{r_x}(v)$. The main disadvantage of these analyses is their high worst-time complexity, which may be even exponential. One important difference among the papers mentioned above is the way they represent abstract states (shape-graphs, logical structures, explicit reachability predicates). Although these analyses are precise, they are consequently often expensive and sometimes limited to some particular data structure, such as linked lists. Some papers consider only a fragment of a real programming languages or their analyses are intra-procedural and do not analyze real-life applications. On the other hand, the cost of the

inter-procedural reachability analysis introduced in this thesis which is aware of side-effects and which deal with exceptional execution is low, and can be applied to an arbitrary data structure implemented in a real-life program, as its experimental evaluation shows (Section 5.5). In the case of Java language, there exist some dynamic shape analyses dealing with it, such as for instance [47, 72]. But dynamic analyses are only sound w.r.t. the execution traces that are generated at run-time and analyzed. As a consequence, they cannot be taken as basis for a sound static reachability analysis. There are two static shape analyses for Java: there is an intraprocedural approach introduced in [30], as well as an intreprocedural one [54]. Although the latter can analyze interprocedural Java programs, the exceptional paths are not mentioned there. Experiments reported in that article show that the analysis of a program of 3,705 statements requires 35.11 seconds; libraries have not been included in the analysis. The reachability analysis analyzes 112,423 statements in 32 seconds (Figures 5.8 and 5.10, see the case of `OnWatch`). Libraries are analyzed along the application. If one considers that sharing is needed before reachability, the total run-time of the reachability analysis amounts to 47 seconds, but the analyzed code base of 112,423 statements is 30 times larger than their 3,705 statements. There is no report on the precision of the analysis in [54] w.r.t. reachability information, but the major difference in the computational cost of the two analyses is apparent. It is true that experiments over reachability have been performed on a multicore hardware, which is potentially faster than the one used in [54], but sharing and reachability analyses are performed sequentially in Julia, so that only one core is used for them.

There already exists a notion of reachability in literature, slightly different from ours [57]. The meaning of the *reachability predicate* is to determine whether a memory location reaches another one, usually along one particular field of the structure of interest. Our definition of reachable locations deals with arbitrary objects and examines all possible fields these objects might have. Shape analysis has been also studied from the predicate abstraction's point of view [22,23]. For instance, [21,28,35] use the reachability predicate during program's abstraction.

The approach closest to the reachability analysis is introduced in [39]. The authors consider a simple Java-like language and define a notion of reachability that coincides with the one introduced in Chapter 5: the definition of the analysis and the propagation rules are however completely different. This definition of reachability is actually inspired by the representation of the memory introduced in [82]. The static analysis proposed in [39] is based on abstract interpretation and uses the same abstract domain Reach proposed in this thesis. Although these two approaches use two different target languages (this thesis considers almost full Java bytecode with exceptions), it is still possible to compare the static analyses these approaches introduce. Some advantages of the reachability analysis introduced in this thesis are:

- 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;
- we deal with exceptions;
- the implementation of our analysis fully corresponds to its formalization;
- we provide an experimental evaluation of our analysis on real-life Java and Android applications and hence show its usefulness.

Chapter 6 introduces the definite expression aliasing analysis. Namely, for each program point p and each variable v available at that point, this analysis infers a set of expression A such that each expression from A for every possible execution of the program must have the same value of v when point p is reached. Alias analyses belong to the large group of pointer analyses [46], and their task is to determine whether a memory location can be accessed in more than one way. There exist two types of alias analyses: possible (may) and definite (must). The former detects those pairs of variables that might point to the same memory location. There are very few tools performing this analysis on Java programs (e.g., `WALA` [12], `soot` [9], `JAAT` [66]). The latter under-approximates the actual aliasing information and the analysis introduced in this thesis is the first of this type dealing with Java bytecode programs and providing expressions aliased to variables. Similarly to our approach, the authors of [38] deal with definite aliasing, but their *must-aliasing* information is used for other goals and they do not deal with aliasing expressions.

The definite expression aliasing analysis is also related to the well-known *available expression analysis* [13] where, however, only variables of primitive type are considered, hence it is much easier to deal with side-effects. Fields can be sometimes transformed into local variables before a static analysis is performed [16], but this requires a preliminary modification of the code, while this thesis deals with more general expressions than just fields.

The analysis introduced in Chapter 6 can also be related to the well-known technique of global value numbering [18, 42, 43, 48, 73, 76], which is a classic analysis for finding must-equalities in programs, hardly used by compilers for a lot of optimizations. It determines equivalent computations inside a program and then eliminates repetitions. Checking equivalence of program expressions is an undecidable problem, and the tools dealing with this problem just try to under-approximate the actual sets of equivalent expressions by considering equivalent operands connected by equal operations. This form of equivalence, where the operators are treated as uninterpreted functions, is also called Herbrand equivalence [55, 56, 76], and global value numbering helps discovering it. There exist two main approaches in global value numbering. The first one discovers all possible Herbrand equivalences [43, 48], while the second one discovers only those Herbrand equivalences created from program sub-expressions [18]. We infer, for each program point p, and each variable v available at p a set of expressions among all possible expressions available at p whose value be equal to v, for any possible program execution. This is similar to the first approach mentioned above. On the other hand, uur technique is applied to a real life programming language, Java bytecode, and is an inter-procedural static analysis, while the papers mentioned above deal with an imperative while language and are intra-procedural.

8

Conclusion

Dynamic allocation of objects is heavily used in (complex and large) real-life programs. When such objects are instantiated on demand, their number might not be statically known. Moreover, objects in general contain references to other objects (i.e., fields in object-oriented parlance) and those references are usually updated at run-time. The most interesting properties of present software products are related to the objects that they dynamically allocate in memory, rather than to non-heap allocated, primitive values such as integers. In this thesis we dealt with static analyses of Java bytecode programs that are, in particular, related to memory-related properties. More precisely, we introduced a general parameterized framework for constrainted-based static analyses of Java bytecode programs, that allows us to formalize different static analyses dealing with both numerical properties of program variables and memory-related properties. These analyses belong to the group of constraint-based analyses. Moreover, the structure of the framework allows us to define static analyses that deal with both side-effects of the methods, and with their exceptional executions. Finally, we simplify the proofs of correctness of the static analyses formalized inside the framework, by providing a general methodology of proof.

Our approach makes a clear difference between the work done by the designer of a new static analysis and the general results that hold inside the framework. Namely, the designer should provide a mathematical representation of the property of interest in terms of an abstract domain whose elements abstract away some irrelevant pieces of information from the concrete states, as well as the propagation rules for every possible arc appearing in the ACG. Moreover, the designer should show that the requirements characterized by the framework are satisfied. These requirements concern different properties about the parameters instantiated by the designer: the abstract domain should satisfy the ACC condition and should form a Galois connection with the concrete domain of states, while the propagation rules should be monotonic and each rule should soundly mimic the semantics of the corresponding bytecode instruction. On the other hand, the construction of both the ACG from the program under analysis and a system of constraints from the ACG annotated by the designer is automatically done by the framework. Moreover, when the requirements characterized by the framework are satisfied by an instantiation of its parameters, the results shown in this thesis guarantee that there exists the least solution of the system of constraints and that this solution represents a sound approximation. Therefore, the designer does not have to consider the operational semantics of the Java bytecode language and prove that the abstract semantics of the *full* program under analysis actually

mimics the operational semantics of that program, but only that each propagation rule mimics the concrete semantics of the bytecode instruction approximated by that rule. Obviously, this is not a simple task, but it is quite easier in comparison to the full proof of soundness.

There is also another important benefit of using our framework. From an implementational point of view, we can provide an abstract (in terms of Java) class implementing the generic engine for constraint generation and solving. Each new specific static analysis is a concrete subclass of that abstract class, providing an implementation for a few methods, where the specific static analyses deviates from the general framework. This largely simplifies the implementation of new static analyses. For instance, the developer need not bother about the implementation of the constraints and the strategy for their solution and inherits highly optimized and already debugged code.

Our framework has been used to *define*, *formally prove sound* and *implement* different static analyses dealing with memory-related properties: the *Possible Reachability Analysis of Program Variables* and the *Definite Expression Aliasing Analysis*. Both analyses represent a possible instantiation of our general parameterized constraint-based framework for static analyses of Java bytecode programs.

Possible Reachability Analysis of Program Variables is an example of a *possible analysis*. Reachability from a program variable v to a program variable w states that starting from v it is possible to follow a path of memory locations that leads to the object bound to w. This useful piece of information is important for improving the precision of other static analyses, such as side-effects, field initialization, cyclicity and path-length analysis, as well as more complex analyses built upon them, such as nullness and termination analysis. It determines, for each program point p, a set of ordered pairs of variables $\langle v, w\rangle$ such that v *might reach* w at p when the program is executed on an arbitrary input. On the other hand, if a pair $\langle v, w\rangle$ is not present in our over-approximation at p, it means that v *definitely does not reach* w at p.

On the other hand, the *Definite Expression Aliasing Analysis* infers, for each variable v at each program point p, a set of expressions whose value at p is equal to the value of v at p, for every possible execution of the program. Namely, it determines which expressions *must* be aliased to local variables and stack elements of the Java Virtual Machine. The approximation produced by this analysis is another useful piece of information for an inter-procedural static analyzer, since it can refine other static analyses at conditional statements or assignments.

Both Possible Reachability Analysis of Program Variables and Definite Expression Aliasing Analysis have been implemented in the Julia static analyzer for Java and Android. A collection of real-life benchmarks has been analyzed by Julia in order to experimentally evaluate these novel static analyses. The results of these evaluations show that both analyses improve the precision of the principal Julia's checkers, that are *nullness* and *termination*. In the case of the reachability analysis, its presence actually reduces the total run-time of both the nullness and termination checkers. This is because the reachability improves their supporting analyses, in particular side-effects analysis, and prevents them from generating too much spurious information. Moreover, the precision of both principal tools has been improved. Similarly, in the case of the aliasing analysis, the precision of Julia's principal tools has been improve, in this case drastically, but the total run-times of these tools increased a bit. Anyway, this is a good trade-off between run-times of our tools and improved precision.

Our novel analyses can be seen as abstractions of a more concrete, shape analysis, that statically builds a conservative description of possible shapes that data structures might have at run-time. The main issue with the shape analysis is that, although it is very precise, its complexity is high. Moreover, to the best of our knowledge, there are very few static shape analysis for Java programs, so we were not able to actually compare our novel and more abstract static analyses with already existing and more concrete shape analysis. The only thing we could notice is that the run-time of our reachability analysis when it is applied to our largest benchmark is much lower than the run-time of the shape analysis of the largest benchmark used for shape analysis.

There exist some other memory-related static analyses formalized in our framework and implemented inside the Julia tool. We mention some of them: Possible Sharing Analysis, Field Initialization Analysis, Creation Points Analysis, Side-Effects Analysis, etc. We are working on a formalization and implementation of many other constraint-based static analyses, and the framework introduced in this thesis is of great help in doing that. In particular, we are working on an information flow analysis which is being formalized in the same framework and implemented inside Julia.

Julia is a semantical tool for static analysis of Java and Android. Currently, it can analyze only monothreaded programs. Static analysis of multithreaded Java programs is a big challenge that we would like to face in the future. It is clear that, in the case of multithreaded programs, proof of soundness of their static analysis would be even more difficult. Therefore, it would be interesting to extend our framework in such a way that it is possible to formalize, prove correct and implement also constraint-based static analyses of multithreaded Java bytecode programs. This goal requires further research.

References

1. Android SDK tools r12: - http://developer.android.com/tools/revisions/platforms.html.
2. International Termination Ccompetition: http://termination-portal.org/wiki/Termination_Competition.
3. Jflex: http://jflex.de/.
4. Julia - A Static Analyzer for Java and Android: - http://www.juliasoft.com.
5. Moonblink: http://moonblink.googlecode.com/svn/trunk/.
6. Non-termination Analyzer NTI: http://personnel.univ-reunion.fr/epayet/Research/NTI/NTI.html.
7. Numerical Simplex Implementation: http://sourceforge.net/projects/lisimplex.
8. plume-lib: http://code.google.com/p/plume-lib/.
9. Soot: A Java Optimization Framework - http://www.sable.mcgill.ca/soot/.
10. Typoweather: http://code.google.com/p/typoweather/.
11. http://f-droid.org/repository/browse/.
12. WALA: T.J. Watson Libraries for Analysis - http://wala.sourceforge.net/.
13. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
14. A. Aiken. Introduction to Set Constraint-Based Program Analysis. *Science of Computer Programming*, 35(2):79–111, 1999.
15. A. Aiken and E. L. Wimmers. Type Inclusion Constraints and Type Inference. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FPCA 1993)*, pages 31–41. ACM, 1993.
16. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Ramírez. From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In *Proceedings of the 17th Static Analysis Symposium (SAS 2010)*, volume 6337 of *Lecture Notes in Computer Science*, pages 100–116. Springer, 2010.
17. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proceedings of the 16th European Symposium on Programming (ESOP 2007)*, pages 157–172. Springer, 2007.
18. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1988)*, pages 1–11. ACM, 1988.
19. C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, volume 3586 of *Lecture Notes in Computer Science*, pages 428–452. Springer, 2005.
20. K.R. Apt and G.D. Plotkin. Countable Non-determinism and Random Assignement. *Journal of the ACM.*, 33(4):724–767, 1986.

21. I. Balaban, A. Pnueli, and L. D. Zuck. Shape Analysis by Predicate Abstraction. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2005)*, volume 3385 of *Lecture Notes in Computer Science*, pages 164–180. Springer, 2005.
22. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proceedings of the 22nd Conference on Programming Language Design and Implementation (PLDI 2001)*, volume 36, pages 203–213. ACM, 2001.
23. T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic Predicate Abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27:314–343, 2005.
24. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape Analysis for Composite Data Structures. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, Lecture Notes in Computer Science, pages 178–192. Springer, 2007.
25. A. R. Bradley and Z. Manna. Verification Constraint Problems with Strengthening. In *Proceedings of the 3th International Colloquium on Theoretical Aspects of Computing (ICTAC 2006)*, volume 4281 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2006.
26. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 8(35):677–691, 1986.
27. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. In *Proc. of the 36th Symposium on Principles of Programming Languages (POPL 2009)*, pages 289–300. ACM, 2009.
28. S. Chatterjee, S.K. Lahiri, S. Qadeer, and Z. Rakamaric. A Low-Level Memory Model and an Accompanying Reachability Predicate. *International Journal on Software Tools for Technology Transfer (STTT 2009)*, 11(2):105–116, 2009.
29. M. Colón, S. Sankaranarayanan, and H. Sipma. Linear Invariant Generation Using Non-linear Constraint Solving. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2003.
30. J. C. Corbett. Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs. *ACM Transactions on Software Engineering and Methodology*, 9(1):51–93, 2000.
31. P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, 1996.
32. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252. ACM, 1977.
33. P. Cousot and R. Cousot. Constructive Versions of Tarski's Fixed Point Theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.
34. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th Symposium on Principles of Programming Languages (POPL 1979)*, pages 269–282. ACM, 1979.
35. D. Dams and K. S. Namjoshi. Shape Analysis through Predicate Abstraction and Model Checking. In *Proceedings of the 4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2003)*, pages 310–324. Springer, 2003.
36. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
37. D. Distefano, P.W. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic. In *Proceedings of the 2th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.
38. S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective Typestate Verification in the Presence of Aliasing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 133–144. ACM, 2006.

39. S. Genaim and D. Zanardini. The Acyclicity Inference of COSTA. In *Proceedings of the International Workshop on Termination (WST 2010)*, 2010.
40. G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. W. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices*. Springer, 1980.
41. G. A. Grätzer. *General Lattice Theory*. Birkhäuser Verlag, 1978.
42. S. Gulwani and G. C. Necula. Global Value Numbering Using Random Interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, pages 342–352. ACM, 2004.
43. S. Gulwani and G. C. Necula. A Polynomial-Time Algorithm for Global Value Numbering. *Science of Computer Programming*, 64(1):97–114, 2007.
44. S. Gulwani, S. Srivastava, and R. Venkatesan. Program Analysis as Constraint Solving. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 281–292. ACM, 2008.
45. B. C. Hardekopf. *Pointer Analysis: Building a Foundation for Effective Program Analysis*. PhD thesis, 2009.
46. M. Hind. Pointer Analysis: Haven't We Solved this Problem Yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001)*, pages 54–61. ACM, 2001.
47. M. Jump and K. S. McKinley. Dynamic Shape Analysis via Degree Metrics. In H. Kolodner and G. L. Jr. Steele, editors, *Proceedings of the 8th International Symposium on Memory Management (ISMM)*, pages 119–128. ACM, 2009.
48. G. A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1973)*, pages 194–206. ACM, 1973.
49. O. Lhoták. *Program Analysis Using Binary Decision Diagrams*. PhD thesis, McGill University, 2006.
50. O. Lhoták and L. J. Hendren. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC 2003)*, pages 153 –169. Springer, 2003.
51. O. Lhoták and A. C. Kwok-Chiang. Points-to Analysis with Efficient Strong Updates. In *Proceedings of the 38th Symposium on Principles of Programming Languages (POPL 2011)*, pages 3–16. ACM, 2011.
52. T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
53. F. Logozzo and M. Fähndrich. On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In *Proceedings of the 17th International Conference on Compiler Construction (CC 2008)*, volume 4959 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2008.
54. M. Marron, M. V. Hermenegildo, D. Kapur, and D. Stefanovic. Efficient Context-Sensitive Shape Analysis with Graph Based Heap Models. In L. J. Hendren, editor, *Proc. of the 17th International Conference on Compiler Construction (CC 2008)*, volume 4959 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2008.
55. M. Müller-Olm, O. Rüthing, and H. Seidl. Checking Herbrand Equalities and Beyond. In *Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, pages 79–96, Berlin, Heidelberg, 2005. Springer-Verlag.
56. M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural Herbrand Equalities. In *Proceedings of the 14th European Conference on Programming Languages and Systems (ESOP 2005)*, pages 31–45, Berlin, Heidelberg, 2005. Springer-Verlag.
57. G. Nelson. Verifying Reachability Invariants of Linked Structures. In *Proceedings of the 8th Symposium on Principles of Programming Languages (POPL 1983)*, pages 38–47, 1983.
58. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, corrected edition, 2004.

59. Đ. Nikolić and F. Spoto. Constraint-based Static Analyses for Java Bytecode Programs - submitted. http://profs.sci.univr.it/~nikolic/download/CBFramework/CBFramework.pdf.
60. Đ. Nikolić and F. Spoto. Definite Expression Aliasing in Java Bytecode Programs: a Constraint-Based Static Analysis - submitted. http://profs.sci.univr.it/~nikolic/download/ICTAC2012/ICTAC2012Ext.pdf.
61. Đ. Nikolić and F. Spoto. Inferring Complete Initialization of Arrays - submitted. http://profs.sci.univr.it/~nikolic/download/IJCAR2012/IJCAR2012Ext.pdf.
62. Đ. Nikolić and F. Spoto. Automaton-based array initialization analysis. In A.-H. Dediu and C Martín-Vide, editors, *Proceedings of the 6th International Conference on Language and Automata Theory and Applications (LATA 2012)*, volume 7183 of *Lecture Notes in Computer Science*, pages 420–432. Springer, Heidelberg, March 2012.
63. Đ. Nikolić and F. Spoto. Definite Expression Aliasing Analysis for Java Bytecode. In *Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing (ICTAC 2012)*, volume 7521 of *Lecture Notes in Computer Science*, pages 74–89. Springer-Verlag Berlin Heidelberg, 2012.
64. Đ. Nikolić and F. Spoto. Reachability Analysis of Program Variables. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR 2012)*, volume 7364 of *Lecture Notes in Artificial Intelligence*, pages 423–438. Springer-Verlag Berlin Heidelberg, 2012.
65. Đ. Nikolić and F. Spoto. Inferring Complete Initialization of Arrays. *Theoretical Computer Science*, 2013.
66. F. Ohata and K. Inoue. JAAT: Java Alias Analysis Tool for Program Maintenance Activities. In *Proceedings of the 9th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 232–244. IEEE, 2006.
67. J. Palsberg and M. I. Schwartzbach. Object-oriented Type Inference. In *Proceedings of Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 1991)*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161. ACM, 1991.
68. M. M. Papi, M. Ali, T. L. Correa, J. H. Perkins, and M. D. Ernst. Practical Pluggable Types for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 201–212. ACM, 2008.
69. É. Payet and F. Spoto. Magic-sets transformation for the analysis of java bytecode. In *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 452–467. Springer, 2007.
70. É Payet and F. Spoto. Static Analysis of Android Programs. *Information & Software Technology*, 54(11):1192–1201, 2012.
71. D. J. Pearce. A Constraint-Based Calculus for Flow Typing. Technical Report ECSTR12-10, Victoria University of Wellington, 2012.
72. S. Pheng and C. Verbrugge. Dynamic Shape and Data Structure Analysis in Java. Technical report, McGill University, School of Computer Science, 2005.
73. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1988)*, pages 12–27. ACM, 1988.
74. S. Rossignoli and F. 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 2006)*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2006.
75. A. Rountev, A. Milanova, and B. G. Ryder. Points-to Analysis for Java Using Annotated Constraints. In *Proceedings of the 16th Proceedings of Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2001)*, pages 43–55. ACM, 2001.
76. O. Rüthing, J. Knoop, and B. Steffen. Detecting Equalities of Variables: Combining Efficiency with Precision. In *Proceedings of the 6th International Symposium on Static Analysis (SAS 1999)*, pages 232–247. Springer-Verlag, 1999.

77. M. Sagiv, T. Reps, and R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20:1–50, 1998.
78. M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24:217–298, May 2002.
79. A. D. Salcianu. jpaul – Java Program Analysis Utilities Library. `http://jpaul.sourceforge.net/`.
80. A. D. Salcianu. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. PhD thesis, 2006.
81. S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-Based Linear-Relations Analysis. In *Proceedings of the 11th International Symposium on Static Analysis (SAS 2004)*, volume 3148, pages 53–68. Springer, 2004.
82. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *Proceedings of the 12th International Symposium on Static Analysis (SAS 2005)*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2005.
83. Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th Symposium on Principles of Programming Languages (POPL 2011)*, pages 17–30. ACM, 2011.
84. F. Spoto. Nullness Analysis in Boolean Form. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008)*, pages 21–30. IEEE, 2008.
85. F. Spoto. Precise Null-pointer Analysis. *Software and System Modeling*, 10(2):219–252, 2011.
86. F. Spoto and M. D. Ernst. Inference of Field Initialization. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 231–240. ACM, 2011.
87. F. Spoto, F. Mesnard, and É. Payet. A Termination Analyzer for Java Bytecode Based on Path-Length. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(3):1–70, 2010.
88. A. Tarski. A Lattice Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.
89. T. Wang and S. F. Smith. Precise Constraint-Based Type Inference for Java. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 99–117. Springer, 2001.

Index