Reachability Analysis of Program Variables

ĐURICA NIKOLIĆ, University of Verona and Microsoft Research - University of Trento Centre for Computational and Systems Biology
FAUSTO SPOTO, University of Verona

Reachability from a program variable v to a program variable w states that from v, it is possible to follow a path of memory locations that leads to the object bound to w. We present a new abstract domain for the static analysis of possible reachability between program variables or, equivalently, definite unreachability between them. This 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 and prove correct our reachability analysis for Java bytecode, defined as a constraint-based analysis, where the constraint is a graph whose nodes are the program points and whose arcs propagate reachability information in accordance to the abstract semantics of each bytecode instruction. For each program point p, our reachability analysis produces an overapproximation of the ordered pairs of variables $\langle v, w\rangle$ such that v *might reach* w at p. Seen the other way around, if a pair $\langle v, w\rangle$ is not present in the overapproximation at p, then v *definitely does not reach* w at p. We have implemented the analysis inside the Julia static analyzer. Our experiments of analysis of nontrivial Java and Android programs show the improvement of precision due to the presence of reachability information. Moreover, reachability analysis actually reduces the overall cost of nullness and termination analysis.

Categories and Subject Descriptors: D.24 [**Software Engineering**]: Software/Program Verification—*Correctness proofs, Formal methods*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis, Algebraic approach to sementics*

General Terms: Theory, Verification, Languages

Additional Key Words and Phrases: Static analysis, constraint-based analysis, abstract interpretation, reachability analysis, pointer analysis, Java bytecode

ACM Reference Format:
Nikolić, Đ. and Spoto, F. 2013. Reachability analysis of program variables. ACM Trans. Program. Lang. Syst. 35, 4, Article 14 (December 2013), 68 pages.
DOI: http://dx.doi.org/10.1145/2529990

1. INTRODUCTION

Static analysis of computer programs lets one gather information about their runtime behavior before even running them. Hence, it becomes possible to prove that they will not perform any illegal operation, such as a division by zero or a dereference of null, will not lead to erroneous executions, such as infinite loops, will not divulge information in incorrect ways (such as security authorizations or GPS position of mobile devices). Static analysis has a long story now and can be formalized in many ways. In particular, here we follow the abstract interpretation approach [Cousot and Cousot 1977], which

This work is an extended version of Nikolić and Spoto [2012c].
Corresponding author's email: nikolic.durica@gmail.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2013 ACM 0164-0925/2013/12-ART14 $15.00
DOI: http://dx.doi.org/10.1145/2529990

allows one to define a static analysis from the formal specification of the property of interest and of the semantics of the programming language.

Dynamic allocation of objects is heavily used in (complex and large) real-life programs. When such objects are instantiated on demand, their number might be statically unknown. Moreover, objects in general contain references to other objects i.e., (*fields* in object-oriented parlance), and those references are typically modified at runtime. The most interesting properties of current software are related to the objects that they dynamically allocate in memory rather than to primitive values, such as integers. Hence, it is not a surprise that a huge amount of literature tackles the analysis of memory-related properties. There are very general techniques, such as shape analysis [Sagiv et al. 1998, 2002], that build, statically, a conservative description of the possible shapes that data structures might take at runtime. There are also more abstract analyses, typically less precise but more efficient. For instance, aliasing analysis exists in uncountable variations and expresses the fact that two variables might (or must always) point to the same location (i.e., they are possibly or definitely aliased to each other). There is also sharing analysis [Secci and Spoto 2005], whose goal is to determine if two variables might ever be bound to overlapping data structures. In other terms, two variables share if they might reach the same location at runtime.

In this article, we present, formalize, prove correct, and implement a new abstraction of the runtime, dynamically allocated memory of computer software. This abstraction is called *reachability*. 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 another abstraction of the memory, such as the result of a sharing or shape analysis. However, we want an analysis that uses the most abstract domain for reachability analysis here, that coincides with the reachability property itself. In other words, since the property of reachability is a way of specifying the ordered pairs of variables such that the first reaches the second, the simplest abstraction for reachability is exactly that: a set of ordered pairs of variables such that the first reaches the second. Nothing else is added or missing from the abstract domain. In this article, 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[1], which allows us to discuss the actual benefits of our reachability analysis.

We observe that ours is a possible reachability analysis in the sense that it provides, for each *program point* p, an over-approximation of the actual reachability information holding at p. This over-approximation contains the full actual reachability information at p, but it might contain some *spurious* pairs of variables as well (i.e., *false positives*), since reachability is an undecidable property. If we look at this overapproximation the other way around, we can assert, definitely, that pairs of variables not in the overapproximation do not reach each other at that program point. Actually, it is simpler to think in terms of propagation of a possible overapproximation and our formalization is, consequently, in terms of possible reachability. However, it is the complementary definite unreachability that is later used for program analysis. Namely, by just considering our work related to the Julia static analyzer, we highlight 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

[1]`http://www.juliasoft.com`.

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`. Since we compute an overapproximation of the reachability pairs, those pairs that do not belong to the overapproximation definitely do not reach each other. If we used nonsharing rather than nonreachability information, that 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 constructors of its defining class before that field is read, ever, in the program. Spotting this frequent situation is important for many analyses, including nullness [Papi et al. 2008; Spoto and Ernst 2011]. Hence, we want to know if a field read operation `a=expression.f` inside a constructor can actually read field `f` of `this`, the object being initialized by the constructor. This might happen only if `this` is an alias of `expression` that we can conservatively approximate by checking if `this` reaches `expression`. In particular, if we know that `this` definitely does not reach `expression` right before the assignment is executed, then `this` cannot be an alias of `expression`, and that assigment will not read the field `f` of `this`. Here, we are using possible reachability as an approximation of possible aliasing. As before, an overapproximation of the reachability pairs means that the other pairs do not reach each other. Again, sharing would be less precise here. We observe that `expression` is, in Java bytecode, held in a stack variable, hence we are testing reachability between variables here.

—*For Cyclicity Analysis.* If we know that `b` holds an acyclical data structure, then the assignment `a.f=b` might make `a` *cyclical* (i.e., hold a cyclical data structure), only if `b` reaches `a`. Originally, this analysis was built upon sharing information [Rossignoli and Spoto 2006], but analysis of reachable variables gives better precision, as already observed in Genaim and Zanardini [2012].

—*For Path-Length Analysis.* Path length is a measure of data structures used in termination analysis [Spoto et al. 2010]. 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 [Spoto et al. 2010]. 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 just mentioned. They are used as building blocks of larger tools, such as a nullness checker and a termination checker tool. The former spots the points where a program might throw a null-pointer exception at runtime, while the latter spots which method calls might diverge at runtime. 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 article, we provide an experimental evaluation of our reachability analysis. Namely, we show that reachability analysis is more precise than a sharing analysis, when the property of interest is reachability. 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 nontrivial Java and Android programs. On the other hand, the performance of the termination tool is not improved for the programs priviosly mentioned. Instead, it is well improved for the analysis of a set of programs from the international termination competition[2], where six 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 hundred lines of source code at most), which means that the shape of the memory can be more easily inferred; they ban complications, such as 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 runtime of the tools (reachability runtime included). This can be actually explained: reachability increases the precision of other analyses (side-effects, field initialization, cyclicity, etc.) 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 article is organized as follows. Section 2 introduces the state of the art and relates our work to the existing approaches, showing similarities and differences. Section 3 introduces syntax and operational semantics of the Java bytecode-like language that we consider in this article. Section 4 formally defines different notions of reachability. Section 5 presents our abstract interpretation-based static analysis, together with formal proofs of correctness. Section 6 shows the application of our analysis to many real-life examples, its precision, and the way it affects other analyses performed by our static analyzer Julia. Section 7 concludes. Most proofs are kept in the Appendix.

2. RELATED WORK

Reachability 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 Hind [2001], 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. If two variables are aliased, they are also reachable from each other, but the opposite is in general false. Sharing analysis [Secci and Spoto 2005] determines if two variables might ever be bound to overlapping data structures, that is, two variables share if they might reach the same location at runtime. 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 might refer to at runtime. Usually, points-to analysis performs a conservative approximation of the heap, which is then used to compute 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 [Salcianu 2006; Lhoták and Hendren 2003; Lhoták 2006; Lhoták and Chung 2011; Smaragdakis et al. 2011; Rountev et al. 2001; Hardekopf 2009]. The `jpaul` tool[3] of Salcianu [2006] implements a pointer analysis that constructs, at each

[2] `http://termination-portal.org/wiki/Termination_Competition`.
[3] `http://jpaul.sourceforge.net`.

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 runtime heap memory, and some of their formulations can be used to overapproximate reachability information. They are often much more concrete than reachability itself, which is our abstract domain.

The goal of shape analysis is to determine the *shape invariants* describing the program's data structures [Sagiv et al. 1998, 2002; Calcagno et al. 2009; Berdine et al. 2007; Distefano et al. 2006]. 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 runtime heap memory from which reachability can be extracted. For example, we can enrich the list of instrumentation predicates introduced in Sagiv et al. [2002] 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 if a pointer variable x reaches a pointer variable y, we should check the satisfiability of the formula: $\exists v \in V.y(v) \wedge \varphi_{r_x}(v)$. The main difference between these papers 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 language or are defined for a toy language without method calls or their techniques do not scale to large, real-life applications.

There already exists a notion of reachability in literature [Nelson 1983], slightly different from ours. The meaning of the *reachability predicate* there is to determine if a memory location reaches another one, usually along a particular field of the structure of interest. Our definition of reachable locations deals with arbitrary objects and examines all fields of these objects. Shape analysis has been also studied from the point of view of predicate abstraction [Ball et al. 2001, 2005]. For instance, some works [Dams and Namjoshi 2003; Balaban et al. 2005; Chatterjee et al. 2009] use the reachability predicate during the abstraction of the program.

The approach closest to ours is in Genaim and Zanardini [2010, 2012]. The authors consider a simple Java-like language and define a notion of reachability that coincides with ours: the definition of the analysis and the propagation rules are, however, completely different from ours. This definition of reachability is actually inspired by the representation of the memory introduced in Secci and Spoto [2005]. The static analysis proposed in Genaim and Zanardini [2010, 2012] is based on abstract interpretation and uses the same abstract domain that we propose in this article, Although our target language is different (we consider almost the full Java bytecode with exceptions), we can still compare the static analyses introduced in these two papers, and we highlight some advantages of our analysis.

- —We explicitly handle the side-effects of the methods.
- —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.

The first point is rather complex. In general, a callee method might introduce reachability among its formal parameters, which is reflected in the introduction of reachability between the actual parameters passed by the caller. One needs to know the reachability between the formal parameters at the end of the callee to reconstruct the effects on the actual parameters of the caller. However, the formal parameters might be reassigned inside the callee, which complicates the task, since their reachability does not represent anymore that of the formal parameters. A solution is to introduce read-only copies of the formal parameters (*shallow variables*), but this increases the number of local variables in a method and consequently the cost of the analysis. We have instead used a technique that avoids the introduction of copies.

3. OPERATIONAL SEMANTICS

This section presents a formal operational semantics for Java bytecode, inspired by its standard informal semantics in Lindholm and Yellin [1999]. This is the same semantics used in Spoto and Ernst [2011]. A similar formalization, but in denotational form, has also been used [Payet and Spoto 2007; Spoto 2008; Spoto et al. 2010]. Another approach using a similar representation of bytecode in an operational setting is that of Albert et al. [2007], although, there, Prolog clauses encode the graph, while we work directly on it.

There exist some other formal semantics for Java bytecode. Our choice has been dictated by the desire of a semantics suitable for abstract interpretation: we want a single concrete domain to abstract (the domain of *states*), and we want the bytecode instructions to be state tranformers, always, also in the case of the conditional bytecode instructions and of those dealing with dynamic dispatch and exception handling. This is exactly the purpose of the semantics in Spoto and Ernst [2011], whose form highly simplifies the definition of abstract interpretations and their proof of soundness.

Java bytecode is the form of instructions executed by the Java Virtual Machine (JVM). Although it is a low-level language, it does support high-level concepts, such as objects, dynamic dispatching, and garbage collection. Our formalization is also at the Java bytecode level for several reasons. First, it is much simpler than Java: there is a relatively small number of bytecode instructions, compared to varieties of source statements, and bytecode instructions lack complexities like inner classes. Second, our implementation of reachability analysis is at the bytecode level, bringing formalism, implementation, and proofs closer. We require a formalization, since one of our goals is to prove the analysis sound.

3.1. Types

For simplicity, we assume that the only primitive type is `int`; that reference types are *classes* containing *instance fields* and *instance methods* and *arrays*. Our implementation handles all Java primitive and reference types as well as the rest of the bytecode instructions and the static fields and methods that, for simplicity, we do not consider in the present article. In particular, note that primitive types are not heap allocated in Java bytecode, so they are irrelevant with respect to reachability. Hence, it is enough to consider just one primitive type, since the others behave equivalently for what we are concerned with in this article. Interfaces are also missing from our formalization. We observe, however, that interfaces are relevant in Java at compilation time, while they have little to do with a dynamic semantics, which is what we are going to abstract. In particular, interfaces do not provide method implementations in Java bytecode, and hence the method lookup rule only considers the superclass chain in that language.

The fact that we do not consider static fields is a consequence of the formal complexity that they would introduce, since static fields are always in scope, at every program point. Static method calls would also complicate the semantics by duplicating the rules for method call: one for instance methods and one for static methods; and they would complicate the way the callee is found by dynamic lookup (from the dynamic type of an object, in the first case; from a fixed starting class, in the second case). Our concrete and abstract semantics would become too complex if static fields and methods were presented in formal terms in this article.

Definition 3.1 (*Types*). Let $\mathbb{K}$ be 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 *array types* of the program. A *type* is an element of $\mathbb{T} = \{\mathtt{int}\} \cup \mathbb{K} \cup \mathbb{A}$. 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 $\mathbb{F}(\kappa) = \{\kappa'.f\!:\!\mathsf{t}' \mid \kappa \le \kappa'\}$ 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 that return void. An array type has the form $\mathsf{t}[\,]$, where t is the type of its elements.

The set of types are ordered by a partial order $\le$ that we define next.

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 $\mathsf{t} \le \mathsf{t}'$, 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
—$\mathsf{t} \in \mathbb{A}$ and $\mathsf{t}' = \mathtt{Object}$, or
—$\mathsf{t} = \mathsf{t}_1[\,]$, $\mathsf{t}' = \mathsf{t}'_1[\,] \in \mathbb{A}$, and $\mathsf{t}_1 \le \mathsf{t}'_1$.

In the following, we show some interesting properties of the subtype relation $\le$. First of all, we show that two supertypes of the same type must be related through $\le$.

LEMMA 3.3. *Consider a type* $\mathsf{t} \in \mathbb{T}$ *and let* t' *and* t'' *be two supertypes of* t*, that is,* $\mathsf{t} \le \mathsf{t}'$ *and* $\mathsf{t} \le \mathsf{t}''$. *Then* $\mathsf{t}' \le \mathsf{t}''$ *or* $\mathsf{t}'' \le \mathsf{t}'$.

PROOF. We proceed by induction on the maximal number of array dimensions allowed for t' and t''. For the base case, t' and t'' are not arrays. In this case, if $\mathsf{t}' = \mathsf{t}''$, the thesis follows trivially. Assume hence that $\mathsf{t}' \ne \mathsf{t}''$. We distinguish the following cases.

—If $\mathsf{t} = \mathtt{int}$, then it must be $\mathsf{t}' = \mathsf{t}'' = \mathtt{int}$, which is impossible.
—If $\mathsf{t}' = \mathtt{Object}$, then $\mathsf{t} \ne \mathtt{int}$, and hence t'' is a reference type; then $\mathsf{t}'' \le \mathsf{t}'$.
—If $\mathsf{t}'' = \mathtt{Object}$, then $\mathsf{t} \ne \mathtt{int}$, and hence t' is a reference type; then $\mathsf{t}' \le \mathsf{t}''$.
—If t' and t'' are classes distinct from $\mathtt{Object}$, then t also must be a class, and 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 $\mathsf{t}' \le \mathsf{t}''$; in the former case $\mathsf{t}'' \le \mathsf{t}'$.

For the inductive case, we also have to consider the following cases.

—If t' is an array and t'' is not $\mathtt{Object}$, then by Definition 3.2, $\mathsf{t} = \mathsf{t}_1[\,]$, $\mathsf{t}' = \mathsf{t}'_1[\,]$, and $\mathsf{t}'' = \mathsf{t}''_1[\,]$ for some types $\mathsf{t}_1, \mathsf{t}'_1, \mathsf{t}''_1$ and $\mathsf{t}'_1 \ne \mathsf{t}''_1$, since otherwise $\mathsf{t}' = \mathsf{t}''$. Since $\mathsf{t} \le \mathsf{t}'$ and $\mathsf{t} \le \mathsf{t}''$ we have, by Definition 3.2, that $\mathsf{t}_1 \le \mathsf{t}'_1$ and $\mathsf{t}_1 \le \mathsf{t}''_1$. By inductive hypothesis, we have $\mathsf{t}'_1 \le \mathsf{t}''_1$ or $\mathsf{t}''_1 \le \mathsf{t}'_1$. Again by Definition 3.2, we obtain that $\mathsf{t}' \le \mathsf{t}''$ or $\mathsf{t}'' \le \mathsf{t}'$ holds.
—If t'' is an array and t' is not $\mathtt{Object}$, the thesis follows symmetrically to the preceding case. □

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:

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

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

LEMMA 3.5. *Let* $\mathsf{t}, \mathsf{t}', \mathsf{t}'' \in \mathbb{T}$ *with* $\mathsf{t}' \le \mathsf{t}''$. *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.

—$\mathsf{t} \le \mathsf{t}'$. Hence $\mathsf{t} \le \mathsf{t}''$ and $\mathsf{t}'' \in \mathsf{compatible}(\mathsf{t})$.
—$\mathsf{t}' \le \mathsf{t}$. Since $\mathsf{t}' \le \mathsf{t}''$, by Lemma 3.3 we have $\mathsf{t} \le \mathsf{t}''$ or $\mathsf{t}'' \le \mathsf{t}$, that is, $\mathsf{t}'' \in \mathsf{compatible}(\mathsf{t})$. □

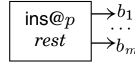
We show that the function compatible is monotonic.

LEMMA 3.6. *Let* $\mathsf{t}', \mathsf{t}'' \in \mathbb{T}$ *with* $\mathsf{t}' \le \mathsf{t}''$. *Then* $\mathsf{compatible}(\mathsf{t}') \subseteq \mathsf{compatible}(\mathsf{t}'')$.

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

—$\mathsf{t} \le \mathsf{t}'$. *Hence* $\mathsf{t} \le \mathsf{t}''$ *and* $\mathsf{t} \in \mathsf{compatible}(\mathsf{t}'')$.
—$\mathsf{t}' \le \mathsf{t}$. *Since* $\mathsf{t}' \le \mathsf{t}''$, *by Lemma 3.3 we have* $\mathsf{t} \le \mathsf{t}''$ *or* $\mathsf{t}'' \le \mathsf{t}$, *that is,* $\mathsf{t} \in \mathsf{compatible}(\mathsf{t}'')$. □

We analyze bytecode instructions preprocessed into a control-flow graph (CFG), that is, a directed graph of *basic blocks*, with no jumps inside the blocks. We graphically write



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

Example 3.7. Figure 2 shows the basic blocks of the second constructor in Figure 1 (Lines 9–12). There is a branch at the implicit call to the constructor of `java.lang.Object` (automatically added by the compiler) that might throw an exception (as 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 two blocks storing the formal parameters (locals 1 and 2) into the fields of `this` (held in local variable 0) and then returns. 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 successor, unless when throw raises an exception that is caught inside the same method.

An exception handler starts with a catch bytecode. A virtual method call (i.e., the typical object-oriented method call, where the method signature is identified at compile time but its implementation is only resolved dynamically at runtime) or a selection of an exception handler is translated into a block linked to many subsequent blocks. Each of these subsequent blocks starts with a *filtering* bytecode, such as exception_is K for exceptional handlers.

Bytecode instructions operate on *variables*, which encompass both *stack elements* allocated in the *operand stack* ($S = \{s_0, \ldots\}$) and *local variables* allocated in the *array of local variables* ($L = \{l_0, \ldots\}$). At any point of execution, we know the exact length of both the array of local variables and the operand stack. Moreover, a standard algorithm [Lindholm and Yellin 1999] infers their static types. These static types are returned by the *type environment* map.

```
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 void main(String[] args) {
15     ...
16     int n = Integer.valueOf(args[0]);
17     ...
18     List list = new List();
19     for (int i = 1; i <= n; i++) {
20       Object o = new Object();
21       List tmp = new List(o, list.tail);
22       list.tail = tmp;
23     }
24   }
25 }
```

Fig. 1. Our running example.

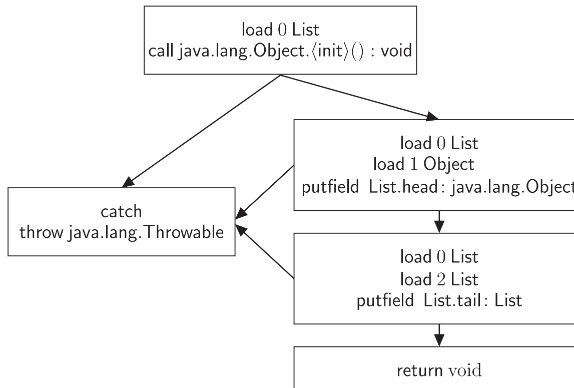


Fig. 2. Our representation of the code of the second constructor from Figure 1.

Definition 3.8 (*Type Environment*). Each program point is enriched with a *type environment* τ, that is, 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 *stack elements* $S = \{s_0, \ldots\}$, that is, $\mathsf{dom}(\tau) = L \cup S$.

Type environments specify the variables in scope at a given program point. Hence, they do not provide static type information for the fields of the objects in memory. This is because variables change number and type from a program point to another, while the fields of the objects have fixed, static types specified by the definition of the class where they are declared, as we will formalize in Definition 3.10.

3.2. States

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 JVM [Lindholm and Yellin 1999].

Definition 3.9 (*Values*). The set of all possible values that our formalization supports is $\mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$, 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 article) and where $\mathbb{L}$ is an infinite set of *memory locations*.

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

Definition 3.10 (*Object Representation*). Given an object o, its type is maintained inside o in a special field $o.\texttt{type}$, and we say that o is an *instance* of $o.\texttt{type}$. Each object o contains its *internal environment* $o.\phi$ that maps every field $\kappa'.f\!:\!\mathsf{t}' \in \mathbb{F}(o.\texttt{type})$ into its value as provided in the object, denoted by $(o.\phi)(\kappa'.f\!:\!\mathsf{t}')$. Hence, the domain of $o.\phi$ is $\mathsf{dom}(o.\phi) = \mathbb{F}(o.\texttt{type})$, and its range $\mathsf{rng}(o.\phi)$ is the set of the values of the fields of o.

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

Definition 3.11 (*Array Representation*). Given an array a, its type is maintained inside a in a special field $a.\texttt{type}$, and we say that a is an *instance* of $a.\texttt{type}$. The length of a is kept inside a special field $a.\texttt{length}$. Each array a contains an *internal environment* $a.\phi$ that maps each index $0 \leq i < a.\texttt{length}$ into the value $(a.\phi)(i)$ of the element at that index. Hence, the domain of $a.\phi$ is $\mathsf{dom}(a.\phi) = \{0, \ldots, a.\texttt{length} - 1\}$, and its range $\mathsf{rng}(a.\phi)$ is the set of the elements of a.

We want to analyze the possible *states* of the JVM at each point of the program under analysis.

Definition 3.12 (*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, one for each local variable of $\mathsf{dom}(\tau)$, s is a stack of values, one for each stack element in $\mathsf{dom}(\tau)$, which grows leftwards, and μ is a *memory* that binds locations to *objects* and *arrays*. 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, that is, that states are well-typed.

Definition 3.13 (*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 $\mathsf{t} = \texttt{int}$, or
—$v = \texttt{null}$ and $\mathsf{t} \in \mathbb{K} \cup \mathbb{A}$, or
—$v \in \mathbb{L}$, $\mathsf{t} \in \mathbb{K} \cup \mathbb{A}$ 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 and every field $\kappa'.f : \mathsf{t}' \in \mathbb{F}(o.\texttt{type})$ available in that object, the value held 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 the memory, such as $a.\texttt{type} = \mathsf{t}'[\,]$, the values in $\mathsf{rng}(a.\phi)$ are consistent with t'.

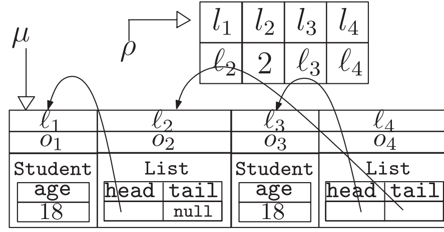


Fig. 3. A JVM state $\sigma = \langle \rho, \mu \rangle$.

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 one variable holding a location bound to the thrown exception object. When we denote a state by σ, we do not specify if 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 3.14 (*Java Virtual Machine State*). The set of *Java virtual machine states* (from now on just *states*) in a 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`, that is, $\tau'(s_0) \leq$ `Throwable`.

Example 3.15. Let `Student` be a class containing one instance field `age` of type `int`. Consider the following type environment.

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

where `List` is the class defined in Figure 1. In Figure 3, we show a state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$. The environment ρ maps local variables l_1, l_2, l_3, and l_4 to values $\ell_2 \in \mathbb{L}$, $2 \in \mathbb{Z}$, $\ell_3 \in \mathbb{L}$, and $\ell_4 \in \mathbb{L}$, respectively. The memory μ maps locations ℓ_2 and ℓ_4 to objects o_2 and o_4 of class `List`, and locations ℓ_1 and ℓ_3 to the objects o_1 and o_3 of class `Student`. Objects are represented as boxes with a class tag and an internal environment mapping fields to values. For instance, fields `head` and `tail` of object o_4 contain locations ℓ_3 and ℓ_2, respectively.

3.3. 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 the local variables and of the variables on the operand stack at each program point are statically known and specified by τ [Lindholm and Yellin 1999]. We assume that we are analyzing a type-checked program, so that, for instance, field and method resolution always succeeds. 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 local variables and stack elements, and let i and j be the cardinalities of these sets. Moreover, we suppose that the semantics is undefined for input states of wrong sizes or types, as is required in Lindholm and Yellin [1999]. Figure 4 defines the semantics of bytecode instructions. We discuss it next.

Load and Store Instructions. The load and store instructions transfer values between the local variables and the operand stack of a state: $\mathsf{load}\ k\ \mathsf{t}$ loads the local variable l_k whose static type is t onto the operand stack; $\mathsf{store}\ k\ \mathsf{t}$ stores the topmost value of the

$$\mathit{const}\ x = \lambda\langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle, \mu\rangle \,.\, \langle\langle \mathsf{l} \,\|\, x :: \mathsf{s}\rangle, \mu\rangle$$
$$\mathit{load}\ k\ \mathsf{t} = \lambda\langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle, \mu\rangle \,.\, \langle\langle \mathsf{l} \,\|\, \mathsf{l}[k] :: \mathsf{s}\rangle, \mu\rangle$$
$$\mathit{store}\ k\ \mathsf{t} = \lambda\langle\langle \mathsf{l} \,\|\, t :: \mathsf{s}\rangle, \mu\rangle \,.\, \langle\langle \mathsf{l}[k \mapsto t] \,\|\, \mathsf{s}\rangle, \mu\rangle$$

$$\mathit{add} = \lambda\langle\langle \mathsf{l} \,\|\, t_1 :: t_2 :: \mathsf{s}\rangle, \mu\rangle \,.\, \langle\langle \mathsf{l} \,\|\, t_1 + t_2 :: \mathsf{s}\rangle, \mu\rangle$$
$$\mathit{sub} = \lambda\langle\langle \mathsf{l} \,\|\, t_1 :: t_2 :: \mathsf{s}\rangle, \mu\rangle \,.\, \langle\langle \mathsf{l} \,\|\, t_2 - t_1 :: \mathsf{s}\rangle, \mu\rangle$$
$$\mathit{mul} = \lambda\langle\langle \mathsf{l} \,\|\, t_1 :: t_2 :: \mathsf{s}\rangle, \mu\rangle \,.\, \langle\langle \mathsf{l} \,\|\, t_1 * t_2 :: \mathsf{s}\rangle, \mu\rangle$$
$$\mathit{div} = \lambda\langle\langle \mathsf{l} \,\|\, t_1 :: t_2 :: \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \,\|\, t_2/t_1 :: \mathsf{s}\rangle, \mu\rangle & \text{if } t_1 \neq 0 \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{ae}]\rangle} & \text{otherwise} \end{cases}$$
$$\mathit{rem} = \lambda\langle\langle \mathsf{l} \,\|\, t_1 :: t_2 :: \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \,\|\, t_2 \% t_1 :: \mathsf{s}\rangle, \mu\rangle & \text{if } t_1 \neq 0 \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{ae}]\rangle} & \text{otherwise} \end{cases}$$
$$\mathit{inc}\ k\ x = \lambda\langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle, \mu\rangle \,.\, \langle\langle \mathsf{l}[k \mapsto \mathsf{l}[k] + x] \,\|\, \mathsf{s}\rangle, \mu\rangle$$

$$\mathit{new}\ \kappa = \lambda\langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \,\|\, \ell :: \mathsf{s}\rangle, \mu[\ell \mapsto o]\rangle & \text{if enough memory} \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{oome}]\rangle} & \text{otherwise} \end{cases}$$
$$\mathit{getfield}\ \kappa.f\!:\!\mathsf{t} = \lambda\langle\langle \mathsf{l} \,\|\, r :: \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \,\|\, (\mu(r).\phi)(f) :: \mathsf{s}\rangle, \mu\rangle & \text{if } r \neq \mathtt{null} \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases}$$
$$\mathit{putfield}\ \kappa.f\!:\!\mathsf{t} = \lambda\langle\langle \mathsf{l} \,\|\, t :: r :: \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle, \mu[(\mu(r).\phi)(f) \mapsto t]\rangle & \text{if } r \neq \mathtt{null} \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases}$$

$$\mathit{arraynew}\ \alpha = \lambda\langle\langle \mathsf{l} \,\|\, n :: \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{nase}]\rangle} & \text{if } n < 0 \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{oome}]\rangle} & \text{otherwise if not enough memory} \\ \langle\langle \mathsf{l} \,\|\, \ell :: \mathsf{s}\rangle, \mu[\ell \mapsto a]\rangle & \text{otherwise} \end{cases}$$
$$\mathit{arraylength}\ \alpha = \lambda\langle\langle \mathsf{l} \,\|\, r :: \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \,\|\, \mu(r).\mathtt{length} :: \mathsf{s}\rangle, \mu\rangle & \text{if } r \neq \mathtt{null} \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases}$$
$$\mathit{arrayload}\ \alpha = \lambda\langle\langle \mathsf{l} \,\|\, k :: r :: \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{if } r = \mathtt{null} \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{obe}]\rangle} & \text{otherwise if } k < 0 \text{ or } k \geq \mu(r).\mathtt{length} \\ \langle\langle \mathsf{l} \,\|\, (\mu(r).\phi)(k) :: \mathsf{s}\rangle, \mu\rangle & \text{otherwise} \end{cases}$$
$$\mathit{arraystore}\ \alpha = \lambda\langle\langle \mathsf{l} \,\|\, v :: k :: r :: \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{if } r = \mathtt{null} \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{obe}]\rangle} & \text{otherwise if } k < 0 \text{ or } k \geq \mu(r).\mathtt{length} \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{ase}]\rangle} & \text{otherwise if } v \in \mathbb{L} \text{ and } \mu(v).\mathtt{type}[\,] \not\leq \mu(r).\mathtt{type} \\ \langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle, \mu[(\mu(r).\phi)(k) \mapsto v]\rangle & \text{otherwise} \end{cases}$$

$$\mathit{dup}\ \mathsf{t} = \lambda\langle\langle \mathsf{l} \,\|\, t :: \mathsf{s}\rangle, \mu\rangle \,.\, \langle\langle \mathsf{l} \,\|\, t :: t :: \mathsf{s}\rangle, \mu\rangle$$

$$\mathit{ifeq}\ \mathsf{t} = \lambda\langle\langle \mathsf{l} \,\|\, t :: \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle, \mu\rangle & \text{if } t \in \{0, \mathtt{null}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$
$$\mathit{ifne}\ \mathsf{t} = \lambda\langle\langle \mathsf{l} \,\|\, t :: \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle, \mu\rangle & \text{if } t \notin \{0, \mathtt{null}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathit{return}\ \mathsf{void} = \lambda\langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle, \mu\rangle \,.\, \langle\langle \mathsf{l} \,\|\, \epsilon\rangle, \mu\rangle$$
$$\mathit{return}\ \mathsf{t} = \lambda\langle\langle \mathsf{l} \,\|\, t :: \mathsf{s}\rangle, \mu\rangle \,.\, \langle\langle \mathsf{l} \,\|\, t\rangle, \mu\rangle, \quad \text{where } \mathsf{t} \neq \mathsf{void}$$

$$\mathit{throw}\ \kappa = \lambda\langle\langle \mathsf{l} \,\|\, t :: \mathsf{s}\rangle, \mu\rangle \,.\, \begin{cases} \underline{\langle\langle \mathsf{l} \,\|\, t\rangle, \mu\rangle} & \text{if } t \neq \mathtt{null} \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases}$$
$$\mathit{catch} = \lambda\underline{\langle\langle \mathsf{l} \,\|\, t\rangle, \mu\rangle} \,.\, \langle\langle \mathsf{l} \,\|\, t\rangle, \mu\rangle$$
$$\mathit{exception_is}\ K = \lambda\langle\langle l \,\|\, t\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle l \,\|\, t\rangle, \mu\rangle & \text{if } t \in \mathbb{L} \text{ and } \mu(t).\mathtt{type} \in K \\ \text{undefined} & \text{otherwise} \end{cases}$$

Fig. 4. The semantics of the bytecode instructions maps states to states. $\ell \in \mathbb{L}$ is a fresh location, o, and a are, respectively, a new object of class κ and a new array of type α. Exceptions ae, $oome$, npe, $nase$, obe and ase are, respectively, new instances of the following: `ArithmeticException`, `OutOfMemoryError`, `NullPointerException`, `NegativeArraySizeException`, `ArrayIndexOutOfBoundsException`, and `ArrayStoreException`.

operand stack, whose static type is t, into the local variable l_k; const v loads an integer constant or `null` onto the operand stack.

Arithmetic Instructions. The arithmetic bytecode instructions pop the topmost two 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 (addition), sub (subtraction), mul (multiplication), div (division), rem (remainder). There is also inc k x, that increments the value of l_k by x.

Object Creation and Manipulation Instructions. These bytecode instructions create or access objects in memory: new κ creates a new instance of class κ; getfield $\kappa.f$:t reads a value from the field f belonging to the class κ and whose static type is t; putfield $\kappa.f$:t writes a value into the field f belonging to the class κ and whose static type is t.

Array Creation and Manipulation Instructions. These bytecode instructions create or access arrays: arraynew α creates a new array of type α whose length is the value popped from the operand stack and puts a reference to this new array onto the top of the operand stack arraylength α pops the topmost value from the operand stack, that must have a type compatible with α, and pushes back onto the operand stack the length of the corresponding array; arrayload α pops from the operand stack an integer value k and a reference to an array of type α and puts back onto the operand stack its kth element; arraystore α pops from the operand stack a value of type t, an integer k, and a reference to an array of type α = t[ ] and writes the value into the kth element of the array.

Operand Stack Management Instructions. The only operand stack management instruction supported by our formalization is dup t, that duplicates the topmost value of the operand stack.

Control Transfer Instructions. In our formalization, conditional bytecodes are used in complementary pairs (such as ifne t and ifeq t) at the beginning of the two conditional branches. The semantics of a conditional bytecode is undefined when its condition is false. For instance, ifeq t checks whether the top of the stack, of type t, is 0 when t = `int`, or is `null` otherwise; the *undefined* case means that the JVM does not continue the execution of that branch of code if the condition is false.

Exception Handling Instructions. An exception is thrown programmatically by using the throw κ bytecode instruction. Exceptions can also be thrown by various other bytecode instructions if they detect an abnormal condition. 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 runtime class of the top of the stack: it filters those states whose top of the stack is an instance of a class in $K \subseteq \mathbb{K}$.

Method Calls and Return. When a caller transfers control to a callee $\kappa.m(\vec{\mathsf{t}})$: t, the JVM runs an operation *makescope* $\kappa.m(\vec{\mathsf{t}})$:t that copies the topmost stack elements, holding the actual arguments of the call, to local variables that correspond to the formal parameters of the callee, and clears the stack. We only consider instance methods, where `this` is a special argument held in local variable l_0 of the callee. More precisely, the ith local variable of the callee is a copy of the $(\pi - 1) - i$th topmost element of the operand stack of the caller.

Definition 3.16 (*Makescope*). Let $\kappa.m(\vec{\mathsf{t}})$: t be a method and π the number of stack elements holding its actual parameters, including the implicit parameter `this`. We

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

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

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

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

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

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

Fig. 5. The transition rules of our semantics.

define a function $(\mathit{makescope}\ \kappa.m(\vec{\mathsf{t}}){:}\mathsf{t}) : \Sigma \to \Sigma$ as

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

provided $\mathit{rec} \neq \mathtt{null}$ and the lookup of $m(\vec{\mathsf{t}}){:}\mathsf{t}$ from $\mu(\mathit{rec}).\mathtt{type}$ leads to $\kappa.m(\vec{\mathsf{t}}){:}\mathsf{t}$. We let it be undefined otherwise.

Bytecode call $\kappa_1.m \ldots \kappa_n.m$ calls, nondeterministically, one of the callees in the enumeration. These are possible targets of a virtual call, since a method call in Java bytecode, which is an object-oriented language, can in general lead to many method implementations. The overapproximation of the possible targets is not explicit in actual Java bytecode, but we assume that it is provided in our simplified bytecode. In particular, that overapproximation can be computed by any *class analysis* [Palsberg and Schwartzbach 1991]. The exact implementation of the method is later selected through a *makescope* instruction, as we will show with Figure 5. Bytecode return t terminates a method and clears its operand stack, leaving only the returned value when $t \neq$ void. This is later moved on top of the stack of the caller of the callee, as Figure 5 shows.

3.4. 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 3.17 (*Configuration*). A *configuration* is a pair $\langle b \parallel \sigma$ of a block b and a state σ representing the fact that the JVM is about to execute b in state σ. An *activation stack* is a stack $c_1 :: c_2 :: \cdots :: c_n$ 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.

Definition 3.18 (*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 Figure 5.

Rule (1) runs the first instruction ins of a block, different from call, by using its semantics *ins* given in Figure 4. 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, and Rule (3) creates a new state whose operand stack contains only a reference to a `NullPointerException`. Instead, 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 runtime target methods. The dynamic semantics of call implements the virtual method resolution of object-oriented languages by looking for the implementation $\kappa_i.m(\vec{\mathsf{t}}) : \mathsf{t}$ of the callee, that is, executed through the dynamic lookup rules of the language, codified inside the *makescope* function. The latter is only defined when that implementation is selected at runtime; in that case, *makescope* yields the initial state σ' that the semantics uses to create a new current configuration containing the first block of the selected implementation and σ'. It pops the actual arguments from the previous configuration and the call from the instructions to be executed at return time. Although this rule seems nondeterministic, only one thread of execution continues, since we assume that the lookup 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 Figure 5 used; for instance, we write $\stackrel{(1)}{\Rightarrow}$ for a derivation step through Rule (1).

4. REACHABILITY

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 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 4.1 (*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

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

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

Fig. 6. Example of computation of reachable locations and types.

$\mathsf{L}^i_\mu(\ell)$ 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 $\mathsf{rng}(\mu(\ell_1).\phi) \cap \mathbb{L}$, that is, the locations held in $\mu(\ell_1)$'s fields or elements, are reachable from ℓ. We say that a variable a reaches a location ℓ if a holds a location that reaches ℓ.

Definition 4.2 (*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 if the former is bound to a location reachable from the latter.

Definition 4.3 (*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 $a \rightsquigarrow^\sigma b$, if and only if $\rho(b) \in \mathsf{L}_\sigma(a)$.

Remark 4.4. It is worth noting that two variables a and b share in a state σ if and only if $\mathsf{L}_\sigma(a) \cap \mathsf{L}_\sigma(b) \neq \varnothing$. As a consequence, if a reaches b or b reaches a, then a and b share. However, it is possible that a and b share and yet neither a reaches b nor b reaches a. For that, it is enough that a and b are bound to overlapping data structures such that none of them is included in the other. It follows that we cannot reconstruct a sound nontrivial sharing analysis from the results of a reachability analysis.

Example 4.5. Consider the state $\sigma \in \Sigma_\tau$ from Example 3.15. On the left-hand side of Figure 6 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 $l_1 \rightsquigarrow^\sigma l_1$, $l_1 \rightsquigarrow^\sigma l_2$, $l_3 \rightsquigarrow^\sigma l_3$, $l_4 \rightsquigarrow^\sigma l_1$, $l_4 \rightsquigarrow^\sigma l_2$, $l_4 \rightsquigarrow^\sigma l_3$, $l_4 \rightsquigarrow^\sigma l_4$.

Let us show a very important and useful result: if two states provide the same value for variables a and a' and for variables b and b', and if the two memories provide the same values for the locations reachable from a in the first state, then reachability from a to b coincides with reachability from a' to b'.

LEMMA 4.6. *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 the following hold.*

(*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 $\rho'(b') \in \mathsf{L}_{\sigma'}(a')$, that is, $a \rightsquigarrow^{\sigma} b$ 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 4.2), 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

$$\begin{aligned}
\mathsf{L}^i_\mu(\rho(a)) &= \mathsf{L}^{i-1}_\mu(\rho(a)) \cup \bigcup_{\ell \in \mathsf{L}^{i-1}_\mu(\rho(a))} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) && \text{[By Definition 4.1]}\\
&= \mathsf{L}^{i-1}_{\mu'}(\rho'(a')) \cup \bigcup_{\ell \in \mathsf{L}^{i-1}_{\mu'}(\rho'(a'))} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) && \text{[By hypothesis]}\\
&= \mathsf{L}^{i-1}_{\mu'}(\rho'(a)) \cup \bigcup_{\ell \in \mathsf{L}^{i-1}_{\mu}(\rho'(a))} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) && \text{[By points (3) and (4)]}\\
&= \mathsf{L}^i_{\mu'}(\rho'(a')). && \text{[By Definition 4.1]}
\end{aligned}$$

□

Note that Points (3) and (4) of Lemma 4.6 hold, in particular when $\mu = \mu'$, but the more general form of the lemma is more useful.

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.[4] 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 later exploit this observation for the definition of the abstract semantics to provide a sound approximation of the side-effects of the execution of a method and of the reachability for its return value.

PROPOSITION 4.7. *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* $\mathsf{ins} = \mathsf{call}\ m_1 \ldots m_n$ *and* $\mathsf{ins}' \neq \mathsf{catch}$ *are executed. Namely,* σ' *is a nonexceptional state obtained at the end of execution of a callee* m_w *in* σ*, for a* $w \in [1..n]$*, the topmost* π *stack elements of* σ *(values* $\mathsf{s}[|\mathsf{s}| - 1], \ldots, \mathsf{s}[|\mathsf{s}| - \pi]$*), and the topmost operand stack element of* σ' *(value* $\mathsf{s}'[|\mathsf{s}'| - 1]$*) are 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*.*

$$\mathcal{L}_\sigma = \mathsf{dom}(\mu) \smallsetminus \bigcup_{|\mathsf{s}|-\pi \leq i \leq |\mathsf{s}|-1} \mathsf{L}_\sigma(\mathsf{s}[i]).$$

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

Then, the following conditions hold:

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

Let us explain these three points in more detail.

(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 bound to ℓ at ins′ ($\mu'(\ell)$), that is, the execution of the method does not modify the values bound to the locations in $\mathcal{L}_\sigma$.
(2) The method's return value $\mathsf{s}'[|\mathsf{s}'|-1]$ at ins′ is not a location in $\mathcal{L}_\sigma$. That value might actually 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 not reachable from the actual arguments of the method at ins), is such that its content (an object or array) is not modified during the execution of the activation of the method, that is, its fields (if $\mu(\ell)$ is an object) or elements (if $\mu(\ell)$ is an array) are left unchanged by that execution.

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 4.10) that this is a weaker, conservative approximation of the dynamic notion of reachability of Definition 4.3: 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 4.8 (*Reachability between Types*). Let $\mathsf{t} \in \mathbb{T}$. The *set of types reachable from* t is $\mathsf{T}(\mathsf{t}) = \bigcup_{i \geq 0} \mathsf{T}^i(\mathsf{t})$, where $\mathsf{T}^i(\mathsf{t})$ are the types reachable from t in at most i steps.

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

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

Example 4.9. Consider class `List` from Figure 1 and suppose that class `Student` contains only one field, of type `int`, as stated in Example 3.15. Both `List` and, `Student` are subclasses of `Object`. On the right of Figure 6, we report the types reachable from these classes. For instance, `List`⇝`Student`, `Object`⇝`Student`, `Student`⇝`Object`, `Object`⇝`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, then the static type of the former is reachable from the static type of the latter.

LEMMA 4.10. *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 4.3, 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 $\mu(\ell).\mathtt{type} \in \mathsf{T}^j(\tau(a))$. 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 $\mu(\rho(b)).\mathtt{type} \in \mathsf{T}^j(\tau(a)) \subseteq \mathsf{T}(\tau(a))$, that is, $\tau(a) \rightsquigarrow \mu(\rho(b)).\mathtt{type}$. Since (Definition 3.12) $\mu(\rho(b)).\mathtt{type} \leq \tau(b)$, by Lemma A.6 in Appendix A, 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 3.12, $\mu(\rho(a)).\mathtt{type} \leq \tau(a)$, that is, $\mu(\rho(a)).\mathtt{type} \in \mathsf{compatible}(\tau(a)) = \mathsf{T}^0(\tau(a))$. Since $j = 0 \leq 0 = i$, $P(0)$ holds.

Inductive Step. Suppose that for every $k < i$, $P(k)$ holds and consider a location $\ell \in \mathsf{L}^i_{\mu}(\rho(a))$. By Definition 4.2 we have two cases.

- If $\ell \in \mathsf{L}^{i-1}_{\mu}(\rho(a))$, then by the inductive hypothesis $P(i-1)$, we know that there exists $0 \leq j \leq i-1 < i$ such that $\mu(\ell).\mathtt{type} \in \mathsf{T}^j(\tau(a))$. Hence $P(i)$ 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 $\mu(\ell').\mathtt{type} \in \mathbb{K}$, then there exists $\kappa'.f{:}\mathsf{t}' \in \mathbb{F}(\mu(\ell').\mathtt{type})$ such that $\ell = (\mu(\ell').\phi)(\kappa'.f{:}\mathsf{t}')$ and $\mu(\ell).\mathtt{type} \leq \mathsf{t}'$. Hence, $\mu(\ell).\mathtt{type} \in \mathsf{compatible}(\mathsf{t}') \subseteq \mathsf{T}^{j+1}(\tau(a))$.
  - If $\mu(\ell').\mathtt{type} \in \mathbb{A}$, then there exists $\kappa'.f{:}\mathsf{t}' \in \mathbb{F}(\mu(\ell').\mathtt{type})$ such that $\ell = (\mu(\ell').\phi)(\kappa'.f{:}\mathsf{t}')$ and $\mu(\ell).\mathtt{type} \leq \mathsf{t}'$. Hence, $\mu(\ell).\mathtt{type} \in \mathsf{compatible}(\mathsf{t}') \subseteq \mathsf{T}^{j+1}(\tau(a))$.

  Since $0 \leq j+1 \leq i$, in both cases, $P(i)$ holds as well. □

Example 4.11. Since $l_4 \rightsquigarrow^{\sigma} l_3$ (Example 4.5), by Lemma 4.10, we conclude that $\tau(l_4) \rightsquigarrow \tau(l_3)$. In fact, Example 4.9 shows that $\tau(l_4) = \mathtt{List} \rightsquigarrow \mathtt{Student} = \tau(l_3)$.

5. REACHABILITY ANALYSIS

In this section, we define an abstract interpretation Cousot and Cousot [1977, 1979] of our concrete semantics of Section 3 with respect to the property of reachability between variables (Definition 4.3). This will be an actual static analysis algorithm for interprocedural, whole-program reachability analysis.

Our choice has been to start from the simplest possible domain for reachability that coincides with the property of reachability between pairs of variables itself, and build a framework where this property is propagated between program points until stabilization. Propagation at a program point depends on the bytecode that occurs there. In most cases, the propagation rules are straightforward, for instance, for those bytecodes that only move values between stack elements or from stack elements to local variables and vice versa. The complex scenarios are those related to field and array accesses (reading or writing) and method calls. Field accesses perform an interaction between the stack variables and the heap memory, which requires the reconstruction of the reachability information after the operation by taking into account properties of the heap memory, such as aliasing, sharing, or reachability itself. Method calls may modify the variables of the caller by side-effect, and the exact sound reconstruction of those side-effects requires, again, information on aliasing and sharing, as well as the exploitation of the reachability information at the end of the callee, wherever it does not reassign its formal arguments. For these reasons, those operations are the actual kernel of our abstract interpretation. Their definition is consequently relatively

complex. We provide six examples in this section that show how their propagation rules work on a concrete example of analysis where a field is modified and a method is called.

5.1. Concrete and Abstract Domains

The concrete semantics works over concrete states (Definition 3.14) that our abstract interpretation abstracts into sets of ordered pairs of variables.

Definition 5.1 (*Concrete and Abstract Domain*). Given a type environment $\tau \in \mathcal{T}$, we define the *concrete domain* over τ as $\mathsf{C}_\tau = \langle \wp(\Sigma_\tau), \subseteq$ and the *abstract domain* over τ as the powerset of the set of ordered pairs of variables $\mathsf{A}_\tau = \langle \wp(\mathsf{dom}(\tau) \times \mathsf{dom}(\tau)), \subseteq$. For every $v, w \in \mathsf{dom}(\tau)$, we write $v \rightsquigarrow w$ for $\langle v, w \rangle$.

An abstract domain element $R \in \mathsf{A}_\tau$ represents those concrete states in Σ_τ whose reachability information is conservatively overapproximated by the pairs of variables in R. By requiring an overapproximation, we induce a possible reachability analysis.

Definition 5.2 (*Concretization Map*). For every type environment $\tau \in \mathcal{T}$, we define the *concretization map* $\gamma_\tau : \mathsf{A}_\tau \rightarrow \mathsf{C}_\tau$ as.

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

Both C_τ and A_τ are complete lattices. Moreover, the following lemma states that γ_τ is co-additive, and therefore it is the concretization map of a Galois connection [Cousot and Cousot 1977]. Its proof can be found in the Appendix. Thus, A_τ is actually an abstract domain in the sense of abstract interpretation.

LEMMA 5.3. *Let $\tau \in \mathcal{T}$. The function γ_τ is co-additive.*

Lemma 4.10 allows us to refine the abstract domain by only considering pairs of variables whose static types allow their reachability. A similar idea is used in Genaim and Zanardini [2010], where, however, type reachability is used in the same definition of the abstract domain but no formal proof of a relationship between variable and type reachability is provided. We prefer to use this relationship in the abstract semantics that we define in the next section in order to avoid the introduction of spurious pairs of unreachable variables.

5.2. The Abstract Constraint Graph

Our analysis is constraint based in the sense that it builds an *abstract constraint graph* from the program under analysis by creating a node of the graph for each bytecode instruction b of the program. This node will be later decorated with an element of A_τ, where τ is the static type information at the beginning of b. Arcs of this graph propagate abstract domain elements, reflecting, in abstract terms, the effects of the concrete semantics (Section 3) over the reachability information. In other words, an arc from the node for the bytecode instruction b_1 to that for the bytecode instruction b_2 propagates the reachability information at b_1 into that at b_2. The exact meaning of *propagates* depends here on b_1, since each bytecode instruction has different effects on reachability.

In the following, we assume the presence of *possible sharing* and *definite aliasing* approximations. Namely, we suppose that, at every program point, there exists a set of (non-ordered) pairs of variables representing an overapproximation of the actual sharing information at that program point, and a set of (non-ordered) pairs of variables representing an underapproximation of the actual aliasing information at that point. Pairs of variables that do not belong to the former set definitely do not share at that point. Dually, pairs of variables that belong to the latter set are definitely aliased at that point, but other variables might be aliased as well. These pieces of information

can be computed statically, before our reachability analysis is performed, and our tool Julia is able to provide them [Secci and Spoto 2005; Nikolić and Spoto 2012b]. Our analysis works correctly also when such information is not available: we can always assume that at every program point, every variable might share with another, and no variable is definitely aliased to another. This would induce a less precise, yet still sound reachability analysis.

Definition 5.4 (*ACG*). Let P be the program under analysis (i.e., a control-flow graph of basic blocks for each method or constructor). The *abstract constraint graph* (ACG) of P is a directed graph $\langle V, E$ (nodes, arcs) where the following hold.

— V contains a node $\boxed{\mathsf{ins}}$ for every bytecode instruction ins of P.
— V contains nodes $\boxed{\mathsf{exit}@m}$ and $\boxed{\mathsf{exception}@m}$ for each method or constructor m in P that represent the normal and exceptional ends of m.
— E contains directed (multi-)arcs with one or two sources and always one sink.
— For every arc in E, there is a *propagation rule* that is, a function over A, from the reachability information at its source(s) to the reachability information at its sink.

The arcs in E are built from P as follows. We assume that τ and τ' are the static type information at and immediately after the execution of a bytecode instruction ins, respectively. Moreover, we assume that τ contains j stack elements and i local variables. In the following, we discuss different 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 an arc is built from $\boxed{\mathsf{ins}}$ to $\boxed{\mathsf{ins}'}$, with one of the propagation rules #1–#11 in Figure 7, based on the bytecode instruction ins itself.

Final Arcs. For each $\mathsf{return}\ \mathsf{t}$ and $\mathsf{throw}\ \kappa$ occurring in a method or in a constructor m of P, there are arcs from $\boxed{\mathsf{return}\ \mathsf{t}}$ to $\boxed{\mathsf{exit}@m}$ and from $\boxed{\mathsf{throw}\ \kappa}$ to $\boxed{\mathsf{exception}@m}$, respectively, with one of the propagation rules #12–#14 in Figure 7.

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 one of the propagation rules #15–#17 in Figure 7.

Parameter Passing Arcs. For each $\mathsf{ins}_c = \mathsf{call}\ m_1 \ldots m_k$ to a method with π parameters (including the implicit parameter `this`), we build an arc from $\boxed{\mathsf{ins}_c}$ to the node corresponding to the first bytecode instruction of m_w, with the propagation rule #18 in Figure 7, for each $1 \leq w \leq k$.

Return Value Arcs. For each $\mathsf{ins}_c = \mathsf{call}\ m_1 \ldots m_k$ to a method with π parameters (including the implicit parameter `this`) returning a value of type $\mathsf{t} \in \mathbb{K}$ and each subsequent bytecode instruction ins' distinct from catch, we build a multi-arc from $\boxed{\mathsf{ins}_c}$ and $\boxed{\mathsf{exit}@m_w}$ (two sources, in that order) to $\boxed{\mathsf{ins}'}$, with the propagation rule #19 defined in Figure 8, for each $1 \leq w \leq k$.

Side-Effects Arcs. For each $\mathsf{ins}_c = \mathsf{call}\ m_1 \ldots m_k$ to a method with π parameters (including the implicit parameter `this`) and each subsequent bytecode instruction ins', we build a multi-arc from $\boxed{\mathsf{ins}_c}$ and $\boxed{\mathsf{exit}@m_w}$ (two sources, in that order) to $\boxed{\mathsf{ins}'}$, where ins' is not a catch, or from $\boxed{\mathsf{ins}_c}$ and $\boxed{\mathsf{exception}@m_w}$ (two sources, in that order) to $\boxed{\mathsf{catch}}$, for each $1 \leq w \leq k$. Its propagation rule #20 is given in Figure 8, where $\max = j - \pi$ if ins' is not a catch and $\max = 0$ otherwise.

		ins	propagation rule
SEQUENTIAL	#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 $\kappa.f$:t	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \neq s_{j-1}\} \cup \{s_{j-1} \rightsquigarrow b \in R \mid \mathsf{t} \rightsquigarrow \tau(b)\} \cup$ $\{a \rightsquigarrow s_{j-1} \mid \tau(a) \rightsquigarrow \mathsf{t} \neq \mathtt{int} \wedge [a \text{ and } s_{j-1} \text{ might share at ins}]\}$
	#5	putfield $\kappa.f$:t	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_{j-1}, s_{j-2}\}\} \cup$ $\{a \rightsquigarrow b \mid a, b \notin \{s_{j-1}, s_{j-2}\} \wedge a \rightsquigarrow s_{j-2} \in R \wedge s_{j-1} \rightsquigarrow b \in R\}$
	#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[ ]	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_{j-1}, s_{j-2}\}\} \cup \{s_{j-2} \rightsquigarrow b \in R \mid \mathsf{t} \rightsquigarrow \tau(b)\} \cup$ $\{a \rightsquigarrow s_{j-2} \mid \tau(a) \rightsquigarrow \mathsf{t} \neq \mathtt{int} \wedge [a \text{ and } s_{j-2} \text{ might share at ins}]\}$
	#9	arraystore α	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_{j-1}, s_{j-2}, s_{j-3}\}\} \cup$ $\{a \rightsquigarrow b \mid a, b \notin \{s_{j-1}, s_{j-2}, s_{j-3}\} \wedge a \rightsquigarrow s_{j-3} \in R \wedge s_{j-1} \rightsquigarrow b \in R\}$
	#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 v, ifne t, ifeq t, add, sub, mul, div, rem, inc k x, catch, exception_is K	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$
FINAL	#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	$\lambda R.\{(a \rightsquigarrow b)[s_{j-1}/s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\} \cup \{s_0 \rightsquigarrow s_0\}$
EXCEPTIONAL	#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 $m_1 \ldots m_k$	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_0, \ldots, s_{j-1}\}\} \cup \{s_0 \rightsquigarrow s_0\}$ $\cup \{a \rightsquigarrow s_0 \mid a \in \{l_0, \ldots, l_{i-1}\} \wedge \tau(a) \rightsquigarrow \mathtt{Throwable}\}$ $\cup \{s_0 \rightsquigarrow a \mid a \in \{l_0, \ldots, l_{i-1}\} \wedge \mathtt{Throwable} \rightsquigarrow \tau(a)\}$
	#17	div, rem, new κ, getfield $\kappa.f$:t, putfield $\kappa.f$:t, arraynew α, arraylength α, arrayload α, arraystore α	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_0, \ldots, s_{j-1}\}\} \cup \{s_0 \rightsquigarrow s_0\}$
PAR	#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. 7. Propagation rules of simple arcs.

Definition 5.4 deserves some explanation. It specifies how the ACG is built from the program under analysis. For each bytecode instruction ins, there is a node $\boxed{\text{ins}}$ in the graph that will be later *decorated* by an element of our abstract domain (Definition 5.1), representing an overapproximation of the actual reachability information at that point. The rules introduced in Definition 5.4 explain how this approximation is propagated along the arcs of the ACG. In the following, we show an example illustrating the construction of an ACG (Example 5.5), and then we explain in more detail the preceding propagation rules defined.

Example 5.5. Figure 9 shows the ACG built for the constructor in Figure 2. It also shows, in grey, three nodes of a caller of this constructor (nodes A, B and C corresponding to line 21 in Figure 1) and two nodes of the callee of call `java.lang.Object.⟨init⟩()`: void, to exemplify the arcs related to method call and return. Arcs are decorated with the number of their associated propagation rule. Note that the graph for the whole program includes other nodes and arcs. Figure 9 only shows those relevant for our example.

In the following examples, 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 n, for each node n. We suppose that $i_A = 5$ and $j_A = 4$. Namely, there are five local variables

RETURN	#19	$\lambda R_1.\lambda R_2.\{s_{j-\pi} \rightsquigarrow s_{j-\pi} \mid s_0 \rightsquigarrow s_0 \in R_2\}$ $\cup \left\{ \begin{array}{c} a \rightsquigarrow s_{j-\pi} \in \\ (\mathsf{dom}(\tau') \setminus \{s_{j-\pi}\}) \times \{s_{j-\pi}\} \end{array} \;\middle\vert\; \begin{array}{l} 1.\ \tau'(a) \rightsquigarrow \mathsf{t} \wedge \\ 2.\ \exists j-\pi \le p < j \text{ such that } a \text{ might share with } s_p \\ \quad \text{at call } m_1 \dots m_k \;\wedge \\ 3.\ \text{if } a \text{ is definitely aliased to } s_p \text{ at call } m_1 \dots m_k \text{ and} \\ \quad \text{no store } l_{p-j+\pi} \text{ occurs in } m_w, \text{ then } l_{p-j+\pi} \rightsquigarrow s_0 \in R_2 \end{array} \right\}$ $\cup \left\{ \begin{array}{c} s_{j-\pi} \rightsquigarrow b \in \\ \{s_{j-\pi}\} \times (\mathsf{dom}(\tau') \setminus \{s_{j-\pi}\}) \end{array} \;\middle\vert\; \begin{array}{l} 1.\ \mathsf{t} \rightsquigarrow \tau'(b) \wedge \\ 2.\ \exists j-\pi \le p < j \text{ s.t. } s_p \rightsquigarrow b \in R_1 \wedge \\ 3.\ \text{if } b \text{ is definitely aliased to } s_p \text{ at call } m_1 \dots m_k \text{ and} \\ \quad \text{no store } l_{p-j+\pi} \text{ occurs in } m_w, \text{ then } s_0 \rightsquigarrow l_{p-j+\pi} \in R_2 \end{array} \right\}$
SIDE-EFFECTS	#20	$\lambda R_1.\lambda R_2.\{a \rightsquigarrow b \in R_1 \mid a, b \in \{l_0, \dots, l_{i-1}, s_0, \dots, s_{\max-1}\}\}$ $\cup \left\{ a \rightsquigarrow b \;\middle\vert\; \begin{array}{l} 1.\ a, b \in \{l_0, \dots, l_{i-1}, s_0, \dots, s_{\max-1}\} \wedge \\ 2.\ \tau'(a) \rightsquigarrow \tau'(b) \wedge \\ 3.\ \exists j-\pi \le p_a < j \text{ such that } a \text{ might share with } s_{p_a} \text{ at call } m_1 \dots m_k \wedge \\ 4.\ \exists j-\pi \le p_b < j \text{ such that } s_{p_b} \rightsquigarrow b \in R_1 \wedge \\ 5.\ \text{if } \exists j-\pi \le q_a < j \text{ such that } a \text{ is definitely aliased to } s_{q_a} \text{ at call } m_1 \dots m_k \text{ and} \\ \quad \text{if } \exists j-\pi \le q_b < j \text{ such that } b \text{ is definitely aliased to } s_{q_b} \text{ at call } m_1 \dots m_k \text{ and} \\ \quad \text{no store } l_{q_a-j+\pi} \text{ nor store } l_{q_b-j+\pi} \text{ occurs in } m_i, \text{ then } l_{q_a-j+\pi} \rightsquigarrow l_{q_b-j+\pi} \in R_2 \end{array} \right\}$

Fig. 8. Propagation rules of multi-arcs.

in scope at line 21 (`args` is unused and for simplicity we do not consider that variable) and four stack elements.

`this`	l_0	`new List`	s_0
`n`	l_1	copy of `new List`	s_1
`list`	l_2	copy of `o`	s_2
`i`	l_3	`list.tail`	s_3
`o`	l_4		

We assume that previous static analyses provided a correct possible sharing information at node A: $\mathsf{share}_A = \{\langle s_0, s_1\rangle, \langle l_4, s_2\rangle, \langle l_2, s_3\rangle\}$ (only these non-ordered pairs of distinct variables might possibly share) and a correct definite aliasing information at node A: $\mathsf{alias}_A = \{\langle s_0, s_1\rangle, \langle l_4, s_2\rangle\}$ (these non-ordered pairs of distinct 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:

$$R_A = \left\{ \begin{array}{l} l_0 \rightsquigarrow l_0,\ l_2 \rightsquigarrow l_2,\ l_4 \rightsquigarrow l_4,\ l_2 \rightsquigarrow s_3,\ l_4 \rightsquigarrow s_2,\ s_2 \rightsquigarrow l_4, \\ s_0 \rightsquigarrow s_0,\ s_0 \rightsquigarrow s_1,\ s_1 \rightsquigarrow s_0,\ s_1 \rightsquigarrow s_1,\ s_2 \rightsquigarrow s_2,\ s_3 \rightsquigarrow s_3 \end{array} \right\}. \tag{1}$$

At the program point corresponding to node A, constructor con = `List.`⟨init⟩(`Object`, `List`):void is invoked. The receiver of con, s_1, is definitely aliased to s_0, while its first actual argument s_2 is definitely aliased to l_4 (since $\langle s_0, s_1, \langle l_4, s_2 \in \mathsf{alias}_A)$. Since con creates a new object of class `List` and instantiates its fields `head` and `tail` to the values held in con's actual arguments s_2 and s_3, it is clear that, at the end of the constructor, s_0 (aliased to the newly created object) reaches l_4 (aliased to the variable whose value is written inside the field `head` of the newly created object), and we expect our reachability analysis to include the pair $s_0 \rightsquigarrow l_4$ in the approximation at node B. This will be confirmed in the following examples.

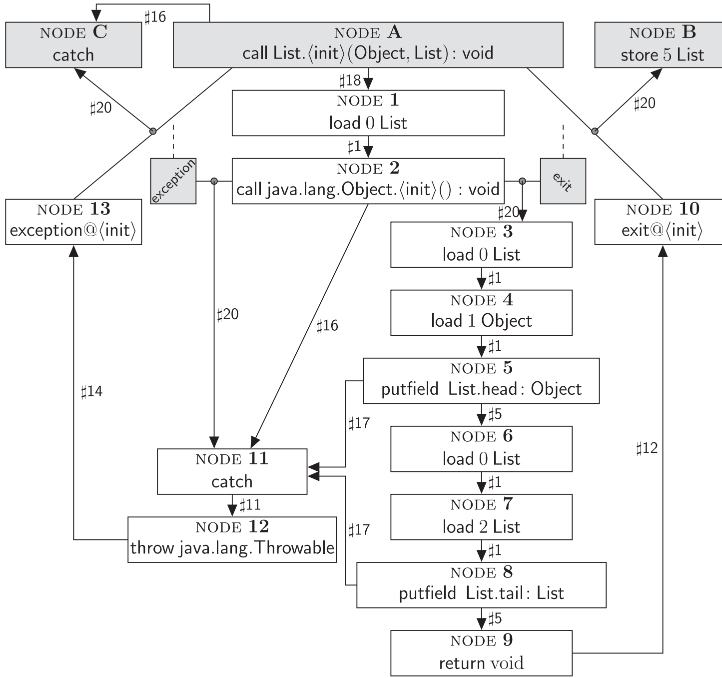


Fig. 9. The ACG for the method ⟨init⟩ in Figure 2.

Let us now explain, in more detail, the propagation rules given in Definition 5.4. They simulate the behavior of the concrete semantics of the bytecode instructions given in Figure 4.

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 propagation rules of the sequential arcs.

—load k t. 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 (hence $R[l_k/s_j]$). Moreover, l_k and s_j contain the same value, and if l_k might reach itself in R (by Definition 4.3 this can only happen if $\tau(l_k) \neq \texttt{int}$), then also s_j might reach itself in R's propagation.

—store k t. 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. Moreover, 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. A new object is created, bound to a location that is pushed onto the operand stack, as s_j. Therefore, the initial approximation R is kept, and since objects are not of primitive type, that is, since $\tau(s_j) \neq \texttt{int}$, we should also add the pair $s_j \rightsquigarrow s_j$, since the newly created object reaches itself. But it does not reach and is not reachable from anything else, since its fields are initialized to default values that are never locations and since it is held in a fresh location.

—getfield $\kappa.f$: t. The location on top of the operand stack, s_{j-1}, is replaced by the value of its field f. Hence any reachability pair in the initial approximation R that does not involve s_{j-1} should be present in the final approximation as well. Additionally, we consider all those variables b that might be reachable from field f (i.e., such that $s_{j-1} \rightsquigarrow b$) and all those variables a that might reach 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) reaches 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 static type of a reaches the type of the field that must be a reference type: $\tau(a) \rightsquigarrow \mathsf{t} \neq \mathtt{int}$. We improve this rule by restricting the preceding condition to only those variables a that might also share with the receiver s_{j-1}: if a and the receiver s_{j-1} do not share, then a does not reach any field of s_{j-1}. We observe that we do need sharing here and we cannot use the available reachability in R instead: if a does not reach the receiver s_{j-1}, it is well possible that a might reach one of its fields.

—putfield $\kappa.f$: t. The value on top of the operand stack, s_{j-1}, is stored in the field f of the object bound to the location below the top, 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 does not involve s_{j-1} nor s_{j-2}. Some additional pairs are added to the final approximation though: a variable a might reach, there, a variable b 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. The topmost operand stack element contains an integer length, replaced by a fresh location bound to the newly created array. The propagation is similar to that for new κ.

—arraylength α. The topmost operand stack element contains a reference to an array, replaced by its integer length, that is hence not reachable from anything and cannot reach anything.

—arrayload α. The kth element of the array, where k is on top of the operand stack and the array is at the location in the second topmost operand stack element s_{j-2}, is written on top of the stack, and both s_{j-1} and s_{j-2} are popped away. The propagation rule is similar to that for getfield $\kappa.f$: t and uses sharing analysis for the same reason. As for getfield $\kappa.f$: t, we cannot replace sharing information with the reachability information available at the beginning of the bytecode.

—arraystore α. The value held on top of the stack at s_{j-1} is stored in the kth element of the array, where k is an integer held at s_{j-2}, and the array is bound to the location at s_{j-3}. These three elements are popped away. The propagation rule is similar to that for putfield $\kappa.f$: t.

—dup t. A copy of s_{j-1} is pushed as s_j. Hence s_j and s_{j-1} become aliases, and 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 (hence $R[s_{j-1}/s_j]$). 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 v, add, sub, mul, div, rem, inc $k\,x$ deal with values of primitive type (or with `null`) and therefore do not introduce nor remove reachability, as it follows from Definition 4.3. On the other hand, catch and exception_is K do not modify the initial state, and therefore do not change the reachability information; ifne t and ifeq t just pop the topmost operand stack element, and therefore do

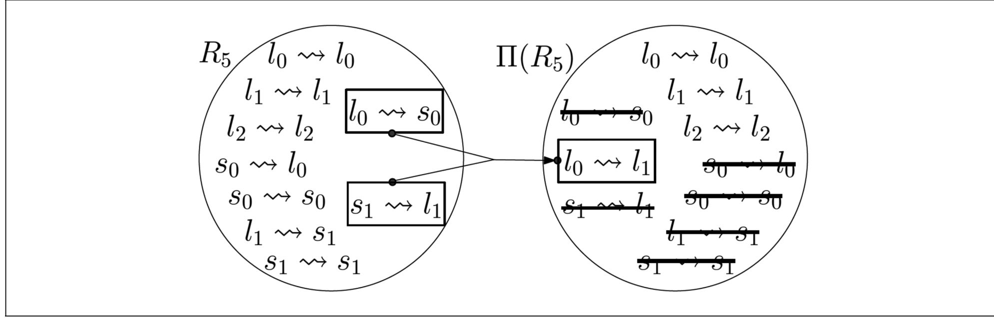


Fig. 10. Construction of $\Pi(R_5)$.

not modify the reachability information with respect to other variables. In all these cases, we keep the reachability pairs available in R if they only refer to variables that survive to the execution of the instruction.

Example 5.6. Consider nodes 4, 5, and 6 in Figure 9 and suppose that the reachability approximation at node 4 is

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

We have three local variables at those program points, one stack element at node 4, and two at node 5.

`this`	l_0	copy of `this`	s_0
`head`	l_1	copy of `head`	s_1
`tail`	l_2		

That is, $i_4 = i_5 = 3$, $j_4 = 1$, and $j_5 = 2$. Nodes 4 and 5 are linked through a sequential arc with propagation rule #1, while nodes 5 and 6 are linked through a sequential arc with propagation rule #5. By Definition 5.4,

$$\begin{aligned}\Pi(R_4) &= R_4 \cup R_4[l_1/s_1] \cup \{l_1 \rightsquigarrow s_1,\ s_1 \rightsquigarrow l_1\}\\ &= \{l_0 \rightsquigarrow l_0,\ l_1 \rightsquigarrow l_1,\ l_2 \rightsquigarrow l_2,\ l_0 \rightsquigarrow s_0,\ s_0 \rightsquigarrow l_0,\ s_0 \rightsquigarrow s_0,\ \underbrace{l_1 \rightsquigarrow s_1,\ s_1 \rightsquigarrow l_1,\ s_1 \rightsquigarrow s_1}_{\text{added pairs}}\}.\end{aligned}$$

Let now $R_5 = \Pi(R_4)$. According to Definition 5.4 (rule #5), we have

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

We illustrate the application of rule #5 on the set R_5 in Figure 10. Namely, the fact that $l_0 \rightsquigarrow s_0$ and $s_1 \rightsquigarrow l_1$ are in R_5 means that l_0 might reach s_0 (the receiver of the putfield) and that s_1 (the value written in a field of s_0) might reach l_1. Hence $\Pi(R_5)$ contains $l_0 \rightsquigarrow l_1$. Moreover, we remove from R_5 all pairs containing s_0 and s_1, since putfield pops these variables from the operand stack.

The *final arcs* feed nodes exit@m and exception@m for each method or constructor m. The former (respectively latter) contains the reachability information present in a state at a nonexceptional (respectively exceptional) end of m. Hence, exit@m is the sink of the arcs starting from the bytecode instructions return t in 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, 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's in Figure 2).

Example 5.7. Consider nodes 9 and 10 in Figure 9. The local variables in scope there are the same as in Example 5.6, and there is no stack element there. 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 through a final arc with propagation rule #12. By Definition 5.4, $\Pi(R_9)$ contains all pairs in R_9 that do not refer to an operand stack variable. Since, as we said, $j_9 = 0$, we conclude that $\Pi(R_9) = \{a \rightsquigarrow b \in R_9 \mid a, b \notin \emptyset\} = \{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_0 \rightsquigarrow l_2, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$.

The *exceptional arcs* link every instruction that might throw an exception to the catch at the beginning of its exception handler(s). Rules #14 and #15 are identical, but the latter is applied for a throw κ with a successor: the beginning of an exception handler inside its same method. 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 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 is cleared but the reachability among local variables remains unaffected.

Example 5.8. Consider nodes 5 and 11 in Figure 9. In Example 5.6, we concluded 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 through an exceptional arc with propagation rule #17. By Definition 5.4, $\Pi(R_5)$ contains the pairs from R_5 that do not refer to an operand stack element, and the pair $s_0 \rightsquigarrow s_0$, where s_0 holds the thrown exception and is the only operand stack variable available at node 11. Namely, $\Pi(R_5) = \{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, s_0 \rightsquigarrow s_0\}$, that is, we removed all pairs that contain an operand stack element different from s_0, since these elements are not available anymore.

We come now to the arcs that deal with method call and return. The *parameter passing arcs* link every node corresponding to a method call to that corresponding to the first bytecode instruction of the method(s) m_w that might be called there. Propagation rule #18 simply states that the actual parameters of m_w, held in the operand stack variables $s_{j-\pi}, \ldots, s_{j-1}$, are renamed into its formal parameters, that is, into the local variables $l_0, \ldots, l_{\pi-1}$. No other variables exist at the beginning of m_w.

Example 5.9. Consider nodes A and 1 in Figure 9. In Example 5.5, we assumed that

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

Nodes A and 1 are linked through 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). By Definition 5.4,

$$\Pi(R_A) = \left\{ (a \rightsquigarrow b) \left[\begin{smallmatrix} s_1/l_0 \\ s_2/l_1 \\ s_3/l_2 \end{smallmatrix} \right] \middle| \, a \rightsquigarrow b \in R_A \text{ and } a, b \in \{s_1, s_2, s_3\} \right\} = \{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}.$$

There is a *return value multi-arc* for each target m_w of a call. Rule #19 uses R_1 and R_2, approximations at the node corresponding to the call and at node $\boxed{\mathsf{exit@}m_w}$, 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 reassigned inside the callee m_w (i.e., there is no $\mathsf{store}\ l_{p-j+\pi}$ 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 reassigned inside the callee m_w (i.e., there is no $\mathsf{store}\ l_{p-j+\pi}$ 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 because of side-effects. 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 reassigned 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 5.10. Consider nodes A and 10 in Figure 9. In Example 5.5, we assumed that a reachability approximation at node A is known (Eq. (1)). Let $R_{10} = \Pi(R_9)$, where $\Pi(R_9) = \{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_0 \rightsquigarrow l_2, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$, as computed in Example 5.7. Consider the side-effect arc linking nodes A and 10 to node B. Let us illustrate the application of the propagation rule #20 on R_A and R_{10} in the presence of the sharing and aliasing approximations share_A and alias_A from Example 5.5. First of all, we note that *con* has $\pi = 3$ actual parameters: the implicit parameter `this` and two parameters of type `Object` and `List`, respectively. Since *con* has no return value, by Definition 3.18, we obtain $i_B = 5$ and $j_B = 1$, and for each variable $v \in \mathsf{dom}(\tau_B) = \{l_0, l_1, l_2, l_3, l_4, s_0\}$, we have $\tau_B(v) = \tau_A(v)$. By Definition 5.4, R_A and R_{10} are propagated by rule #20 as follows.

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

The left-hand side set $\{a \rightsquigarrow b \in R_A \mid a, b \in \mathsf{dom}(\tau_B)\}$ extracts from R_A the pairs composed of only the variables available in $\mathsf{dom}(\tau_B)$. Hence, the following pairs are added to R_B: $l_0 \rightsquigarrow l_0$, $l_2 \rightsquigarrow l_2$, $l_4 \rightsquigarrow l_4$, and $s_0 \rightsquigarrow s_0$. The right-hand side set enlarges R_B by adding new reachability pairs, as specified by the five conditions of 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 and gives rise to the following pairs: $\{l_0, l_2, s_0\} \times \{l_0, l_2, l_4, s_0\} \cup \{l_4 \rightsquigarrow l_4\}$. Conditions 3 and 4 improve the precision of this approximation. Namely, condition 3 allows as first element of a pair only those variables that might share with an actual parameter of *con* at A, and only l_2, l_4, and s_0 satisfy this condition according to share_A. On the other hand, condition 4 allows as second element of a pair only those variables that might be reachable from an actual parameter of *con* at A, and only l_4, and s_0 satisfy this requirement ($s_2 \rightsquigarrow l_4, s_1 \rightsquigarrow s_0 \in R_A$). Therefore, these two conditions restrict the former approximation to $\{l_2, s_0\} \times \{l_4, s_0\} \cup \{l_4 \rightsquigarrow l_4\}$. Condition 5 adds no further improvement. Therefore, rule #20 adds to R_B, the overapproximation of the actual reachability information at node B, the following pairs of variables.

$$R_B = \{l_0 \rightsquigarrow l_0, l_2 \rightsquigarrow l_2, l_4 \rightsquigarrow l_4, s_0 \rightsquigarrow s_0, \underbrace{l_2 \rightsquigarrow l_4, l_2 \rightsquigarrow s_0, s_0 \rightsquigarrow l_4}_{\text{added pairs}}\}. \quad (2)$$

We note that, as we hinted at the end of Example 5.5, our reachability analysis actually provides the pair $s_0 \rightsquigarrow l_4$ in the approximation at node B.

Example 5.11. Propagation rules #4, #19, and #20 use possible sharing and definite aliasing information between program variables. As we mentioned previously, 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 that we do not know if they are definitely aliased ($\mathsf{alias}'_A = \varnothing$), 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., without possible sharing and definite aliasing approximations), conditions 1–4 give rise to the following pairs: $\{l_0, l_2, s_0\} \times \{l_4, s_0\} \cup \{l_4 \rightsquigarrow l_4\}$, and condition 5 would not remove any pair. In this case, the propagated reachability information becomes

$$\{l_0 \rightsquigarrow l_0,\, l_2 \rightsquigarrow l_2,\, l_4 \rightsquigarrow l_4,\, s_0 \rightsquigarrow s_0,\, \underbrace{l_0 \rightsquigarrow l_4,\, l_2 \rightsquigarrow l_4,\, l_2 \rightsquigarrow s_0,\, s_0 \rightsquigarrow l_4}_{\text{added pairs}}\},$$

which is less precise than Eq. (2).

In our experiments (Section 6), the reachability analysis is performed inside the nullness and termination tools of Julia that already perform definite aliasing [Nikolić and Spoto 2012b] and possible sharing [Secci and Spoto 2005] analyses.

Once an ACG is built from the program, we can consider its solution.

Definition 5.12 (*Reachability Analysis*). A *solution* of an ACG is an assignment of an element $S_n \in \mathsf{A}_\tau$ to each node n of the ACG, where τ is the type environment associated to n such that the approximation for the initial node *first*(`main`) of the `main` method is $S_{\mathit{first}(\mathtt{main})} = \{l_0 \rightsquigarrow l_0,\, l_1 \rightsquigarrow l_1\}$ and such that the propagation rules of the arcs are satisfied, that is, for every arc from nodes $n_1 \dots n_k$[5] to n' with propagation rule $\lambda R_1, \dots, \lambda R_k.\Pi(R_1, \dots, R_k)$, the condition $\Pi(S_{n_1}, \dots, S_{n_k}) \subseteq S_{n'}$ holds. The reachability analysis of the program is *the least solution* of its ACG *with respect to set inclusion*. The result of our *reachability analysis at* a bytecode instruction of the program ins is $S_{\boxed{\mathsf{ins}}}$.

The condition on the initial node of `main` states that at the beginning of `main`, variable `this`, held in local variable 0, might reach itself and variable `args`, held in local variable 1, might reach itself.[6] We observe that a minimal solution exists, since all propagation rules are monotonic with respect to set-inclusion (Definition 5.4). It can hence be computed by starting from the empty approximation for every node and then propagating this approximation along the arcs, until stabilization.

Example 5.13. In Figure 11, we give the minimal solution of the abstract constraint graph from Example 5.5, in the hypotheses of that example. Consider, for example, node 11. By Definition 5.12 it should satisfy the following constraints.

$$\begin{aligned} S_{11} &\supseteq \Pi^{\#20}(S_2, S_{\mathsf{exception}}) \\ S_{11} &\supseteq \Pi^{\#16}(S_2) \\ S_{11} &\supseteq \Pi^{\#17}(S_5) \\ S_{11} &\supseteq \Pi^{\#17}(S_8), \end{aligned}$$

[5] According to Definition 5.4, either $k = 1$ or $k = 2$.

[6] In Java, `main` is static and has consequently no `this` variable. In that case, the initial reachability would only state that the `args` parameter might reach itself.

<table>
<tr><th>NODE n</th><th>SOLUTION (REACHABILITY APPROXIMATION AT S_n)</th></tr>
<tr><td>A</td><td>$\{l_0 \rightsquigarrow l_0, l_2 \rightsquigarrow l_2, l_4 \rightsquigarrow l_4, l_2 \rightsquigarrow s_3, l_4 \rightsquigarrow s_2, s_2 \rightsquigarrow l_4, s_0 \rightsquigarrow s_0, s_0 \rightsquigarrow s_1, s_1 \rightsquigarrow s_0, s_1 \rightsquigarrow s_1, s_2 \rightsquigarrow s_2, s_3 \rightsquigarrow s_3\}$</td></tr>
<tr><td>1</td><td>$\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$</td></tr>
<tr><td>2</td><td>$\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, s_0 \rightsquigarrow l_0, s_0 \rightsquigarrow s_0\}$</td></tr>
<tr><td>3</td><td>$\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$</td></tr>
<tr><td>4</td><td>$\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, s_0 \rightsquigarrow l_0, s_0 \rightsquigarrow s_0\}$</td></tr>
<tr><td>5</td><td>$\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, l_1 \rightsquigarrow s_1, s_0 \rightsquigarrow l_0, s_1 \rightsquigarrow l_1, s_0 \rightsquigarrow s_0, s_1 \rightsquigarrow s_1\}$</td></tr>
<tr><td>6</td><td>$\{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$</td></tr>
<tr><td>7</td><td>$\{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, s_0 \rightsquigarrow l_0, s_0 \rightsquigarrow l_1, s_0 \rightsquigarrow s_0\}$</td></tr>
<tr><td>8</td><td>$\{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, l_2 \rightsquigarrow s_1, s_0 \rightsquigarrow l_0, s_0 \rightsquigarrow l_1, s_1 \rightsquigarrow l_2, s_0 \rightsquigarrow s_0, s_1 \rightsquigarrow s_1\}$</td></tr>
<tr><td>9</td><td rowspan="2">$\{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_0 \rightsquigarrow l_2, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$</td></tr>
<tr><td>10</td></tr>
<tr><td>11</td><td rowspan="3">$\{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, s_0 \rightsquigarrow s_0\}$</td></tr>
<tr><td>12</td></tr>
<tr><td>13</td></tr>
<tr><td>B</td><td rowspan="2">$\{l_0 \rightsquigarrow l_0, l_2 \rightsquigarrow l_2, l_2 \rightsquigarrow s_0, l_2 \rightsquigarrow l_4, l_4 \rightsquigarrow l_4, s_0 \rightsquigarrow l_4, s_0 \rightsquigarrow s_0\}$</td></tr>
<tr><td>C</td></tr>
</table>

Fig. 11. The solution of our abstract constraint graph.

where $\Pi^{\#16}$, $\Pi^{\#17}$, and $\Pi^{\#20}$ are instances of the propagation rules #16, #17, and #20 of Definition 5.4. Since there is no other arc entering node 11, we conclude that a least solution must be such that $S_{11} = \Pi^{\#20}(S_2, S_{\mathsf{exception}}) \cup \Pi^{\#16}(S_2) \cup \Pi^{\#17}(S_5) \cup \Pi^{\#17}(S_8)$.

5.3. Soundness

In this section, we show the soundness of our analysis. We first enunciate several lemmas stating that the propagation rules corresponding to each type of arcs of our ACG are sound (Lemmas 5.14– 5.20). These lemmas are proven in Appendix B. Then, the main soundness result is shown in Theorem 5.21.

Lemma 5.14 states that the sequential arcs propagate only the nonexceptional 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 with its normally subsequently executed bytecode, when no exception is thrown, and their semantics must hence be consistent with that situation.

LEMMA 5.14. *The propagation rules for the sequential arcs of Definition 5.4 are sound. That is, 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 nonexceptional execution. Then, for every* $R \in \mathsf{A}_\tau$, *we have*

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

(We recall that ins is the semantics of ins, *see Figure 4.)*

The propagation rules of the final arcs soundly approximate the concrete behavior of a final bytecode instruction (return t, return void, throw κ) of a method or a constructor. We can similarly prove soundness for the propagation rules of the exceptional arcs that simulate the exceptional executions of the bytecode instructions that might throw an exception. Lemmas 5.15 and 5.16 formalize these facts.

LEMMA 5.15. *The propagation rules for the final arcs of Definition 5.4 are sound. That is, consider a final arc from* ins *and its propagation rule* Π. *Assume that* ins *has static type information* τ *at its beginning and* τ' *immediately after its execution (its*

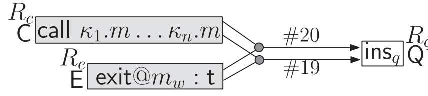


Fig. 12. Arcs going into the node corresponding to ins$_q$.

nonexceptional execution if ins *is a* return*, its exceptional execution if* ins *is a* throw κ*). Then, for every* $R \in \mathsf{A}_\tau$*, we have*

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

(We recall that ins is the semantics of ins*, see Figure 4.)*

LEMMA 5.16. *The propagation rules for the exceptional arcs of Definition 5.4 not leaving a* call *are sound. That is, 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' *after its exceptional execution. Then, for every* $R \in \mathsf{A}_\tau$*, we have*

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

(We recall that ins is the semantics of ins*, see Figure 4.)*

Similarly, Lemma 5.17 shows that the propagation rules of the parameter passing arcs are sound. Namely, they soundly approximate the behavior of the *makescope* function.

LEMMA 5.17. *The propagation rules for the parameter passing arcs of Definition 5.4 are sound. That is, consider a parameter passing arc from a* call $m_1 \ldots m_n$ *to the first bytecode of* m_w*, for some* $w \in [1..k]$*, and its propagation rule* Π*. Assume that* call $m_1 \ldots m_n$ *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* $R \in \mathsf{A}_\tau$*, we have*

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

The following lemmas deal with the return from a method call. Namely, in the case of a non-void method, the propagation rule of the return value arc expands the reachability approximation immediately after a call to that method with those reachability pairs related to the returned value. A method execution might also have side-effects on the memory, and this is captured by the propagation rule of the side-effects arcs. The reachability approximation after the call to the method is, therefore, determined as the union of the propagations of a return value arc (for non-void methods) and a side-effects arc (Figure 12), which is proved sound (Lemmas 5.18 and 5.19).

LEMMA 5.18. *The propagation rules for the return value arcs and side-effect 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* C = [call $m_1 \ldots m_n$] *and* E = [exit@m_w] *to a node* Q = [ins$_q$] *and their propagation rules* $\Pi^{\#19}$*, and* $\Pi^{\#20}$*, respectively. We depict this situation in Figure 12. 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 *, that is, a partial function from a state at its beginning to the corresponding state at its end. Then, for every* $R_c \in \mathsf{A}_{\tau_c}$ *and* $R_e \in \mathsf{A}_{\tau_e}$*, we have*

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

LEMMA 5.19. *The propagation rule for the side-effects arcs is sound for* void *methods. Namely, let* $w \in [1..n]$ *and consider a side-effect arc from nodes* C = [call $m_1 \ldots m_n$] *and* E = [exit@m_w] *to a node* Q = [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*,*

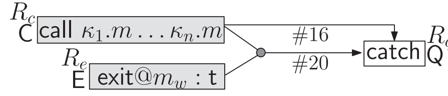


Fig. 13. Arcs going into the node corresponding to catch.

that is, a partial function from a state at its beginning to the corresponding state at its end. Then, for every $R_c \in \mathsf{A}_{\tau_c}$ and $R_e \in \mathsf{A}_{\tau_e}$, we have

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

The following lemma deals with the the executions of a method that end up in an exception being thrown. Namely, the approximation of the reachability information at the catch that runs from the exceptional states must consider the possible side-effects on the initial memory due to the execution of the method. This is the task of the propagation rules of the side-effects arcs. On the other hand, that approximation must also consider the case when the method is invoked on `null`. As in the previous case, the approximated reachability information must hence be consistent with both these situations and Lemma 5.20 shows it correct.

LEMMA 5.20. *The propagation rules for the exceptional arcs of the* call *and side-effects arcs are sound when a method call throws an exception. Namely, given nodes* $\mathsf{Q} =$ [catch], $\mathsf{C} =$ [call $m_1 \ldots m_n$], *and* $\mathsf{E} =$ [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* $\Pi^{\#16}$ *and* $\Pi^{\#20}$*, respectively. We depict this situation in Figure 13. 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, that is, a partial function from a state at its beginning to the corresponding state at its end. Then, for every $R_c \in \mathsf{A}_{\tau_c}$ and $R_e \in \mathsf{A}_{\tau_e}$, we have*

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

Finally, Theorem 5.21 shows that our reachability analysis is sound, that is, at each program point, the set of reachability pairs computed by our analysis overapproximates the actual reachability information at that point. We give only a sketch of the proof. More details are in Appendix B.

THEOREM 5.21 (SOUNDNESS). *Let* $\langle b_{first(\mathtt{main})} \parallel \xi \rangle \Rightarrow^* \langle$ [ins / rest $\rightarrow b_1 \cdots \rightarrow b_m$] $\parallel \sigma \rangle :: a$ *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 reachability except* `this` *that reaches itself and the* `args` *parameter that 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 $A \in \mathsf{A}_\tau$ be the reachability approximation at* ins*, as computed by our reachability analysis. Then, $\sigma \in \gamma_\tau(A)$ holds.*

PROOF. The blocks in the configurations of an activation stack, except the topmost, cannot be empty and without successor. This is because the configurations are only stacked by rule (2) of Figure 5, and, if *rest* is empty there, then $m \geq 1$, otherwise the code ends with a call bytecode with no return, which is illegal in Java bytecode [Lindholm and Yellin 1999].
We proceed by induction on the length n of the execution $\langle b_{first(\,\mathtt{main})} \parallel \xi \rangle \Rightarrow^*$ $\langle$ [ins / rest $\rightarrow b_1 \cdots \rightarrow b_m$] $\parallel \sigma \rangle :: a$.

Base Case. If $n = 0$, the execution is just $\langle b_{\mathit{first}(\text{main})} \parallel \xi\rangle$. In this case, $\tau_0 = \tau$ and $A_0 = A = S_{\mathit{first}(\mathtt{main})}$. Since ξ contains no reachability except for `this` and `args` held in l_0 and l_1 that reach themselves, that is, $l_0 \rightsquigarrow^{\xi} l_0$, $l_1 \rightsquigarrow^{\xi} l_1$, and since our reachability analysis is a solution where $S_{\mathit{first}(\mathtt{main})} = \{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1\}$ (Definition 5.12), we have $\sigma = \xi \in \gamma_{\tau_0}(\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1\}) \subseteq \gamma_\tau(S_{\mathit{first}(\mathtt{main})}) = \gamma_\tau(A_0) = \gamma_\tau(A)$.

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

$$\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi\rangle \Rightarrow^{n_p} \langle \overbrace{\boxed{\begin{smallmatrix}\mathsf{ins}_p\\ \mathit{rest}_p\end{smallmatrix}}}^{b_p}\begin{smallmatrix}\to b'_1\\ \cdots\\ \to b'_{m'}\end{smallmatrix} \parallel \sigma_p\rangle :: a_p \Rightarrow^{n+1-n_p} \langle b_q \parallel \sigma_q\rangle :: a_q, \tag{3}$$

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 nonempty block b_p. Let such n_p be maximal. Given a bytecode ins_a, let τ_a and R_a be the static type information and the approximation of the reachability information at the ACG node $\boxed{\mathsf{ins}_a}$, respectively. By inductive hypothesis, we know that $\sigma_p \in \gamma_{\tau_p}(R_p)$. We show that $\sigma_q \in \gamma_{\tau_q}(R_q)$ as well. We distinguish on the basis of the rule of the operational semantics (Figure 5) that is applied at the beginning of the derivation $\Rightarrow^{n+1-n_p}$ in (3).

Rule (1). If this rule is applied, then $\mathit{ins}_p(\sigma_p)$ is defined, and ins_p is not a call. We distinguish the following cases.

—ins_p is not a return nor a throw. In this case, the ACG contains a sequential arc connecting the ACG nodes corresponding to ins_p and ins_q and by Lemma 5.14, $\sigma_q \in \gamma_{\tau_q}(R_q)$.

—ins_p is a return. in this case, the operational semantics introduced in Figure 5 imposes that after $n_c < n_p$ transitions, there is a method invocation $\mathsf{ins}_c = \mathsf{call}\ m_1 \ldots m_n$, and that ins_p is placed at the end of a callee m_w for a $0 \leq w \leq n$. We suppose that m_w is a non-void method; the other case can be dealt with in a similar way. By Definition 5.4, the ACG contains nodes $\boxed{\mathsf{ins}_c}$, $\boxed{\mathsf{ins}_p}$, $\boxed{\mathsf{exit@}m_w\mathsf{:t}}$, and $\boxed{\mathsf{ins}_q}$. Let σ_c be the state the method is invoked from and let $\sigma_e = \mathit{ins}_p(\sigma_p)$. There is a final arc connecting $\boxed{\mathsf{ins}_p}$ and $\boxed{\mathsf{exit@}m_w\mathsf{:t}}$, and by the hypothesis $\sigma_p \in \gamma_{\tau_p}(R_p)$ and by Lemma 5.15, we have $\sigma_e \in \gamma_{\tau_e}(R_e)$, where $R_e = \Pi(R_p)$. There are also two arcs (a return value and a side-effect arc) going from $\boxed{\mathsf{ins}_c}$ and $\boxed{\mathsf{exit@}m_w\mathsf{:t}}$ into $\boxed{\mathsf{ins}_q}$. Then, by the hypothesis $\sigma_c \in \gamma_{\tau_c}(R_c)$ and by Lemma 5.18, we have $\sigma_q \in \gamma_{\tau_q}(R_q)$, where $R_q = \Pi^{\#19}(R_c, R_e) \cup \Pi^{\#20}(R_c, R_e)$.

—ins_p is a throw. Then, if $\mathsf{ins}_q = \mathsf{catch}$, $\boxed{\mathsf{ins}_p}$ and $\boxed{\mathsf{ins}_q}$ are connected by an exceptional arc, and by hypothesis $\sigma_p \in \gamma_{\tau_p}(R_p)$ and by Lemma 5.16, we have $\sigma_q \in \gamma_{\tau_q}(R_q)$, where $R_q = \Pi^{\#15}(R_p)$. Otherwise, that is, when $\mathsf{ins}_q \neq \mathsf{catch}$, ins_p must occur at the end of a method, and similarly to the previous case (but using Lemma 5.20 instead of Lemma 5.18), we conclude that $\sigma_q \in \gamma_{\tau_q}(R_q)$.

Rule (2). In this case, the ACG contains nodes $\boxed{\mathsf{ins}_p}$ and $\boxed{\mathsf{ins}_q}$ connected through a parameter passing arc. Hence, by hypothesis $\sigma_p \in \gamma_{\tau_p}(R_p)$ and by Lemma 5.17, we have $\sigma_q \in \gamma_{\tau_q}(R_q)$, where $R_q = \Pi^{\#18}(R_p)$.

Rule (3). The result follows from the soundness of propagation rule #16. □

6. EXPERIMENTS

We have implemented our reachability analysis inside the Julia analyzer for Java and Android.[7] It is a commercial tool developed by Julia Srl, a spin-off company of the University of Verona. In this section, we describe the tool and the experiments that we have performed in order to evaluate our reachability analysis.

6.1. 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 to analyze loops and recursion. An example is the nullness analysis in Spoto [2008]. 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 [Bryant 1986] are typically used to implement denotations, and Julia provides support for this choice.

Constraint-based analyses follow the approach used, for instance, in this article. A constraint 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 5.2). They might also stand for local variables, stack elements, fields or return values, when the abstract interpretation abstracts values rather than states (e.g., [Spoto and Ernst 2011; Nikolić and Spoto 2012b]). 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 article, see Definition 5.12) or their intersection. In the first case, we get a possible analysis (an overapproximation of the property under analysis); in the second case, we get a definite analysis (an underapproximation of the property under analysis) [Nikolić 2013]. In all cases, Julia provides standard implementations of the construction of the constraint 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 `longs`) and keep the memory footprint small (singletons are created once; bitsets are very compact). The fixpoint algorithm that finds a solution (Definition 5.12) 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 of the program; each bytecode instruction induces a state transition in the automaton. The possible states of the automaton at a program point are an abstraction of all the execution paths that

[7] `http://www.juliasoft.com`.

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 [Nikolić and Spoto 2012a, 2013].

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 Secci and Spoto [2005].

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 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. The rationale here is that, for those fields, the fact that they hold `null` before their first assignment is irrelevant to determine if they can hold `null` or not when they are accessed. Only explicit assignments are relevant. This consideration is important, for instance, for a subsequent nullness analysis that is only left to prove that the values explicitly written into f are non-`null` without bothering about the default value. 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 Spoto [2008, 2011] and Nikolić and Spoto [2013]. It exploits the available reachability information (Section 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 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 1). More detail in Spoto et al. [2010].

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 Spoto [2011]. It uses a denotational nullness analysis for local and stack variables [Spoto 2008] 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 [Nikolić and Spoto 2012a, 2013]; 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 [Nikolić and Spoto 2012b]). 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 Spoto et al. [2010]. 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.

6.2. Sample Programs

We have analyzed a set of sample programs. Most of them are Android applications: Mileage, OpenSudoku, Solitaire, and TiltMazes[8]; ChimeTimer, Dazzle, OnWatch, and Tricorder[9]; TxWthr.[10] Others are Java programs: JFlex is a lexical analyzers generator[11]; Plume is a library by Michael D. Ernst[12]; NTI is a nontermination analyzer by Étienne Payet[13]; Lisimplex is a numerical simplex implementation by Ricardo Gobbo.[14] The remaining are sample Android programs taken from the Android 3.1 distribution by Google and are bundled with the Android SDK Tool r12.[15]

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

6.3. Sharing vs. Reachability Analysis

Figure 14 shows that reachability analysis is in general more expensive than sharing analysis, also because its times include those of the preliminary, supporting 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 15, 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 this 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 6.5). 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.

[8]`http://f-droid.org/repository/browse.`
[9]`http://moonblink.googlecode.com/svn/trunk.`
[10]`http://typoweather.googlecode.com/svn/trunk.`
[11]`http://jflex.de.`
[12]`http://code.google.com/p/plume-lib.`
[13]`http://personnel.univ-reunion.fr/epayet/Research/NTI/NTI.html.`
[14]`http://sourceforge.net/projects/lisimplex.`
[15]`http://developer.android.com/tools/revisions/platforms.html.`

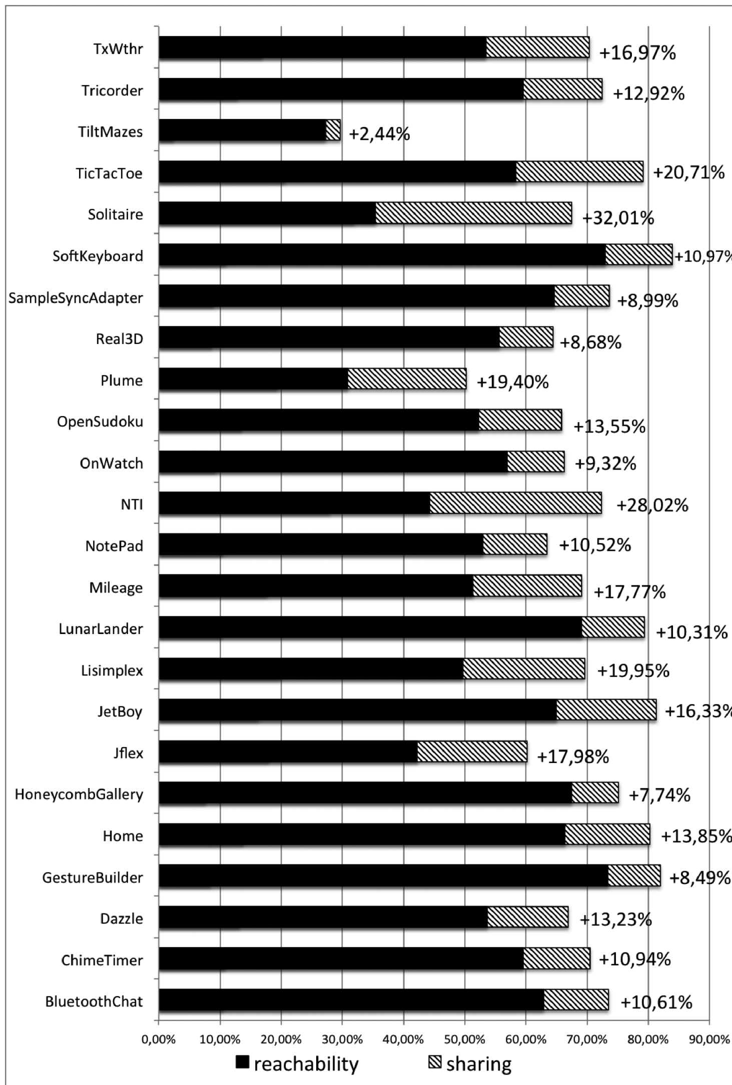


Fig. 14. Comparison between the runtimes of sharing analysis with the runtimes of sharing and reachability analyses together.

6.4. 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, (e.g., [Pheng and Verbrugge 2005; Jump and McKinley 2009]), but dynamic analyses are only sound with respect to the execution traces that are generated at runtime 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 [Corbett 2000] is intraprocedural only; experiments do not report its cost in time. The second [Marron et al. 2008] is able to analyze interprocedural Java programs;

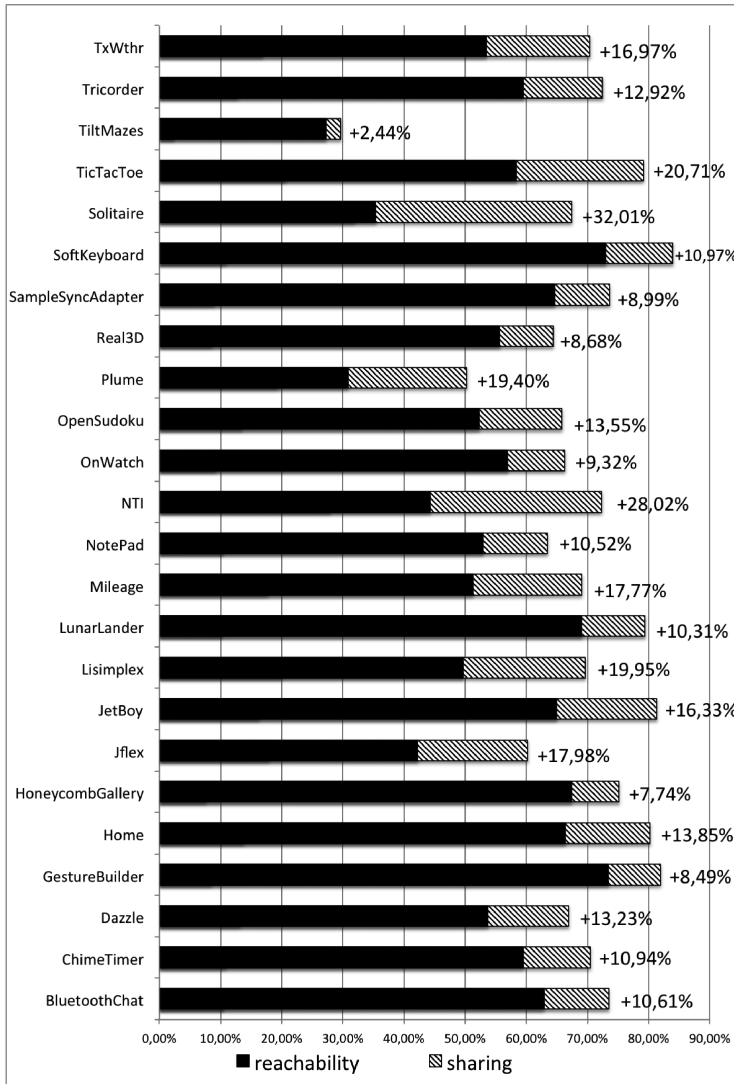


Fig. 15. Improvement of precision (in percentage) of Julia when the reachability property is computed by our reachability analysis with respect to the previous approach where this property was computed by sharing analysis. Here, precision is the ratio of ordered pairs of distinct variables $\langle v, w\rangle$ such that 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 (Definition 4.8), 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 4.10).

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. For comparison, our reachability analysis analyzes `OnWatch` in 32 seconds, although it is made up of 112,423 statements (Figure 16) and although we analyze the libraries along with the application. If one considers that

sharing is needed before reachability, the total time of our analysis amounts in this case 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 Marron et al. [2008] with respect to reachability information, but the major difference in the computational cost of the two analyses is apparent. It is true that our hardware is multicore and so potentially faster than that used in Marron et al. [2008], but sharing and reachability analyses are each performed sequentially in Julia, so that only one core is used for each of them.

6.5. 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 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 16 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 17 presents our experiments with the nullness and termination tools of Julia and reports their runtime, including reachability analysis. In eight cases over 24, the extra reachability information improves the precision of the nullness checking tool, but this never happens for termination, consistent with the fact that cyclicity is not improved (Figure 16). 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[16] that is performed every year. Those programs, although small and often unrealistic, are nevertheless interesting, since the proof of their termination often requires nontrivial arguments, also related to objects dynamically allocated in memory. We have performed their termination analysis with reachability analysis and then again without reachability analysis (but always with sharing analysis turned on). Over a total of 436 test programs, Julia without reachability proves that 276 of them terminate or can definitely diverge. If reachability is used to strengthen the analysis, this figure grows to 282: 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. Note that even when reachability is not applied, sharing is in place and can support the missing reachability. Those six examples are consequently those where reachability is needed and sharing is not enough. We observe that among the remaining 154 tests, there are some that are known to diverge and a few for which no proof of termination or divergence exists, not even by hand.

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

Program	Lang.	Source Lines	Analyzed Lines	Prec. of Side-Effects Analysis		Prec. of Field Initial. Analysis		Prec. of Cyclicity Analysis	
				without reach.	with reach.	without reach.	with reach.	without reach.	with reach.
BluetoothChat	Android	616	84,415	645.99	540.23	2,185	2,325	12.85%	12.85%
ChimeTimer	Android	1,090	89,565	730.68	618.08	2,348	2,486	13.54%	13.54%
Dazzle	Android	1,791	77,828	309.89	225.96	2,417	2,447	22.19%	22.19%
GestureBuilder	Android	502	84,346	667.70	557.52	2,162	2,282	16.57%	16.57%
Home	Android	870	87,413	693.80	584.01	2,274	2,415	10.89%	10.89%
HoneycombGallery	Android	948	71,558	333.25	242.32	2,131	2,175	33.33%	33.33%
JFlex	Java	7,681	40,779	357.59	243.89	1,092	1,146	33.67%	33.67%
JetBoy	Android	839	65,174	281.48	198.71	2,173	2,202	11.79%	11.79%
Lisimplex	Java	768	49,303	637.69	347.96	1,356	1,433	14.13%	14.13%
LunarLander	Android	538	57,675	270.87	191.07	1,880	1,911	18.11%	18.11%
Mileage	Android	5,877	104,009	959.30	804.98	2,636	2,794	25.45%	25.45%
NotePad	Android	705	73,742	293.57	218.17	2,108	2,139	37.50%	37.50%
NTI	Java	2,372	13,486	24.11	13.51	465	467	32.59%	32.59%
OnWatch	Android	6,295	112,423	1,299.51	796.89	3,232	3,399	32.59%	32.59%
OpenSudoku	Android	5,877	90,810	440.36	344.92	2,622	2,660	22.57%	22.57%
Plume	Java	8,586	43,637	186.31	126.71	1,316	1,335	57.11%	57.11%
Real3D	Android	1,228	74,350	497.94	400.73	2,093	2,189	36.43%	36.43%
SampleSyncAdapter	Android	978	65,971	328.80	235.68	2,111	2,142	42.77%	42.77%
SoftKeyboard	Android	703	58,088	174.01	116.96	2,112	2,131	11.21%	11.21%
Solitaire	Android	3,905	62,065	243.19	166.57	1,957	1,982	50.06%	50.06%
TicTacToe	Android	607	59,160	228.27	154.35	1,919	1,943	20.73%	20.73%
TiltMazes	Android	1,853	89,653	650.45	562.57	2,313	2,454	71.57%	71.57%
Tricorder	Android	5,317	98,389	783.59	663.23	2,806	2,942	33.91%	33.91%
TxWthr	Android	2,024	74,537	309.24	229.79	2,220	2,258	15.39%	15.39%
average precision				**472.81**	**361.86 (-23.47%)**	**2,080.33**	**2,152.37 (+3.46%)**	**26.84%**	**26.84% (+0.00%)**

Fig. 16. The effects of reachability analysis on the precision of side-effects, field initialization, and cyclicity analyses. *Source lines* counts noncomment nonblank 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 noncyclical 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	**5,915.32**			**5,456.24 (-7.77%)**			**4,206.71**			**4,138.92 (-1.62%)**		
total warnings	**830**			**802 (-3.38%)**			**307**			**307 (+0.00%)**		

Fig. 17. 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 recursion (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.

For both nullness and termination checking, the presence of reachability analysis actually reduces the total runtime of the tools. This is because reachability helps subsequent analyses, in particular side-effects analysis, and prevents them from generating spurious information. For instance, side-effects analysis computes much smaller sets of affected fields per method (Figure 16, compare the 5th and the 6th columns).

7. CONCLUSION

Our reachability analysis is an instantiation of the general parameterized framework for constraint-based static analyses of Java bytecode [Nikolić 2013]. Another analysis that can be instantiated in that framework is, for example, field initialization analysis [Spoto and Ernst 2011]. The difference between the latter and our reachability analysis is that in the present article. the constraint nodes stand for program points rather than for single variables, as is the case in Spoto and Ernst [2011]. As a consequence, the abstract domain as well as the propagation rules are completely different in the two instances. A sound reachability analysis, in particular, requires a precise approximation of the side-effects of method calls.

Our constraint-based approach is now being applied to develop other analyses, such as definite aliasing of variables to expressions [Nikolić and Spoto 2012b]. In general, one obtains a new constraint-based static analysis of Java bytecode by specifying an opportune abstract domain and the propagation rules representing an abstract semantics of the bytecode instructions. It is worth noting that the construction of the abstract constraint graph is always performed along a fixed, given pattern. Our Julia analyzer includes a library for defining constraint-based analyses and for computing their solution, highly optimized with respect to space and time.

APPENDIXES

A. REACHABILITY

In this appendix, we show some technical lemmas related to Section 4. They are used in Appendix B, where Lemmas 5.14–5.20 are proved.

Lemma A.1 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 A.1. *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

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

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

since $\ell'' \in \mathsf{L}_\mu^i(\ell'')$. □

Proposition A.2 (Proposition 4.7). *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* $\mathsf{ins} = \mathsf{call}\ m_1 \dots m_n$ *and* $\mathsf{ins}' \neq \mathsf{catch}$ *are executed. Namely,* σ' *is a nonexceptional state obtained at the end of execution of a callee* $m(\vec{\mathsf{t}})\colon \mathsf{t}$ *in* σ*, for a* $w \in [1..n]$*, the topmost* π *stack elements of* σ *(values* $\mathsf{s}[|\mathsf{s}|-1], \dots, \mathsf{s}[|\mathsf{s}|-\pi]$*) and the topmost operand stack element of* σ' *(value* $\mathsf{s}'[|\mathsf{s}'|-1]$*) 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*.*

$$
\mathcal{L}_\sigma = \mathsf{dom}(\mu) \smallsetminus \bigcup_{|\mathsf{s}|-\pi \le i \le |\mathsf{s}|-1} \mathsf{L}_\sigma(\mathsf{s}[i]).
$$

Then, the following conditions hold:

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

Proof. It is enough to prove that during the execution of the callee(s) $m_1, \dots, m_n$ 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 following thesis.

(1) Only $\mathsf{putfield}\ \kappa.f\colon \mathsf{t}$ 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 $\mathsf{putfield}\ \kappa.f\colon \mathsf{t}$ 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 putfield $\kappa.f$: t and arraystore α 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 getfield $\kappa.f$: t and arrayload α 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 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. □

Lemmas A.3, A.4, and A.5 highlight some important properties of the set of locations that are not reachable from the actual arguments passed to a method at call time ($\mathcal{L}_\sigma$). This set has been introduced in Proposition 4.7. The following lemmas show why that proposition is important: it allows us to restrict the effects of a method call on the state of the caller and allows us to restrict the possible reachability of the return value of the method to and from the variables of the caller. Consequently, we will use these results in the proofs of soundness for the propagation rules in Lemmas 5.18–5.20.

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 A.3 shows that this variable does not share with any location belonging to $\mathcal{L}_\sigma$.

LEMMA A.3. *Under the hypotheses of Proposition 4.7, consider a variable x such that $\rho'(x) \in \mathbb{L} \smallsetminus \mathcal{L}_\sigma$. Then, $\mathsf{L}_{\sigma'}(x) \cap \mathcal{L}_\sigma = \varnothing$.*

PROOF. We prove that $\forall i \in \mathbb{N}.\mathsf{L}^i_{\sigma'}(x) \cap \mathcal{L}_\sigma = \varnothing$, and we do it by induction on i.

Base Case. Since $\mathsf{L}^0_{\sigma'}(x) = \rho'(x) \notin \mathcal{L}_\sigma$, we have $\mathsf{L}^0_{\sigma'}(x) \cap \mathcal{L}_\sigma = \varnothing$.

Inductive Step. Suppose that $\mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma = \varnothing$, let us prove that $\mathsf{L}^{n+1}_{\sigma'}(x) \cap \mathcal{L}_\sigma = \varnothing$. It is worth noting that $\mathsf{L}^n_{\sigma'}(x) \subseteq \mathsf{dom}(\mu')$. By the third condition of Proposition 4.7, we have $\forall \ell \in \mathsf{dom}(\mu') \smallsetminus \mathcal{L}_\sigma.\mathsf{rng}(\mu'(\ell).\phi) \cap \mathcal{L}_\sigma = \varnothing$. Therefore, $\mathsf{L}^n_{\sigma'}(x) \subseteq \mathsf{dom}(\mu') \smallsetminus \mathcal{L}_\sigma$, since $\mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma = \varnothing$ and $\mathsf{L}^n_{\sigma'}(x) \subseteq \mathsf{dom}(\mu')$. This entails that $\forall \ell \in \mathsf{L}^n_{\sigma'}(x).\mathsf{rng}(\mu'(\ell).\phi) \cap \mathcal{L}_\sigma = \varnothing$, which implies

$$\bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma = \varnothing. \tag{4}$$

We have

$$\begin{aligned}
\mathsf{L}^{n+1}_{\sigma'}(x) \cap \mathcal{L}_\sigma &= \left(\mathsf{L}^n_{\sigma'}(x) \cup \bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L})\right) \cap \mathcal{L}_\sigma && \text{[By Definition 4.2]}\\
&= \left(\mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma\right) \cup \left(\bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma\right) && \text{[By distributivity]}\\
&= \varnothing \cup \varnothing = \varnothing. && \text{[By hypothesis and Eq. (4)]}
\end{aligned}$$

□

The following lemma states that the set of locations, only reachable from variables that do not share with any actual parameter at call time, cannot be affected by the

execution of a method. This will be important in the proof of soundness of the propagation rule for method call, since those variables cannot be made to reach or be reachable from other variables, during that call, as a side-effect.

LEMMA A.4. *Under the hypotheses of Proposition 4.7, 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 $\mathsf{L}^n_\sigma(x) = \mathsf{L}^n_{\sigma'}(x)$, let us prove that $\mathsf{L}^{n+1}_\sigma(x) = \mathsf{L}^{n+1}_{\sigma'}(x)$. By the first condition of Proposition 4.7, 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

$$\bigcup_{\ell \in \mathsf{L}^n_\sigma(x)} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) = \bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}). \tag{5}$$

We have

$$\begin{aligned}
\mathsf{L}^{n+1}_{\sigma'}(x) &= \mathsf{L}^n_{\sigma'}(x) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x)} (\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) && \text{(by Definition 4.2)}\\
&= \mathsf{L}^n_\sigma(x) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^n_\sigma(x)} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L}) && \text{(by hypothesis, Eq. (5) and Proposition 4.7 (1)),}\\
&= \mathsf{L}^{n+1}_\sigma(x). && \text{(by Definition 4.2).}
\end{aligned}$$

□

The following lemma shows that if a variable x is bound to the same location before and after a method is executed, then x reaches a location in $\mathcal{L}_\sigma$ before the method is executed if and only if it reaches the same location after the mehod is executed. This will be important for the proof of soundness of the propagation rules for method call, since it entails that no object and no array reachable from x is modified during the execution of the method, by making x reach a location, not reachable from the actual parameters, that was not already reachable from x before the call. Hence, the set of locations reachable from x can be modified, but only by adding locations the are not in $\mathcal{L}_\sigma$. That is, x might well become reachable or reach other variables, as a consequence of the execution of the method, as a side-effect, but only if such variables share with the actual parameters of the call.

LEMMA A.5. *Under the hypotheses of Proposition 4.7, 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 $\mathsf{L}^n_\sigma(x) \cap \mathcal{L}_\sigma = \mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma$, 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 4.7) 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, that is,

$$\forall \ell \in \mathsf{L}^n_{\sigma_1}(x) \setminus \mathcal{L}_\sigma.\mathsf{rng}(\mu_1(\ell).\phi) \cap \mathcal{L}_\sigma = \varnothing. \tag{6}$$

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 4.7, $\mu(\ell) = \mu'(\ell)$, which entails

$$\bigcup_{\ell \in \mathsf{L}^n_\sigma(x) \cap \mathcal{L}_\sigma} ((\mathsf{rng}(\mu(\ell)) \cap \mathbb{L}) \cap \mathcal{L}_\sigma) = \bigcup_{\ell \in \mathsf{L}^n_{\sigma'}(x) \cap \mathcal{L}_\sigma} ((\mathsf{rng}(\mu'(\ell)) \cap \mathbb{L}) \cap \mathcal{L}_\sigma). \tag{7}$$

We have

$$
\begin{aligned}
\mathsf{L}^{n+1}_{\sigma'}(x) \cap \mathcal{L}_\sigma &= \big(\mathsf{L}^{n}_{\sigma'}(x) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^{n}_{\sigma'}(x)}(\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L})\big) \cap \mathcal{L}_\sigma && \text{[By Definition 4.2]}\\
&= \big(\mathsf{L}^{n}_{\sigma'}(x) \cap \mathcal{L}_\sigma\big) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^{n}_{\sigma'}(x)}((\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma) && \text{[By distributivity]}\\
&= \big(\mathsf{L}^{n}_{\sigma'}(x) \cap \mathcal{L}_\sigma\big) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^{n}_{\sigma'}(x)\cap\mathcal{L}_\sigma}((\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma) && \text{[By Eq. (6)]}\\
&= \big(\mathsf{L}^{n}_{\sigma}(x) \cap \mathcal{L}_\sigma\big) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^{n}_{\sigma}(x)\cap\mathcal{L}_\sigma}((\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma) && \text{[By hyp., Eq. (7), Prop. 4.7 (1)]}\\
&= \big(\mathsf{L}^{n}_{\sigma}(x) \cap \mathcal{L}_\sigma\big) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^{n}_{\sigma}(x)}((\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L}) \cap \mathcal{L}_\sigma) && \text{[By Eq. (6)]}\\
&= \big(\mathsf{L}^{n}_{\sigma}(x) \cup \textstyle\bigcup_{\ell \in \mathsf{L}^{n}_{\sigma}(x)}(\mathsf{rng}(\mu'(\ell).\phi) \cap \mathbb{L})\big) \cap \mathcal{L}_\sigma && \text{[By distributivity]}\\
&= \mathsf{L}^{n+1}_{\sigma}(x) \cap \mathcal{L}_\sigma. && \text{[By Definition 4.2]}
\end{aligned}
$$

□

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 A.6) 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 A.7 and A.8). These lemmas are used for the proofs of Lemmas 5.14 and 5.18.

LEMMA A.6. *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 4.8, we have $\mathsf{t}' \in \mathsf{compatible}(\mathsf{t})$, and by Lemma 3.5, we have $\mathsf{t}'' \in \mathsf{compatible}(\mathsf{t})$, that is, $\mathsf{t}'' \in \mathsf{T}^0(\mathsf{t})$. 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 4.8, 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 4.8, $\mathsf{t}' \in \mathsf{compatible}(\mathsf{t}_1)$, where there exists $\kappa \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{K}$ and $\kappa'.f\colon \mathsf{t}_1 \in \mathbb{F}(\kappa)$, or there exists $\mathsf{t}_1[\,] \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{A}$.
Both cases, by Definition 4.8, are $\mathsf{compatible}(\mathsf{t}_1) \subseteq \mathsf{T}^i(\mathsf{t})$. 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 A.7. *Let* $\mathsf{t} \in \mathbb{T}$ *and* $i \geq 0$*. The set* $\mathsf{T}^i(\mathsf{t})$ *is closed with respect to* $\leq$*.*

PROOF. The set $\mathsf{compatible}(\mathsf{t}')$ is closed with respect to $\leq$ for every $\mathsf{t}' \in \mathbb{T}$. The thesis follows by induction on i and Definition 4.8. □

LEMMA A.8. *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,

$$
\begin{aligned}
\mathsf{T}^i(\mathsf{t}) &= \mathsf{T}^{i-1}(\mathsf{t}) \cup \bigcup_{\substack{\kappa \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{K}\\ \kappa'.f:\mathsf{t}'' \in \mathbb{F}(\kappa)}} \mathsf{compatible}(\mathsf{t}'') \cup \bigcup_{\mathsf{t}''[\,] \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{A}} \mathsf{compatible}(\mathsf{t}'') && \text{(by Definition 4.8),}\\
&\subseteq \mathsf{T}^{i-1}(\mathsf{t}') \cup \bigcup_{\substack{\kappa \in \mathsf{T}^{i-1}(\mathsf{t}') \cap \mathbb{K}\\ \kappa'.f:\mathsf{t}'' \in \mathbb{F}(\kappa)}} \mathsf{compatible}(\mathsf{t}'') \cup \bigcup_{\mathsf{t}''[\,] \in \mathsf{T}^{i-1}(\mathsf{t}') \cap \mathbb{A}} \mathsf{compatible}(\mathsf{t}'') && \text{(by hypothesis),}\\
&= \mathsf{T}^i(\mathsf{t}'). && \text{(By Definition 4.8)}
\end{aligned}
$$

□

B. REACHABILITY ANALYSIS

In this section, we show the soundness of our analysis. We first show that the concretization map introduced in Definition 5.2 is co-additive (Lemma B.1) and then

that the propagation rules corresponding to each type of arcs of our ACG are sound (Lemmas 5.14–5.20). These results are then used by Theorem 5.21, which shows the soundness of whole static analysis.

LEMMA B.1 (LEMMA 5.3). *Let $\tau \in \mathcal{T}$. The function γ_τ is co-additive.*

PROOF. Let $R_i \in \mathsf{A}_\tau$ for $i \geq 0$. We have

$$
\begin{aligned}
\gamma_\tau(\cap_{i\geq 0} R_i) &\overset{\text{Def. 5.2}}{=} \{\sigma \in \Sigma_\tau \mid \forall a, b \in \mathsf{dom}(\tau). a \rightsquigarrow^\sigma b \Rightarrow a \rightsquigarrow b \in \cap_{i\geq 0} R_i\} \\
&\overset{x \in \cap_i X_i \Leftrightarrow \wedge_i x \in X_i}{=} \{\sigma \in \Sigma_\tau \mid \forall a, b \in \mathsf{dom}(\tau). a \rightsquigarrow^\sigma b \Rightarrow \textstyle\bigwedge_{i\geq 0} a \rightsquigarrow b \in R_i\} \\
&\overset{y \Rightarrow \wedge_i x_i \Leftrightarrow \wedge_i y \Rightarrow x_i}{=} \{\sigma \in \Sigma_\tau \mid \forall a, b \in \mathsf{dom}(\tau). \textstyle\bigwedge_{i\geq 0}(a \rightsquigarrow^\sigma b \Rightarrow a \rightsquigarrow b \in R_i)\} \\
&\overset{\forall x \in X. \wedge_i f(x, y_i) \Leftrightarrow \wedge_i \forall x \in X. f(x, y_i)}{=} \{\sigma \in \Sigma_\tau \mid \textstyle\bigwedge_{i\geq 0}(\forall a, b \in \mathsf{dom}(\tau).(a \rightsquigarrow^\sigma b \Rightarrow a \rightsquigarrow b \in R_i))\} \\
&= \textstyle\bigcap_{i\geq 0}\{\sigma \in \Sigma_\tau \mid \forall a, b \in \mathsf{dom}(\tau). a \rightsquigarrow^\sigma b \Rightarrow a \rightsquigarrow b \in R_i\} \\
&= \textstyle\bigcap_{i\geq 0} \gamma_\tau(R_i).
\end{aligned}
$$

□

Lemma 5.14 states that in the case of the propagation rules of the sequential arcs, only nonexceptional 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 nonexceptional concrete states, and undefined on the exceptional ones.

LEMMA B.2 (LEMMA 5.14). *The propagation rules for the sequential arcs of Definition 5.4 are sound. That is, 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 nonexceptional execution, then, for every $R \in \mathsf{A}_\tau$, we have*

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

(We recall that ins is the semantics of ins*, see Figure 4).*

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 $R \in \mathsf{A}_\tau$ and a state $\omega' = \langle \rho', \mu' \rangle \in ins(\gamma_\tau(R)) \cap \Xi_{\tau'}$. We prove that $\omega' \in \gamma_{\tau'}(\Pi(R))$, that is, (Definition 5.2)

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

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 ins = load k 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 5.4, $\Pi(R) = R \cup R[l_k/s_j] \cup R_1$, 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.

(i) If $x, y \neq s_j$, then, $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 4.6,

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

(ii) 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 4.6,

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

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

(iii) 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 4.6,

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

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

(iv) If $x = y = s_j$, then $\rho'(x) = \rho(l_k)$ and $\rho'(y) = \rho(l_k)$. Hence, by Lemma 4.6,

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

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 5.4, $\Pi(R) = \{(a \rightsquigarrow b)[s_{j-1}/l_k] \mid a \rightsquigarrow b \in R \wedge a, b \neq l_k\}$. We distinguish the following cases.

(i) If $x, y \neq l_k$, then, $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 4.6,

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

and therefore $x \rightsquigarrow y \in \Pi(R)$.

(ii) 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 4.6,

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

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

(iii) 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 4.6,

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

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

(iv) If $x = y = l_k$, then $\rho'(x) = \rho'(y) = \rho'(l_k) = \rho(s_{j-1})$. Hence, by Lemma 4.6,

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

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

—$\mathsf{ins} = \mathsf{const}\,v$. 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 5.4, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$. We distinguish the following cases.

(i) 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 $x \not\rightsquigarrow^{\omega'} y$.

(ii) If $x, y \neq s_j$, then $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 4.6, $x \rightsquigarrow^{\omega'} y$ if and only if $x \rightsquigarrow^{\omega} y$, 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 5.4, $\Pi(R) = R \cup \{s_j \rightsquigarrow s_j\}$. We distinguish the following cases.

(i) If $x, y \neq s_j$, then, $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$, and for every $\ell \in \mathsf{dom}(\mu)$, $\mu'(\ell) = \mu(\ell)$. Hence, by Lemma 4.6,

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

(ii) If $x = y = s_j$, then $x \rightsquigarrow y = s_j \rightsquigarrow s_j \in \Pi(R)$.
(iii) If $x = s_j$ and $y \neq s_j$, then since ℓ is fresh, $\rho'(y) \notin \{\ell\} = \mathsf{L}_{\omega'}(x)$, hence $x \not\rightsquigarrow^{\omega'} y$.
(iv) 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 ins = getfield$\kappa.f$:t. 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 5.4,

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

We distinguish the following cases.

(i) If $x, y \neq s_{j-1}$, then, $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 4.6,

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

and therefore $x \rightsquigarrow y \in R_1 \subseteq \Pi(R)$.

(ii) If $x = s_{j-1}$ and $y \neq s_{j-1}$, if $s_{j-1} \rightsquigarrow^{\omega'} y$ then, by Lemma 4.10 and Definition 4.8, 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 A.8, $\mathsf{T}(\tau'(s_{j-1})) \subseteq \mathsf{T}(\mathsf{t})$, hence $\tau(y) \in \mathsf{T}(\tau'(s_{j-1})) \in \mathsf{T}(\mathsf{t})$, that is,

$$\mathsf{t} \rightsquigarrow \tau(y). \tag{8}$$

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

$$s_{j-1} \rightsquigarrow y \in R. \tag{9}$$

From Eqs. (8) and (9), we conclude that $x \rightsquigarrow y = s_{j-1} \rightsquigarrow y \in R_2 \subseteq \Pi(R)$.

(iii) If $y = s_{j-1}$, then if $x \rightsquigarrow^{\omega'} s_{j-1}$, by Definition 4.3 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 4.10, $x \rightsquigarrow^{\omega'} s_{j-1}$ implies that $\tau'(x) \rightsquigarrow \tau'(s_{j-1})$, and since $\tau'(s_{j-1}) \leq \mathsf{t}$, Lemma A.6 entails $\tau'(x) \rightsquigarrow \mathsf{t}$. If $x = s_{j-1}$, then by Definition 4.8, $\tau(s_{j-1}) \rightsquigarrow \tau'(s_{j-1}) \rightsquigarrow \mathsf{t}$, hence $\tau(s_{j-1}) \neq \mathtt{int}$, and it is obvious that s_{j-1} shares with itself at 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 4.6 entail:

$$\rho'(s_{j-1}) \in \mathsf{L}_{\omega'}(x) = \mathsf{L}_{\omega}(x). \tag{10}$$

Moreover, Definition 4.1 and the fact that $\rho'(s_{j-1}) \in \mathbb{L}$ ensure that

$$\rho'(s_{j-1}) = (\mu(\rho(s_{j-1})).\phi)(f) \in \mathsf{rng}(\mu(\rho(s_{j-1})).\phi) \cap \mathbb{L} \subseteq \mathsf{L}_{\omega}(s_{j-1}). \tag{11}$$

Thus, Eqs. (10) and (11) entail that $\mathsf{L}_{\omega}(x) \cap \mathsf{L}_{\omega}(s_{j-1}) \neq \varnothing$, that is, 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 ins $=$ putfield $\kappa.f:\mathsf{t}$. We have $L' = L$, $S' = S \setminus \{s_{j-2}, s_{j-1}\}$, $\rho' = \rho$, and $\mu' = \mu[(\mu(\rho(s_{j-2})).\phi)(\kappa.f\!:\!\mathsf{t}) \mapsto \rho(s_{j-1})]$. By Definition 5.4,

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

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

(i) If $x \rightsquigarrow^{\omega} y$, then $x \rightsquigarrow y \in R$, and since $x, y \notin \{s_{j-2}, s_{j-1}\}$, we have $x \rightsquigarrow y \in R_1 \subseteq \Pi(R)$.

(ii) 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 4.3, $\rho(s_{j-2}) \notin \mathsf{L}_{\omega}(x)$. 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 4.6, $x \rightsquigarrow^{\omega'} y$ entails $x \rightsquigarrow^{\omega} y$, a contradiction. Therefore, $x \rightsquigarrow^{\omega} s_{j-2}$ and $x \rightsquigarrow s_{j-2} \in R$. Since $\mu' = \mu[(\mu(\rho(s_{j-2})).\phi)(\kappa.f\!:\!\mathsf{t}) \mapsto \rho(s_{j-1})]$, we have, by Definition 4.2 and Lemma A.1,

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

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 $\rho(y) \in \mathsf{L}_{\omega}(s_{j-1})$, that is, $s_{j-1} \rightsquigarrow^{\omega} y$ 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 ins $=$ arraynew α. 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 5.4, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \neq s_{j-1}\} \cup \{s_{j-1} \rightsquigarrow s_{j-1}\}$. We distinguish the following cases.

(i) 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 4.6,

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

(ii) If $x = y = s_{j-1}$, then $x \rightsquigarrow y = s_{j-1} \rightsquigarrow s_{j-1} \in \Pi(R)$.

(iii) 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 $x \not\rightsquigarrow^{\omega'} y$.

(iv) 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 ins $=$ arraylength α. 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 5.4, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \neq s_{j-1}\}$. We distinguish the following cases:

(i) 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 4.6,

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

(ii) 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 ins $=$ arrayload t[ ]. Analogously to the case ins $=$ getfield $\kappa.f:\mathsf{t}$, we have $L' = L$, $S' = S \setminus \{s_{j-1}\}$, $\mu' = \mu$, 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 5.4,

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

We distinguish the following cases.

(i) If $x, y \neq s_{j-2}$, then $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$. Hence, by Lemma 4.6,

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

and therefore $x \rightsquigarrow y \in R_1 \subseteq \Pi(R)$.

(ii) If $x = s_{j-2}$ and $y \neq s_{j-2}$, if $s_{j-2} \rightsquigarrow^{\omega'} y$ then, by Lemma 4.10 and Definition 4.8, 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 A.8, $\mathsf{T}(\tau'(s_{j-2})) \subseteq \mathsf{T}(\mathsf{t})$, hence $\tau(y) \in \mathsf{T}(\tau'(s_{j-2})) \in \mathsf{T}(\mathsf{t})$, that is,

$$\mathsf{t} \rightsquigarrow \tau(y). \tag{12}$$

The locations reachable from an 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

$$s_{j-2} \rightsquigarrow y \in R. \tag{13}$$

From Eqs. (12) and (13) we conclude that $x \rightsquigarrow y = s_{j-2} \rightsquigarrow y \in R_2 \subseteq \Pi(R)$.

(iii) If $y = s_{j-2}$, then if $x \rightsquigarrow^{\omega'} s_{j-2}$, by Definition 4.3, we have $\rho'(s_{j-2}) = (\mu(\rho(s_{j-2})).\phi)(\rho(s_{j-1})) \in \mathbb{L}$, 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 4.10, $x \rightsquigarrow^{\omega'} s_{j-2}$ implies that $\tau'(x) \rightsquigarrow \tau'(s_{j-2})$, and since $\tau'(s_{j-2}) \leq \mathsf{t}$, Lemma A.6 entails $\tau'(x) \rightsquigarrow \mathsf{t}$. If $x = s_{j-2}$, then by Definition 4.8, $\tau(s_{j-2}) \rightsquigarrow \tau'(s_{j-2}) \rightsquigarrow \mathsf{t}$, hence $\tau(s_{j-2}) \neq \mathtt{int}$, and it is obvious that s_{j-2} shares with itself at 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 4.6 entails:

$$\rho'(s_{j-2}) \in \mathsf{L}_{\omega'}(x) = \mathsf{L}_{\omega}(x). \tag{14}$$

Moreover, Definition 4.1 and the fact that $\rho'(s_{j-2}) \in \mathbb{L}$ ensure that

$$\rho'(s_{j-2}) = (\mu(\rho(s_{j-2})).\phi)(\rho(s_{j-1})) \in \mathsf{rng}(\mu(\rho(s_{j-2})).\phi) \cap \mathbb{L} \subseteq \mathsf{L}_{\omega}(s_{j-2}). \tag{15}$$

Thus, Eqs. (14) and (15) entail that $\mathsf{L}_{\omega}(x) \cap \mathsf{L}_{\omega}(s_{j-2}) \neq \varnothing$, that is, 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 ins = arraystore t[ ]. Analogously to the case ins = putfield $\kappa.f$: t, we have $L' = L$, $S' = S \setminus \{s_{j-3}, s_{j-2}, s_{j-1}\}$, $\rho' = \rho$, and $\mu' = \mu[(\mu(\rho(s_{j-3})).\phi)(\rho(s_{j-2})) \mapsto \rho(s_{j-1})]$. By Definition 5.4,

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

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

(i) 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 $x \rightsquigarrow y \in R_1 \subseteq \Pi(R)$.

(ii) 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 4.3, $\rho(s_{j-3}) \notin \mathsf{L}_{\omega}(x)$. 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 4.6, $x \rightsquigarrow^{\omega'} y$ 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 4.2 and Lemma A.1,

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

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 $\rho(y) \in \mathsf{L}_{\omega}(s_{j-1})$, that is, $s_{j-1} \rightsquigarrow^{\omega} y$ 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 $x \rightsquigarrow y \in R_2 \subseteq \Pi(R)$.

—ins = 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 5.4, $\Pi(R) = R \cup R[s_{j-1} \mapsto s_j] \cup R_1$, 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.

(i) If $x, y \neq s_j$, then $\rho'(x) = \rho(x)$, $\rho'(y) = \rho(y)$. Hence, by Lemma 4.6,

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

(ii) 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 three 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 4.6,

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

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 4.6,

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

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 4.6,

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

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 5.4, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$. By Lemma 4.6, $x \rightsquigarrow^{\omega'} y$ if and only if $x \rightsquigarrow^{\omega} y$, 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 5.4, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$. Suppose that $x \rightsquigarrow^{\omega'} y$, then $x, y \neq s_{j-1}$. By Lemma 4.6, $x \rightsquigarrow^{\omega'} y$ if and only if $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$, and therefore $x \rightsquigarrow y \in \Pi(R)$, since $x, y \in \mathsf{dom}(\tau')$.

—If $\mathsf{ins} = \mathsf{inc}\ k\ x$. 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 5.4, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$. Suppose that $x \rightsquigarrow^{\omega'} y$, then $x, y \neq l_k$. By Lemma 4.6, $x \rightsquigarrow^{\omega'} y$ if and only if $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$, and therefore $x \rightsquigarrow y \in \Pi(R)$, since $x, y \in \mathsf{dom}(\tau')$.

—$\mathsf{ins} \in \{\mathsf{catch}, \mathsf{exception_is}\ K\}$. We have $L' = L$, $S' = S = \{s_0\}$, $\mu' = \mu$, and for every $a \in \mathsf{dom}(\tau')$, $\rho'(a) = \rho(a)$. By Definition 5.4, $\Pi(R) = \{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$. By Lemma 4.6, we have $x \rightsquigarrow^{\omega'} y$ if and only if $x \rightsquigarrow^{\omega} y$, and since $x, y \in \mathsf{dom}(\tau')$, it entails $x \rightsquigarrow y \in R \in \Pi(R)$. □

Moreover, we require that the propagation rules of the final arcs soundly approximate the concrete behavior of a final bytecode instruction (return t, return void, throw κ) of a method or a constructor belonging to the program under analysis. Similarly, the propagation rules of the exceptional arcs simulating the exceptional executions of the bytecode instructions which can throw an exception have to be sound. Lemmas 5.15 and 5.16 formalize these two facts, and we prove them in the following.

LEMMA B.3 (LEMMA 5.15). *The propagation rules for the final arcs of Definition 5.4 are sound. That is, consider a final arc from* ins *and its propagation rule* Π*, assume that* ins *has static type information* τ *at its beginning and* τ' *immediately after its execution (its nonexceptional execution if* ins *is a* return*, its exceptional execution if* ins *is a* throw κ*). Then, for every* $R \in \mathsf{A}_\tau$*, we have*

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

(We recall that ins is the semantics of ins*, see Figure 4.)*

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 $R \in \mathsf{A}_\tau$ and a state $\omega' = \langle \rho', \mu' \rangle \in ins(\gamma_\tau(R)) \cap \Xi_{\tau'}$. We prove that $\omega' \in \gamma_{\tau'}(\Pi(R))$, that is, (Definition 5.2)

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

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 final arcs.

—$\mathsf{ins} = \mathsf{return\ void}$. We have $L' = L$, $S' = \emptyset$, $\mu' = \mu$, and for every $a \in \mathsf{dom}(\tau')$, $\rho'(a) = \rho(a)$. By Definition 5.4, $\Pi(R) = \{a \rightsquigarrow b \mid a, b \notin S\}$. By Lemma 4.6, $x \rightsquigarrow^{\omega'} y$ if and only if $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$, and then $x \rightsquigarrow y \in \Pi(R)$ (since x and y are local variables).

—$\mathsf{ins} = \mathsf{return\ t}$. 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 5.4, $\Pi(R) = \{(a \rightsquigarrow b)[s_{j-1}/s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\}$. We consider the following cases.

(i) 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 4.6,

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

Therefore, since x and y are local variables, we conclude $x \rightsquigarrow y \in \Pi(R)$.

(ii) 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 4.6,

$$x \rightsquigarrow^{\omega'} s_0 \Leftrightarrow x \rightsquigarrow^{\omega} s_{j-1} \Rightarrow x \rightsquigarrow y \in R.$$

Therefore, since x is a local variable and $s_{j-1} \notin \{s_0, \ldots, s_{j-2}\}$, we conclude $x \rightsquigarrow y = x \rightsquigarrow s_0 = (x \rightsquigarrow s_{j-1})[s_{j-1}/s_0] \in \Pi(R)$.

(iii) 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 4.6,

$$s_0 \rightsquigarrow^{\omega'} y \Leftrightarrow s_{j-1} \rightsquigarrow^{\omega} y \Rightarrow x \rightsquigarrow y \in R.$$

Therefore, since $s_{j-1} \notin \{s_0, \ldots, s_{j-2}\}$ and y is a local variable, we conclude $x \rightsquigarrow y = s_0 \rightsquigarrow y = (s_{j-1} \rightsquigarrow y)[s_{j-1}/s_0] \in \Pi(R)$.

(iv) If $x = y = s_0$, then $\rho'(x) = \rho'(y) = \rho'(s_0) = \rho(s_{j-1})$. Hence, by Lemma 4.6,

$$s_0 \rightsquigarrow^{\omega'} s_0 \Leftrightarrow s_{j-1} \rightsquigarrow^{\omega} s_{j-1} \Rightarrow x \rightsquigarrow y \in R.$$

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 5.4, $\Pi(R) = \{(a \rightsquigarrow b)[s_{j-1}/s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\} \cup \{s_0 \rightsquigarrow s_0\}$. From Figure 4, 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, 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 $\mu' = \mu[\ell \mapsto npe]$, 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.

(i) If $x, y \neq s_0$, then, $\rho'(x) = \rho(x)$ and $\rho'(y) = \rho(y)$; hence, by Lemma 4.6, $x \rightsquigarrow^{\omega'} y$ if and only if $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$, and therefore $x \rightsquigarrow y \in \Pi(R)$ (since x and y are local variables).

(ii) 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 $x \not\rightsquigarrow^{\omega'} y$.

(iii) 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 $x \not\rightsquigarrow^{\omega'} y$.

(iv) If $x = y = s_0$, we have $s_0 \rightsquigarrow s_0 \in \Pi(R)$. □

LEMMA B.4 (LEMMA 5.16). *The propagation rules for the exceptional arcs of Definition 5.4 not leaving a* call *are sound. That is, 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' *after its exceptional execution. Then, for every* $R \in \mathsf{A}_\tau$, *we have*

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

(We recall that ins is the semantics of ins, *see Figure 4.)*

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 $R \in \mathsf{A}_\tau$ and a state $\omega' = \langle \rho', \mu' \in ins(\gamma_\tau(R)) \cap \Xi_{\tau'}$. We prove that $\omega' \in \gamma_{\tau'}(\Pi(R))$, that is, (Definition 5.2)

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

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 5.4, $\Pi(R) = \{a \rightsquigarrow b \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-1}\}\} \cup \{s_0 \rightsquigarrow s_0\}$. We distinguish the following cases.
  (i) 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 4.6, $x \rightsquigarrow^{\omega'} y$ if and only if $x \rightsquigarrow^{\omega} y$, which entails $x \rightsquigarrow y \in R$, and therefore $x \rightsquigarrow y \in \Pi(R)$ (since $x, y \in L'$).
  (ii) If $x = y = s_0$, we have $s_0 \rightsquigarrow s_0 \in \Pi(R)$.
  (iii) 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 $x \not\rightsquigarrow^{\omega'} y$.
  (iv) 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 $x \not\rightsquigarrow^{\omega'} y$.

—ins = throw κ. Analogously to the proof of Lemma 5.15 for throw κ, when $\rho'(s_0) = \ell$, where ℓ is a fresh location. □

Similarly, Lemma 5.17 shows that the propagation rules of the parameter passing arcs are sound. Namely, they soundly approximate the behavior of the *makescope* function.

Lemma B.5 (Lemma 5.17). *The propagation rules for the parameter passing arcs of Definition 5.4 are sound. That is, consider a parameter passing arc from a* call $m_1 \ldots m_n$ *to the first bytecode of* m_w*, for some* $w \in [1..k]$*, and its propagation rule* Π*. Assume that* call $m_1 \ldots m_n$ *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* $R \in \mathsf{A}_\tau$*, we have*

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

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contain local variables L and $j \geq \pi$ operand stack elements $S = \{s_0, \ldots, s_{j-\pi}, \ldots, s_{j-1}\}$, where π is the number of parameters of method m_w (including `this`). Then, $\mathsf{dom}(\tau') = \{l_0, \ldots, l_{\pi-1}\}$. Consider an arbitrary abstract element $R \in \mathsf{A}_\tau$ and a state $\omega' = \langle \rho', \mu' \rangle \in ins(\gamma_\tau(R)) \cap \Xi_{\tau'}$. We prove that $\omega' \in \gamma_{\tau'}(\Pi(R))$, that is, (Definition 5.2)

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

By the choice of ω', there exists $\omega = \langle \rho, \mu \rangle \in \gamma_\tau(R)$ such that $\omega' = (makescope\ m_w)(\omega)$. 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 5.4,

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

By Definition 3.16, 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 4.6,

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

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

The following lemmas deal with the return from a method call. Namely, in the case of a non-void method, the propagation rule of the return value arc expands the reachability approximation immediately after a call to that method with those reachability pairs related to the returned value. A method execution might also have side-effects on the memory, and this is captured by the propagation rule of the side-effects arcs. The reachability approximation after the call to the method is, therefore, determined as the union of the propagations of a return value arc (for non-void methods) and a side-effects arc, which is proved sound (Lemma 5.18 and Lemma 5.19).

LEMMA B.6 (LEMMA 5.18). *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*, that is, a partial function from a state at its beginning to the corresponding state at its end. Then, for every* $R_c \in \mathsf{A}_{\tau_c}$ *and* $R_e \in \mathsf{A}_{\tau_e}$*, 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 5.2,

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

In the following, we assume that $\mathsf{dom}(\tau_a) = L_a \cup S_a$, 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 the number of operand stack elements in $\mathsf{dom}(\tau_c)$ and the number of the parameters of method m, respectively. By Definition 3.18, σ_c, σ_e, and σ_q 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 is only possible when $\tau_e(s_0) = \mathtt{int}$. Since $\tau_q(s_{j-\pi}) = \tau_e(s_0) = \mathtt{int}$, 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 $\Pi^{\#19}$, one of the following hold.

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

(2) There is no $j - \pi \le p < j$ such that a might share with s_p at C.

(3) There exists a $j - \pi \le p < j$ 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 three cases.

(1) If $\tau_q(a) \not\rightsquigarrow \mathsf{t}$, then since $\tau_q(s_{j-\pi}) \le \mathsf{t}$ ($s_{j-\pi}$ contains the value returned by method m, whose type is t), by Lemma A.6, $\tau_q(a) \not\rightsquigarrow \tau_q(s_{j-\pi})$ holds which, by Lemma 4.10, implies $a \not\rightsquigarrow^{\sigma_q} s_{j-\pi}$.

(2) If there is no $j - \pi \le p < j$ such that a might share with s_p at C, that is, if at call-time a cannot share with the parameters of method m then, by Proposition 4.7, $\mathsf{L}_{\sigma_c}(a) \subseteq \mathcal{L}_{\sigma_c}$ 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 A.4, we have $\mathsf{L}_{\sigma_c}(a) = \mathsf{L}_{\sigma_q}(a)$. Hence, $\rho_q(s_{j-\pi}) \notin \mathsf{L}_{\sigma_c}(a) = \mathsf{L}_{\sigma_q}(a)$, that is, $a \not\leadsto^{\sigma_q} s_{j-\pi}$.

(3) If there exists a $j - \pi \le p < j$ such that a is definitely aliased to s_p at C, if no store $l_{p-j+\pi}$ occurs in m_w and $l_{p-j+\pi} \leadsto s_0 \notin R_e$, then we have the following.

$$\begin{array}{lll}
\rho_c(a) &=& \rho_c(s_p), \quad [a \text{ is definitely aliased to } s_p \text{ at C}],\\
\rho_c(s_p) &=& \rho_e(l_{p-j+\pi}), \quad [s_p \text{ at C corresponds to } l_{p-j+\pi} \text{ at E and there is no store } l_{p-j+\pi} \text{ in } m_w],\\
\rho_c(a) &=& \rho_q(a), \quad [a \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\} \text{ and Definition 3.18}],\\
\multicolumn{3}{l}{l_{p-j+\pi} \leadsto s_0 \notin R_e \Rightarrow l_{p-j+\pi} \not\leadsto^{\sigma_e} s_0, \quad [\text{By Definition 5.2}].}
\end{array}$$

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 4.6, $a \leadsto^{\sigma_q} s_{j-\pi} \Leftrightarrow l_{p-j+\pi} \leadsto^{\sigma_e} s_0$. Since $l_{p-j+\pi} \not\leadsto^{\sigma_e} s_0$, we conclude that $a \not\leadsto^{\sigma_q} s_{j-\pi}$.

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

Case C. If $a = s_{j-\pi}$ and $b \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\}$. If $s_{j-\pi} \leadsto b \notin R_q$ then, by rule $\Pi^{\#19}$, one of the following hold.

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

(2) There is no $j - \pi \le p < j$ such that $s_p \leadsto b \in R_c$.

(3) There exists a $j - \pi \le p < j$ such that b is definitely aliased to s_p at C and no store $l_{p-j+\pi}$ occurs in m_w and $s_0 \leadsto l_{p-j+\pi} \notin R_e$.

We analyze these three cases.

(1) If $\mathsf{t} \not\leadsto \tau_q(b)$, 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 $\tau_q(s_{j-\pi}) \le \mathsf{t}$. By Lemma A.8, $\mathsf{T}(\tau_q(s_{j-\pi})) \subseteq \mathsf{T}(\mathsf{t})$, which entails $\tau_q(b) \notin \mathsf{T}(\tau_q(s_{j-\pi}))$, that is, $\tau_q(s_{j-\pi}) \not\leadsto \tau_q(b)$, and by Lemma 4.10, $s_{j-\pi} \not\leadsto^{\sigma_q} b$.

(2) If there is no $j - \pi \le p < j$ such that $s_p \leadsto b \in R_c$, then by Definition 5.2, for each $j - \pi \le p < j$, $s_p \not\leadsto^{\sigma_c} b$, that is, $\rho_c(b) \notin \bigcup_{p \in [j-\pi, j)} \mathsf{L}_{\sigma_c}(s_p)$. By Proposition 4.7, $\rho_c(b) \in \mathcal{L}_{\sigma_c}$ and $\rho_q(s_{j-\pi}) \notin \mathcal{L}_{\sigma_c}$. By Lemma A.3, $\mathsf{L}_{\sigma_q}(s_{j-\pi}) \cap \mathcal{L}_{\sigma_c} = \varnothing$, hence $\rho_c(b) \notin \mathsf{L}_{\sigma_q}(s_{j-\pi})$ and, since $\rho_c(b) = \rho_q(b)$, we conclude that $\rho_q(b) \notin \mathsf{L}_{\sigma_q}(s_{j-\pi})$, that is, $s_{j-\pi} \not\leadsto^{\sigma_q} b$.

(3) If there exists a $j - \pi \le p < j$ such that b is definitely aliased to s_p at C, if no store $l_{p-j+\pi}$ occurs in m_w and $s_0 \leadsto l_{p-j+\pi} \notin R_e$, we have as follows.

$$\begin{array}{lll}
\rho_c(b) &=& \rho_c(s_p), \quad [b \text{ is definitely aliased to } s_p \text{ at C}],\\
\rho_c(s_p) &=& \rho_e(l_{p-j+\pi}), \quad [s_p \text{ at C corresponds to } l_{p-j+\pi} \text{ at E and there is no store } l_{p-j+\pi} \text{ in } m_w],\\
\rho_c(b) &=& \rho_q(b), \quad [b \in \mathsf{dom}(\tau_q) \setminus \{s_{j-\pi}\} \text{ and Definition 3.18}],\\
\multicolumn{3}{l}{s_0 \leadsto l_{p-j+\pi} \notin R_e \Rightarrow s_0 \not\leadsto^{\sigma_e} l_{p-j+\pi}, \quad [\text{By Definition 5.2}].}
\end{array}$$

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 4.6, $s_{j-\pi} \leadsto^{\sigma_q} b \Leftrightarrow s_0 \leadsto^{\sigma_e} l_{p-j+\pi}$. Since $s_0 \not\leadsto^{\sigma_e} l_{p-j+\pi}$, we conclude that $s_{j-\pi} \not\leadsto^{\sigma_q} b$.

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

Case D. If $a, b \in \mathsf{dom}(\tau_q) \smallsetminus \{s_{j-\pi}\}$. In this case, $\rho_c(a) = \rho_q(a)$ and $\rho_c(b) = \rho_q(b)$ (Definition 3.18). If $a \rightsquigarrow b \notin R_q$ then, by rule $\Pi^{\#20}$, one of the following hold.

(1) $[a \rightsquigarrow b \notin R_c$ and $\tau_q(a) \not\rightsquigarrow \tau_q(b)]$.
(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$]$.
(3) $[a \rightsquigarrow b \notin R_c$ and $\forall j - \pi \leq p_b < j$, $s_{p_b} \rightsquigarrow b \notin R_c]$.
(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 four cases.

(1) If $\tau_q(a) \not\rightsquigarrow \tau_q(b)$, then by Lemma 4.10, $a \not\rightsquigarrow^{\sigma_q} b$.
(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 last condition, we conclude that a does not share with any actual parameter of m at call time, and by Proposition 4.7, $\mathsf{L}_{\sigma_c}(a) \subseteq \mathcal{L}_{\sigma_c}$. By Lemma A.4, $\rho_c(a) = \rho_q(a)$ entails $\mathsf{L}_{\sigma_c}(a) = \mathsf{L}_{\sigma_q}(a)$. Since $a \rightsquigarrow b \notin R_c$, by Definition 5.2 $a \not\rightsquigarrow^{\sigma_c} b$, that is, $\rho_c(b) \notin \mathsf{L}_{\sigma_c}(a) = \mathsf{L}_{\sigma_q}(a)$. 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 last condition, we conclude that $\rho_c(b)$ is not a location reachable from the actual parameters of m at call time, and therefore, by Proposition 4.7, $\rho_c(b) \in \mathcal{L}_{\sigma_c}$. Since $a \rightsquigarrow b \notin R_c$, by Definition 5.2 $a \not\rightsquigarrow^{\sigma_c} b$, that is, $\rho_c(b) \notin \mathsf{L}_{\sigma_c}(a)$, and therefore $\rho_c(b) \notin \mathsf{L}_{\sigma_c}(a) \cap \mathcal{L}_{\sigma_c}$. By Lemma A.5, $\rho_c(a) = \rho_q(a)$ 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 $\rho_q(b) \notin \mathsf{L}_{\sigma_q}(a)$, that is, $b \not\rightsquigarrow^{\sigma_q} a$.
(4) In this case, for every $j - \pi \leq q_a, q_b < j$, we have as follows.

$$
\begin{array}{lll}
\rho_c(a) = \rho_c(s_{q_a}), & & [a \text{ is definitely aliased to } s_{j-\pi+q_a} \text{ at C}],\\
\rho_c(s_{q_a}) = \rho_e(l_{q_a-j+\pi}), & & [s_{q_a} \text{ at C corresponds to } l_{q_a-j+\pi} \text{ at E and no store } l_{q_a-j+\pi} \text{ occurs in } m_w],\\
\rho_c(a) = \rho_q(a), & & [a \in \mathsf{dom}(\tau_q) \smallsetminus \{s_{j-\pi}\} \text{ and Definition 3.18}],\\
\rho_c(b) = \rho_c(s_{q_b}), & & [b \text{ is definitely aliased to } s_{q_b} \text{ at C}],\\
\rho_c(s_{q_b}) = \rho_e(l_{q_b-j+\pi}), & & [s_{q_b} \text{ at C corresponds to } l_{q_b-j+\pi} \text{ at E and no store } l_{q_b-j+\pi} \text{ occurs in } m_w],\\
\rho_c(b) = \rho_q(b), & & [b \in \mathsf{dom}(\tau_q) \smallsetminus \{s_{j-\pi}\} \text{ and Definition 3.18}],\\
l_{q_a-j+\pi} \rightsquigarrow l_{q_b-j+\pi} \notin R_e \Rightarrow l_{q_a-j+\pi} \not\rightsquigarrow^{\sigma_e} l_{q_b-j+\pi}, & & [\text{By Definition 5.2}].
\end{array}
$$

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 4.6, $a \rightsquigarrow^{\sigma_q} b \Leftrightarrow l_{q_a-j+\pi} \rightsquigarrow^{\sigma_e} l_{q_b-j+\pi}$. 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 B.7 (LEMMA 5.19). *The propagation rule for the side-effects arcs is sound for* void *methods. 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* $\Pi^{\#20}$. *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*, that is, a partial function from a state at its beginning to the corresponding state at its end. Then, for every* $R_c \in \mathsf{A}_{\tau_c}$ *and* $R_e \in \mathsf{A}_{\tau_e}$*, we have*

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

PROOF. The proof is analogous to that of Case D of Lemma B.6. □

The following lemma deals with the the executions of a method that end up in an exception being thrown. Namely, the approximation of the reachability information at

the catch that runs from the exceptional states must consider the possible side-effects on the initial memory due to the execution of the method. This is the task of the propagation rules of the side-effects arcs. On the other hand, that approximation must also consider the case when the method is invoked on `null`. As in the previous case, the approximated reachability information must hence be consistent with both these situations and Lemma 5.20 shows it correct.

LEMMA B.8 (LEMMA 5.20). *The propagation rules for the exceptional arcs of the* call *and side-effects arcs are sound when a method call throws an exception. Namely, given nodes* $\mathsf{Q} = \boxed{\text{catch}}$, $\mathsf{C} = \boxed{\text{call } m_1 \dots m_n}$, *and* $\mathsf{E} = \boxed{\text{exception@}m_w}$, *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, *that is, a partial function from a state at its beginning to the corresponding state at its end. Then, for every* $R_c \in \mathsf{A}_{\tau_c}$ *and* $R_e \in \mathsf{A}_{\tau_e}$, *we have*

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

PROOF. Consider states $\sigma_c \in \gamma_{\tau_c}(R_c)$, $\sigma_e \in \gamma_{\tau_e}(R_e)$, and $\sigma_q = d((\textit{makescope } m_w)(\sigma)) \in \underline{\Xi}_{\tau_q}$ 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 5.2,

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

In the following, we assume that $\mathsf{dom}(\tau_a) = L_a \cup S_a$, 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 the number of the operand stack elements in $\mathsf{dom}(\tau_c)$ and the number of the parameters of method m, respectively. By Definition 3.18, σ_c, σ_e, and σ_q have to satisfy the following conditions: $\sigma_c = \langle\!\langle \mathsf{l}_c \parallel v_{j-1} :: \dots :: v_{j-\pi} :: \dots :: v_0 \rangle, \mu_c \rangle$, $\sigma_e = \underline{\langle\!\langle \mathsf{l}_e \parallel \ell \rangle, \mu_e \rangle}$, and $\sigma_q = \underline{\langle\!\langle \mathsf{l}_c \parallel \ell \rangle, \mu_e \rangle}$, where ℓ represents the location holding the exception thrown by m_w. Hence, $\tau_e(s_0) \leq$ `Throwable` and $\tau_q(s_0) \leq$ `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 this case, $a \rightsquigarrow b \in R_q$. We distinguish the following cases.

— $a = s_0$ and $b \neq s_0$. Since $s_0 \rightsquigarrow^{\sigma_q} x$ then, by Lemma 4.6 and Definition 4.8, $\tau_q(s_0) \rightsquigarrow \tau_q(b)$, that is, $\tau_q(b) \in \mathsf{T}(\tau_q(s_0))$. Since $\tau_q(s_0) \leq$ `Throwable` we have, by Lemma A.8, $\mathsf{T}(\tau_q(s_0)) \subseteq \mathsf{T}($`Throwable`$)$, and therefore $\tau_q(b) \in \mathsf{T}($`Throwable`$)$, that is, `Throwable`$\rightsquigarrow \tau_q(b)$. Moreover $x \notin S_q$ and hence $a \rightsquigarrow b = s_0 \rightsquigarrow b \in \Pi^{\#16}(R_c) \subseteq R_q$.

— $a \neq s_0$ and $b = s_0$. Since $a \rightsquigarrow^{\sigma_q} s_0$, then by Lemma 4.6, $\tau_q(a) \rightsquigarrow \tau_q(s_0)$. Moreover, $\tau_q(s_0) \leq$ `Throwable`, and by Lemma A.6, $\tau_q(a) \rightsquigarrow$`Throwable`. Since $a \notin S_q$, we have $a \rightsquigarrow b = a \rightsquigarrow s_0 \in \Pi^{\#16}(R_c) \subseteq R_q$.

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

— $a, b \in \mathsf{dom}(\tau_q) \setminus \{s_0\}$. In this case, we suppose that $a \rightsquigarrow b \notin R_q$ and show that $a \not\rightsquigarrow^{\sigma_q} b$. The proof is analogous to that of Case D of Lemma B.6. □

Finally, Theorem 5.21 shows that our reachability analysis is sound, that is, at each program point, the set of reachability pairs obtained by our analysis represents an over-approximation of the actual reachability information available at that point.

THEOREM B.9 (THEOREM 5.21). *Let* $\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi\rangle \Rightarrow^* \langle \boxed{\begin{smallmatrix}\mathsf{ins}\\ \mathit{rest}\end{smallmatrix}} \rightrightarrows^{b_1}_{b_m} \parallel \sigma\rangle :: a$ *be the execution of our operational semantics, from the block* $b_{\mathit{first}(\mathtt{main})}$ *starting with the first bytecode instruction of method* `main`*,* ins_0*, and an initial state* $\xi \in \Sigma_{\tau_0}$ *(containing no reachability except* `this` *that reaches itself and the* `args` *parameter that 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* $A \in \mathsf{A}_\tau$ *be the reachability approximation at* ins*, as computed by our reachability analysis. Then,* $\sigma \in \gamma_\tau(A)$ *holds.*

PROOF. The blocks in the configurations of an activation stack, except the topmost, cannot be empty and with no successor. This is because the configurations are only stacked by Rule (2) of Figure 5, 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 [Lindholm and Yellin 1999].

We proceed by induction on the length n of the execution $\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi\rangle \Rightarrow^* \langle \boxed{\begin{smallmatrix}\mathsf{ins}\\ \mathit{rest}\end{smallmatrix}} \rightrightarrows^{b_1}_{b_m} \parallel \sigma\rangle :: a$.

Base Case. If $n = 0$ the execution is just $\langle b_{\mathit{first}(\mathrm{main})} \parallel \xi$. In this case, $\tau_0 = \tau$ and $A_0 = A = S_{\mathit{first}(\mathtt{main})}$. Since ξ contains no reachability, except for `this` and `args` held in l_0 and l_1 that reach themselves, that is, $l_0 \rightsquigarrow^\xi l_0$, $l_1 \rightsquigarrow^\xi l_1$, and since our reachability analysis is a solution where $S_{\mathit{first}(\mathtt{main})} = \{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1\}$ (Definition 5.12), we have $\sigma = \xi \in \gamma_{\tau_0}(\{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1\}) \subseteq \gamma_\tau(S_{\mathit{first}(\mathtt{main})}) = \gamma_\tau(A_0) = \gamma_\tau(A)$.

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

$$\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi\rangle \Rightarrow^{n_p} \langle \overbrace{\boxed{\begin{smallmatrix}\mathsf{ins}_p\\ \mathit{rest}_p\end{smallmatrix}}}^{b_p} \rightrightarrows^{b'_1}_{b'_{m'}} \parallel \sigma_p\rangle :: a_p \Rightarrow^{n+1-n_p} \langle b_q \parallel \sigma_q\rangle :: a_q, \tag{16}$$

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 nonempty block b_p. Let such n_p be maximal. Given a bytecode ins_a, let τ_a and R_a be the static type information and the approximation of the reachability information at the ACG node $\boxed{\mathsf{ins}_a}$, respectively. By inductive hypothesis, we know that $\sigma_p \in \gamma_{\tau_p}(R_p)$, and we show that also $\sigma_q \in \gamma_{\tau_q}(R_q)$ 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 (16).

Rule (1). If this rule is applied, then $\mathit{ins}_p(\sigma_p)$ is defined, and ins_p is not a call. We distinguish the following cases.

case a: ins_p is not a return nor a throw.

If rest_p is nonempty, then by the maximality of n_p, Eq. (16) must be

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

Otherwise, $m' \geq 1$ (legal Java bytecode can only end with a return or a throw κ), and by the maximality of n_p, it must be $b_q = b'_h$ for a suitable $1 \leq h \leq m'$ so that Eq. (16) must have the form

$$\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\mathsf{ins}_p} \rightarrow b'_1 \cdots b'_{m'}}_{b_p} \parallel \sigma_p \rangle :: a_p$$

$$\stackrel{(1)}{\Rightarrow} \langle \boxed{} \rightarrow b'_1 \cdots b'_{m'} \parallel \overbrace{\mathit{ins}_p(\sigma_p)}^{\sigma_q} \rangle :: \overbrace{a_p}^{a_q} \stackrel{(6)}{\Rightarrow} \langle b_q \parallel \sigma_q \rangle :: a_q.$$

In both cases, the ACG contains a sequential arc from $\boxed{\mathsf{ins}_p}$ to $\boxed{\mathsf{ins}_q}$ and $R_q = \Pi(R_p)$, where Π is the propagation rule of that arc (Definition 5.4). We have

$$\begin{aligned}
\sigma_q &= \mathit{ins}_p(\sigma_p) & \\
&= \mathit{ins}_p(\sigma_p) \cap \Xi_{\tau_q} & [\sigma_q \in \Xi_{\tau_q}] \\
&\subseteq \mathit{ins}_p(\gamma_{\tau_p}(R_p)) \cap \Xi_{\tau_q} & [\text{By hypothesis and by monotonicity of } \mathit{ins}_p] \\
&\subseteq \gamma_{\tau_q}(\Pi(R_p)) = \gamma_{\tau_q}(R_q) & [\text{By Lemmas 5.14 and 5.16}].
\end{aligned}$$

case b: ins_p is a return 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 $\mathit{ins}_p(\sigma_p) \in \Xi$ (definition of return t), Eq. (16) must have one of the following two forms, depending on the emptiness of block b in Rule (4):

$$\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\text{return t}}}_{b_p} \parallel \underbrace{\langle\!\langle \mathsf{l}_p \parallel \mathit{top} :: \mathsf{s}_p \rangle, \mu_p \rangle\!\rangle}_{\sigma_p} :: \underbrace{\overbrace{\langle b_q \parallel \langle\!\langle \mathsf{l}_c \parallel \mathsf{s}_c \rangle, \mu_c \rangle\!\rangle}^{\mathtt{call\text{-}time}} :: a_q}_{a_p} \tag{17}$$

$$\stackrel{(1)}{\Rightarrow} \langle \boxed{} \parallel \langle\!\langle \mathsf{l}_p \parallel \mathit{top} \rangle, \mu_p \rangle\!\rangle :: a_p \stackrel{(4)}{\Rightarrow} \langle b_q \parallel \overbrace{\langle\!\langle \mathsf{l}_c \parallel \mathit{top} \rangle :: \mathsf{s}_c \rangle, \mu_p \rangle\!\rangle}^{\sigma_q} :: a_q,$$

or

$$\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\text{return t}}}_{b_p} \parallel \underbrace{\langle\!\langle \mathsf{l}_p \parallel \mathit{top} :: \mathsf{s}_p \rangle, \mu_p \rangle\!\rangle}_{\sigma_p} :: \underbrace{\overbrace{\langle \boxed{} \rightarrow b'_1 \cdots b'_{m'} \parallel \langle\!\langle \mathsf{l}_c \parallel \mathsf{s}_c \rangle, \mu_c \rangle\!\rangle}^{\mathtt{call\text{-}time}} :: a_q}_{a_p}$$

$$\stackrel{(1)}{\Rightarrow} \langle \boxed{} \parallel \langle\!\langle \mathsf{l}_p \parallel \mathit{top} \rangle, \mu_p \rangle\!\rangle :: a_p \stackrel{(4)}{\Rightarrow} \langle \boxed{} \rightarrow b'_1 \cdots b'_{m'} \parallel \langle\!\langle \mathsf{l}_c \parallel \mathit{top} :: \mathsf{s}_c \rangle, \mu_p \rangle\!\rangle :: a_q$$

$$\stackrel{(6)}{\Rightarrow} \langle b_q \parallel \langle\!\langle \mathsf{l}_c \parallel \mathit{top} :: \mathsf{s}_c \rangle, \mu_p \rangle\!\rangle :: a_q$$

where, in the latter case, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \leq h \leq m'$. We only prove the case for Eq. (17), the other being similar. Consider the configuration at `call-time`. Since only Rule (2) can stack configurations, `call-time` was the topmost one when the call was executed and, for a suitable $1 \leq w \leq n$, Eq. (17) must have the

following form

$$\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi \rangle \Rightarrow^{n_c} \langle \boxed{\begin{array}{c}\mathsf{call}\ \kappa.m_1\ldots\kappa.m_n\\ \mathsf{ins}_q\\ \mathit{rest}_q\end{array}} \begin{array}{l}\to b'_1\\ \cdots\\ \to b'_{m'}\end{array} \parallel \underbrace{\langle\!\langle \mathsf{l}_c \parallel v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0\rangle, \mu_c\rangle\!\rangle}_{\sigma_c} :: a_q$$

$$\stackrel{(2)}{\Rightarrow} \langle \mathit{first}(\kappa.m_w) \parallel \langle\!\langle [v_{j-\pi}\rangle\!\rangle :: \ldots :: v_{j-1}] \parallel \epsilon\rangle, \mu_c\rangle\!\rangle :: a_p$$

$$\Rightarrow^{n_p-n_c-1} \langle b_p \parallel \sigma_p\rangle :: a_p \stackrel{(1)}{\Rightarrow} \langle \boxed{} \parallel \langle\!\langle \mathsf{l}_p \parallel \mathit{top}\rangle, \mu_p\rangle\!\rangle :: a_p$$

$$\stackrel{(4)}{\Rightarrow} \langle b_q \parallel \sigma_q\rangle :: a_q,$$

where j is the number of operand stack elements before $\mathsf{ins}_C = \mathsf{call}\ \kappa.m_1\ldots\kappa.m_n$ is executed, π is the number of parameters of method m, and the rules in the portion $\Rightarrow^{n_p-n_c-1}$ never make the operand stack lower than at the beginning of that portion.

We consider $\sigma_c = \langle\!\langle \mathsf{l}_c \parallel v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0\rangle, \mu_c\rangle$ and $\sigma_p = \langle\!\langle \mathsf{l}_p \parallel \mathit{top} :: \mathsf{s}_p\rangle, \mu_p\rangle$. By inductive hypothesis for n_c and n_p, we know that $\sigma_c \in \gamma_{\tau_c}(R_c)$ and $\sigma_p \in \gamma_{\tau_p}(R_p)$. It is worth noting that in this case, $\sigma_q = d((\mathit{makescope}\ m_w)(\sigma_c))$, and since $\sigma_q \in \Xi_{\tau_q}$ and $\sigma_c \in \gamma_{\tau_c}(R_c)$, we have

$$\sigma_q \subseteq d((\mathit{makescope}\ m_w)(\gamma_{\tau_c}(R_c))) \cap \Xi_{\tau_q}. \tag{18}$$

Let $\sigma_e = \mathit{return}\ \mathsf{t}(\sigma_p) = \langle\!\langle \mathsf{l}_p \parallel \mathit{top}\rangle, \mu_p\rangle$. Then, the ACG contains a final arc from $\boxed{\mathsf{return\ t}}$ to $\boxed{\mathsf{exit@}m_w\mathsf{:t}}$ for a suitable $1 \le w \le n$, and $R_e = \Pi(R_p)$, where Π is the propagation rule #13 (Definition 5.4). The following relations hold.

$$\begin{aligned}\sigma_e &= \mathit{return}\ \mathsf{t}(\sigma_p)\\ &\subseteq \mathit{return}\ \mathsf{t}(\gamma_{\tau_p}(R_p)) && \text{[by hypothesis and monotonicity of } \mathit{return}\text{]},\\ &\subseteq \gamma_{\tau_e}(\Pi(R_p)) = \gamma_{\tau_e}(R_e) && \text{[by Lemma 5.15]}.\end{aligned}$$

In this case, there are two multi-arcs (a return value and a side-effect arc) going into $\boxed{\mathsf{ins}_q}$ (see Figure 12), and R_c and R_e represent the correct approximations of the reachability information at the sources of these arcs. Let $R_q = \Pi^{\#19}(R_c, R_e) \cup \Pi^{\#20}(R_c, R_e)$, where $\Pi^{\#19}$ and $\Pi^{\#20}$ are the propagation rules #19 and #20 introduced in Definition 5.4. By Eq. (18) and by Lemma 5.18, we have $\sigma_q \in \gamma_{\tau_q}(R_q)$.

case c: ins_p is a throw.

If rest_p is empty and $m' > 0$, the execution of Eq. (16) must have the form

$$\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\mathsf{throw}\ \kappa}}_{b_p} \begin{array}{l}\to b'_1\\ \cdots\\ \to b'_{m'}\end{array} \parallel \underbrace{\langle\!\langle \mathsf{l}_p \parallel e :: \mathsf{s}_p\rangle, \mu_p\rangle\!\rangle}_{\sigma_p} :: a_p$$

$$\stackrel{(1)}{\Rightarrow} \langle \boxed{} \begin{array}{l}\to b'_1\\ \cdots\\ \to b'_{m'}\end{array} \parallel \overbrace{\langle\!\langle \mathsf{l}_p \parallel e\rangle, \mu_p\rangle\!\rangle}^{\sigma_q} :: a_p \stackrel{(6)}{\Rightarrow} \langle b_q \parallel \sigma_q\rangle :: \overbrace{a_p}^{a_q},$$

where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \le h \le m'$. If rest_p is nonempty, the Execution of Eq. (16) must have the form

$$\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\begin{array}{c}\mathsf{throw}\ \kappa\\ \mathsf{catch}\\ \mathit{rest}_q\end{array}}}_{b_p} \begin{array}{l}\to b_1\\ \cdots\\ \to b_m\end{array} \parallel \underbrace{\langle\!\langle \mathsf{l}_p \parallel e :: \mathsf{s}_p\rangle, \mu_p\rangle\!\rangle}_{\sigma_p} :: a_p$$

$$\stackrel{(1)}{\Rightarrow} \langle \overbrace{\boxed{\begin{array}{c}\mathsf{catch}\\ \mathit{rest}_q\end{array}}}^{b_q} \begin{array}{l}\to b'_1\\ \cdots\\ \to b'_{m'}\end{array} \parallel \overbrace{\langle\!\langle \mathsf{l}_p \parallel e\rangle, \mu_p\rangle\!\rangle}^{\sigma_q} :: \overbrace{a_p}^{a_q},$$

since catch is the only bytecode whose semantics can be defined on the exceptional state σ_q. In both these cases, by inductive hypothesis, we have $\sigma_p \in \gamma_{\tau_p}(R_p)$, the ACG contains an exceptional arc from $\boxed{\text{throw } \kappa}$ to $\boxed{\text{catch}}$, and $R_q = \Pi(R_p)$, where Π is the propagation rule #15 (Definition 5.4). In this case, $\sigma_q \in \underline{\Xi}_{\tau_q}$, and we have

$$\begin{aligned}
\sigma_q &= \mathit{throw}\ \kappa(\sigma_p), \\
&= \mathit{throw}\ \kappa(\sigma_p) \cap \underline{\Xi}_{\tau_q} && [\sigma_q \in \underline{\Xi}_{\tau_q}], \\
&\subseteq \mathit{throw}\ \kappa(\gamma_{\tau_p}(R_p)) \cap \underline{\Xi}_{\tau_q} && [\text{by hypothesis and monotonicity of } \mathit{throw}], \\
&\subseteq \gamma_{\tau_q}(\Pi(R_p)) = \gamma_{\tau_q}(R_q) && [\text{by Lemma 5.16}].
\end{aligned}$$

If rest_p is empty and $m' = 0$, the execution of Eq. (16) must have one of these two forms, depending on the emptiness of block b in Rule (5):

$$\begin{aligned}
\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi \rangle &\Rightarrow^{n_p} \langle \underbrace{\boxed{\text{throw } \kappa}}_{b_p} \parallel \underbrace{\langle\!\langle \mathsf{l}_p \parallel e :: \mathsf{s}_p \rangle, \mu_p \rangle}_{\sigma_p} :: \underbrace{\overbrace{\langle b_q \parallel \langle\!\langle \mathsf{l}_c \parallel \mathsf{s}_c \rangle, \mu_c \rangle\!\rangle :: a_q}^{\mathtt{call\text{-}time}}}_{a_p} \\
&\overset{(1)}{\Rightarrow} \langle \boxed{} \parallel \underline{\langle\!\langle \mathsf{l}_p \parallel e \rangle, \mu_p \rangle\!\rangle} :: a_p \overset{(5)}{\Rightarrow} \langle b_q \parallel \overbrace{\underline{\langle\!\langle \mathsf{l}_c \parallel e \rangle, \mu_p \rangle\!\rangle}}^{\sigma_q} :: a_q,
\end{aligned} \tag{19}$$

or

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

where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \le h \le m'$. We only prove Eq. (19), the other being similar. Consider the configuration at `call-time`. Since only Rule (2) can stack configurations, `call-time` was the topmost one when the call was executed and Eq. (19) must have the following form

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

where j is the number of operand stack elements before $\text{ins}_C = \text{call } \kappa.m_1 \ldots \kappa.m_n$ is executed, π is the number of parameters of method m, and the rules in the portion $\Rightarrow^{n_p - n_c - 1}$ never make the operand stack lower than at the beginning of that portion. Since $\sigma_q \in \underline{\Xi}$, the only possibility for ins_q is to be a catch.

We 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 $\sigma_c \in \gamma_{\tau_c}(R_c)$ and $\sigma_p \in \gamma_{\tau_p}(R_p)$. It is worth noting that, in this case, $\sigma_q = d((\mathit{makescope}\ m_w)(\sigma_c))$, and since $\sigma_q \in \underline{\Xi}_{\tau_q}$ and $\sigma_c \in \gamma_{\tau_c}(R_c)$, we have

$$\sigma_q \subseteq d((\mathit{makescope}\ m_w)(\gamma_{\tau_c}(R_c))) \cap \underline{\Xi}_{\tau_q}. \tag{20}$$

Let $\sigma_e = throw\ \kappa(\sigma_p) = \underline{\langle\!\langle \mathsf{l}_p \parallel e\rangle, \mu_p\rangle}$. Then, the ACG contains a final arc from $\boxed{\text{throw } \kappa}$ to $\boxed{\text{exit@}m_w\text{:t}}$, for a suitable $1 \le w \le n$, $R_e = \Pi(R_p)$, where Π is the propagation rule #14 (Definition 5.4), and the following relations hold

$$
\begin{aligned}
\sigma_e &= throw\ \mathsf{t}(\sigma_p), \\
&= throw\ \mathsf{t}(\sigma_p) \cap \underline{\Xi}_{\tau_q} && [\sigma_q \in \underline{\Xi}_{\tau_q}], \\
&\subseteq throw\ \mathsf{t}(\gamma_{\tau_p}(R_p)) \cap \underline{\Xi}_{\tau_1} && [\text{by hypothesis and monotonicity of } throw], \\
&\subseteq \gamma_{\tau_e}(\Pi(R_p)) = \gamma_{\tau_e}(R_e) && [\text{by Lemma 5.15}].
\end{aligned}
$$

In this case, there are two arcs (a side-effect and an exceptional arc) going into $\boxed{\text{catch}}$ (see Figure 13), and R_c and R_e represent the correct approximations of the reachability information at the sources of these arcs. Let $R_q = \Pi^{\#16}(R_c) \cup \Pi^{\#20}(R_c, R_e)$, where $\Pi^{\#16}$ and $\Pi^{\#20}$ are the propagation rules #16 and #20 introduced in Definition 5.4. By Eq. (20) and by Lemma 5.20, we have $\sigma_q \in \gamma_{\tau_q}(R_q)$.

Rule (2). By definition of *makescope*, Eq. (16) must have the form

$$
\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi\rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\text{call } \kappa.m_1 \ldots \kappa.m_n}}_{b_p} \rightarrow b'_1 \cdots b'_{m'} \parallel \underbrace{\langle\!\langle \mathsf{l}_p \parallel v_{j-1}\rangle :: \ldots :: v_{j-\pi} :: \ldots :: v_0\rangle\!\rangle, \mu_p}_{\sigma_p} :: a_p
$$

$$
\stackrel{(2)}{\Rightarrow} \overbrace{\langle \mathit{first}(\kappa.m_w)\rangle}^{b_q} \parallel \overbrace{\langle\!\langle [v_{j-\pi} :: \ldots :: v_{j-1}] \parallel \epsilon\rangle, \mu_p\rangle\!\rangle}^{\sigma_q} :: a_q,
$$

where j is the number of operand stack elements before call $\kappa.m_1 \ldots \kappa.m_n$ is executed, and π is the number of parameters of method m. In this case, the ACG contains a parameter passing arc from $\boxed{\text{call } \kappa.m_1 \ldots \kappa.m_n}$ to $\boxed{\mathit{first}(\kappa.m_w)}$ for a suitable $w \in [1..n]$ and $R_q = \Pi(R_p)$, where Π is the propagation rule #18 (Definition 5.4). We have

$$
\begin{aligned}
\sigma_q &= makescope(\sigma_p), \\
&\subseteq makescope(\gamma_{\tau_p}(R_p)) && [\text{by hypothesis and monotonicity of } makescope], \\
&\subseteq \gamma_{\tau_q}(\Pi(R_p)) = \gamma_{\tau_q}(R_q) && [\text{by Lemma 5.17}].
\end{aligned}
$$

Rule (3). Let i and j be the number of local variables and operand stack elements before call $\kappa.m_1 \ldots \kappa.m_n$ is executed, and π be the number of parameters of method m. In this case, Eq. (16) must have the form

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

when $rest_p$ is nonempty, while otherwise it has the form

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

where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \le h \le m'$. In both cases, the ACG contains an exceptional arc from $\boxed{\mathsf{ins}_p}$ to $\boxed{\mathsf{ins}_q}$, and $R_q = \Pi^{\#16}(R_p)$, where $\Pi^{\#16}$ is the propagation rule #16 introduced in Definition 5.4.

Suppose that $\sigma_p = \langle \rho_p, \mu_p \in \Xi_{\tau_p}$ and $\sigma_q = \langle \rho_q, \mu_q \in \Xi_{\tau_q}$, where τ_p and τ_q are such that $\mathsf{dom}(\tau_p) = \{l_0, \ldots, l_{i-1}, s_0, \ldots, s_{j-1}\}$, while $\mathsf{dom}(\tau_q) = \{l_0, \ldots, l_{i-1}, s_0\}$. We have that for each $r \in [0, i)$, $\rho_q(l_r) = \rho_p(l_r)$, while $\rho_q(s_0) = \ell \in \mathbb{L}$, where ℓ is a fresh location and $\mu_q = \mu_p[\ell \mapsto o]$, where o is a new instance of `Throwable`. Moreover, by Definition 5.4,

$$\Pi^{\#16}(R_p) = \{a \rightsquigarrow b \in R_p \mid a, b \in \{l_0, \ldots, l_{i-1}\} \cup \{s_0 \rightsquigarrow s_0\} \cup \underbrace{\{a \rightsquigarrow s_0 \mid a \in L \wedge \tau(a) \rightsquigarrow \mathtt{Throwable}\}}_{R_1} \cup \underbrace{\{s_0 \rightsquigarrow a \mid a \in L \wedge \mathtt{Throwable} \rightsquigarrow \tau(a)\}}_{R_2}.$$

We must prove that $\sigma_q \in \gamma_{\tau_q}(\Pi^{\#16}(R_p))$, that is, (Definition 5.2) that is,

$$\text{for every } x, y \in \mathsf{dom}(\tau_q),\ x \rightsquigarrow^{\sigma_q} y \Rightarrow x \rightsquigarrow y \in \Pi^{\#16}(R_p).$$

The latter can be proved by showing that either $x \not\rightsquigarrow^{\sigma_q} y$ or $x \rightsquigarrow y \in \Pi^{\#16}(R_p)$. Note that by hypothesis, $\sigma_p \in \gamma_{\tau_p}(R_p)$, that is, for every $x, y \in \mathsf{dom}(\tau_p)$, $x \rightsquigarrow^{\sigma_p} y \Rightarrow x \rightsquigarrow y \in R_p$. Let x and y be arbitrary variables from $\mathsf{dom}(\tau_q)$. We distinguish the following cases.

—If $x, y \in \{l_0, \ldots, l_{i-1}\}$, then $\rho_q(x) = \rho_p(x)$ and $\rho_q(y) = \rho_p(y)$; hence, by Lemma 4.6, $x \rightsquigarrow^{\sigma_q} y$ if and only if $x \rightsquigarrow^{\sigma_p} y$, which entails $x \rightsquigarrow y \in R_p$, and therefore $x \rightsquigarrow y \in \Pi^{\#16}(R_p)$.
—If $x = s_0$ and $y \ne s_0$, then if there exists a variable $z \in \{l_0, \ldots, l_{i-1}\}$ such that $s_0 \rightsquigarrow^{\sigma_q} z$, then by Lemma 4.10, $\tau_q(s_0) \rightsquigarrow \tau_q(z) = \tau_p(z)$, that is, $\tau_p(z) \in \mathsf{T}(\tau_q(s_0))$. Moreover, $\tau_q(s_0) \le \mathtt{Throwable}$, hence by Lemma A.8, $\mathsf{T}(\tau(s_0)) \subseteq \mathsf{T}(\mathtt{Throwable})$, which entails $\tau(z) \in \mathsf{T}(\mathtt{Throwable})$, that is, $\mathtt{Throwable} \rightsquigarrow \tau_p(z)$. Hence, $s_0 \rightsquigarrow z \in R_2$.
—If $x \ne s_0$ and $y = s_0$, then if there exists a variable $z \in \{l_0, \ldots, l_{i-1}\}$ such that $z \rightsquigarrow^{\sigma_q} s_0$, then by Lemma 4.10, $\tau_p(z) = \tau_q(z) \rightsquigarrow \tau_q(s_0)$, that is, $\tau_q(s_0) \in \mathsf{T}(\tau_p(z))$. Moreover, $\tau_q(s_0) \le \mathtt{Throwable}$, that is, $\mathtt{Throwable} \in \mathsf{compatible}(\tau_q(s_0))$, hence by Lemma A.6, $\tau_p(z) \rightsquigarrow \mathtt{Throwable}$. Hence, $z \rightsquigarrow s_0 \in R_1$.
—If $x = y = s_0$, then since $\tau_q(s_0) \in \mathbb{K}$, $x \rightsquigarrow^{\sigma_q} y$. Moreover, $s_0 \rightsquigarrow s_0 \in \Pi^{\#16}(R_p)$.

Therefore, $\sigma_q \in \gamma_{\tau_q}(\Pi^{\#16}(R_p))$. $\square$

REFERENCES

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. 2007. Cost analysis of Java bytecode. In *Proceedings of the 16th European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 4421, Springer, Berlin, 157–172.

Balaban, I., Pnueli, A., and Zuck, L. D. 2005. Shape analysis by predicate abstraction. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*. Lecture Notes in Computer Science, vol. 3385, Springer, 164–180.

Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the 22nd Conference on Programming Language Design and Implementation (PLDI)*. Vol. 36, ACM, New York, 203–213.

Ball, T., Millstein, T., and Rajamani, S. K. 2005. Polymorphic predicate abstraction. *ACM Trans. Program. Lang. Syst. (TOPLAS) 27*, 314–343.

Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., and Yang, H. 2007. Shape analysis for composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 4590, Springer, 178–192.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput. 8,* 35, 677–691.

CALCAGNO, C., DISTEFANO, D., O'HEARN, P., AND YANG, H. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 289–300.

CHATTERJEE, S., LAHIRI, S., QADEER, S., AND RAKAMARIC, Z. 2009. A low-level memory model and an accompanying reachability predicate. *Int. J. Softw. Tools Technol. Transfer 11,* 2, 105–116.

CORBETT, J. C. 2000. Using shape analysis to reduce finite-state models of concurrent Java programs. *ACM Trans. Softw. Eng. Methodo. 9,* 1, 51–93.

COUSOT, P. AND COUSOT, R. 1977. 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)*. ACM, 238–252.

COUSOT, P. AND COUSOT, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th Symposium on Principles of Programming Languages (POPL)*. ACM, 269–282.

DAMS, D. AND NAMJOSHI, K. S. 2003. Shape analysis through predicate abstraction and model checking. In *Proceedings of the 4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*. Springer, Berlin, 310–324.

DISTEFANO, D., O'HEARN, P., AND YANG, H. 2006. A local shape analysis based on separation logic. In *Proceedings of the 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, vol. 3920, Springer, 287–302.

GENAIM, S. AND ZANARDINI, D. 2010. The acyclicity inference of COSTA. In *Proceedings of the International Workshop on Termination (WST)*. Edinburgh.

GENAIM, S. AND ZANARDINI, D. 2012. Reachability-based acyclicity analysis by abstract interpretation. *Theoretical Comput. Sci. 474*, 25, 60–79.

HARDEKOPF, B. C. 2009. Pointer analysis: Building a foundation for effective program analysis. Ph.D. thesis, University of Texas, Austin.

HIND, M. 2001. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, New York, 54–61.

JUMP, M. AND MCKINLEY, K. S. 2009. Dynamic shape analysis via degree metrics. In *Proceedings of the 8th International Symposium on Memory Management (ISMM)*. H. Kolodner and G. L. J. Steele, Eds., ACM, 119–128.

LHOTÁK, O. 2006. Program analysis using binary decision Diagrams. Ph.D. thesis, McGill University.

LHOTÁK, O. AND CHUNG, K.-C. A. 2011. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Symposium on Principles of Programming Languages (POPL)*. ACM, 3–16.

LHOTÁK, O. AND HENDREN, L. 2003. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction*. Lecture Notes in Computer Science, vol. 2622. Springer, Berlin, 153–169.

LINDHOLM, T. AND YELLIN, F. 1999. *The Java^{TM} Virtual Machine Specification* 2nd Ed. Addison-Wesley.

MARRON, M., HERMENEGILDO, M. V., KAPUR, D., AND STEFANOVIC, D. 2008. Efficient context-sensitive shape analysis with graph based heap models. In *Proceedings of the 17th International Conference on Compiler Construction (CC)*. L. J. Hendren, Ed., Lecture Notes in Computer Science, vol. 4959, Springer, 245–259.

NELSON, G. 1983. Verifying reachability invariants of linked structures. In *Proceedings of the 8th Symposium on Principles of Programming Languages (POPL)*. 38–47.

NIKOLIĆ, Đ. 2013. A general framework for constraint-based static analyses of Java bytecode programs. Ph.D. thesis, University of Verona.

NIKOLIĆ, Đ. AND SPOTO, F. 2012a. Automaton-based array initialization analysis. In *Proceedings of the 6th International Conference on Language and Automata Theory and Applications (LATA'12)*. Lecture Notes in Computer Science, vol. 7183. Springer, Berlin, 420–432.

NIKOLIĆ, Đ. AND SPOTO, F. 2012b. Definite expression aliasing analysis for Java bytecode. In *Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing (ICTAC'12)*. Lecture Notes in Computer Science, vol. 7521, Springer-Verlag, Berlin, 74–89.

NIKOLIĆ, Đ. AND SPOTO, F. 2012c. Reachability analysis of program variables. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR'12)*. Lecture Notes in Artificial Intelligence, vol. 7364, Springer-Verlag, Berlin, 423–438.

NIKOLIĆ, Đ. AND SPOTO, F. 2013. Inferring complete initialization of arrays. *Theor. Comput. Sci. 484*, 16–40.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1991. Object-oriented type inference. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages & Applications (OOPSLA)*. ACM SIGPLAN Notices, vol. 26, 11, ACM, 146–161.

PAPI, M. M., ALI, M., CORREA, T. L., PERKINS, J. H., AND ERNST, M. D. 2008. Practical pluggable types for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 201–212.

Payet, É. and Spoto, F. 2007. Magic-sets transformation for the analysis of Java bytecode. In *Proceedings of the 14th International Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 4634, Springer, 452–467.

Pheng, S. and Verbrugge, C. 2005. Dynamic shape and data structure analysis in Java. Tech. rep., School of Computer Science, McGill University.

Rossignoli, S. and Spoto, F. 2006. Detecting non-cyclicity by abstract compilation into boolean functions. In *Proceedings of the 7th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*. Lecture Notes in Computer Science, vol. 3855, Springer, 95–110.

Rountev, A., Milanova, A., and Ryder, B. G. 2001. Points-to analysis for Java using annotated constraints. In *Proceedings of the 16th ACM Conference on of Object-Oriented Programming: Systems, Languages & Applications (OOPSLA)*. ACM, 43–55.

Sagiv, M., Reps, T., and Wilhelm, R. 1998. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst. 20*, 1–50.

Sagiv, M., Reps, T., and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst. 24*, 217–298.

Salcianu, A. D. 2006. Pointer analysis for Java programs: Novel techniques and applications. Ph.D. thesis, MIT, Cambridge, MA.

Secci, S. and Spoto, F. 2005. Pair-sharing analysis of object-oriented programs. In *Proceedings of the 12th International Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 3672, Springer, 320–335.

Smaragdakis, Y., Bravenboer, M., and Lhoták, O. 2011. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Symposium on Principles of Programming Languages (POPL)*. ACM, 17–30.

Spoto, F. 2008. Nullness analysis in boolean form. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods*. IEEE, Los Alamitos, CA, 21–30.

Spoto, F. 2011. Precise null-pointer analysis. *Softw. Syst. Model. 10,* 2, 219–252.

Spoto, F. and Ernst, M. D. 2011. Inference of field initialization. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM, 231–240.

Spoto, F., Mesnard, F., and Payet, E. 2010. A termination analyzer for Java bytecode based on path-length. *ACM Trans. Program. Lang. Syst. 32,* 3, 1–70.

Received May 2012; revised January, July 2013; accepted August 2013