Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

Definite Expression Aliasing in Java Bytecode Programs: a Constraint-based Static Analysis

Đurica Nikolić · Fausto Spoto

Received: date / Accepted: date

Abstract We define a novel static analysis for Java bytecode, called *definite expression aliasing*. It infers, for each variable v at each program point p, a set of expressions whose value at p is equal to the value of v at p, for every possible execution of the program. Namely, it determines which expressions *must* be aliased to local variables and stack elements of the Java Virtual Machine. This must aliasing or must equality is a useful piece of information for an inter-procedural static analyzer, such as Julia, since it can be used to refine other analyses at conditional statements or assignments. We formalize and implement a *constraint-based* analysis, based on and proved correct in the abstract interpretation framework. Moreover, we show the benefits of our definite expression aliasing analysis for nullness and termination analyses with Julia.

Keywords Definite aliasing expressions · Must aliasing · Static analysis · Abstract interpretation · Java bytecode

1 Introduction

Static analyses infer properties of computer programs and prove the absence of some classes of bugs inside those programs. Modern programming languages are, however, very complex. Static analysis must cope with that complexity and remain precise enough to be of practical interest. This is particularly true for low-level languages such as Java bytecode [23], whose instructions operate on stack and local variables, which are typically aliased to expressions. Consider, for instance, the method `onOptionsItemSelected` in Fig. 1, taken from the Google's `HoneycombGallery` Android application. The statement `if (mCamera!=null)` at line 4 is compiled into the following bytecode instructions:

```
aload_0
getfield mCamera:Landroid/hardware/Camera;
```

Đ. Nikolić
Chair of Software Engineering, ETH Zurich
E-mail: durica.nikolic@inf.ethz.ch

F. Spoto
Dipartimento di Informatica, University of Verona
E-mail: fausto.spoto@univr.it

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

Fig. 1 A method of the `CameraFragment` class by Google

```
ifnull [go to the else branch]
[then branch]
```

Bytecode `ifnull` checks whether the topmost variable of the stack, *top*, is `null` and passes control to the opportune branch. A static analysis that infers non-`null` variables can, therefore, conclude that *top* is non-`null` at the `[then branch]`. But this information is irrelevant: *top* gets consumed by the `ifnull` and disappears from the stack. It is, instead, much more important to know that *top* was a definite alias of the field `mCamera` of local 0, i.e., of `this.mCamera`, because of the previous two bytecodes (local 0 stands for `this`). That observation is important at the subsequent call to `mCamera.stopPreview()` at line 5, since it allows us to conclude that `this.mCamera` is still non-`null` there: line 5 is part of the then branch starting at line 4 and we proved that *top* (definitely aliased to `this.mCamera`) is non-`null` at that point. Note that here the source of complexity is not the presence of the operand stack: it is rather the fact the field of a variable (local variable or stack element, this is irrelevant) is first checked for non-nullness and then used for subsequent computations. A code transformation into three-address code or similar would not change anything in this case.

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

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

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

The previous examples show the importance of definite expression aliasing analysis for nullness analysis. However, the former is useful for other analyses as well. For instance, consider the termination analysis of a loop whose upper bound is the return value of a function call:

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

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

These examples show that it is important to know which expressions are *definitely* aliased to stack and local variables of the Java Virtual Machine (JVM) at a given program point. In this article, we introduce a static analysis called *definite expression aliasing analysis*, which provides, for each program point p and each variable v, a set of expressions E such that the values of E and v at point p coincide, for every possible execution path. We call these expressions *definite expression aliasing information*. In general, we want to deal with relatively complex expressions (e.g., a field of a field of a variable, the return value of a method call, possibly non-pure, and so on), dealing with values in the heap memory, hence shared or affected by methods side-effects and field updates. We show, experimentally, that our aliasing analysis supports nullness and termination analyses of our tool Julia, but this paper is only concerned with the expression aliasing analysis itself. Moreover, we prove our analysis sound.

We analyze Java bytecode directly, without relying on any intermediate representation. This avoids the definition of the translation from Java bytecode (with all its intricacies) into the intermediate language and the proof of correctness for this translation. When a warning is issued by the analyzer, it can be immediately reported to the user in terms of the original code. Moreover, the theoretical analysis matches its implementation, that is the Julia analyzer for Java bytecode. That analyzer is the result of more than ten years of development and we want to provide new analyses for it instead of developing analyses for a different framework. It is true however that an intermediate representation, in terms for instance of three-address code, would remove the stack variables and simplify the formalization. However, the complexity that we faced for this analysis and is reflected in our proofs of correctness and in the implementation is related to the fact that we define a static analysis for properties dealing with the heap memory and hence with all kinds of side-effects. An intermediate representation that makes stack variables disappear would not change anything at that.

We opt for a semantical analysis rather than simple syntactical checks. For instance, in Fig. 1, the result of the analysis must not change if we introduce a temporary variable `temp = this.mCamera` and then check whether `temp != null`: it is still the value of `this.mCamera` that is compared to `null` there. Moreover, since we analyze Java bytecode, a semantical approach is important in order to be independent from the specific compilation style of high-level expressions and be able to analyze obfuscated code (for instance, malware) or code not decompilable into Java (for instance, not organized into scopes).

1.1 Overview of our Static Analysis

In the following sections, we give a detailed description of our analysis. Here, we only provide a short overview of how the analysis works, in order to give a preliminary global picture of the technique.

Our analysis starts by the translation of the Java bytecode from its compiled form, usually contained in a set of `jar` files, into a graph of basic blocks, where each method or constructor gives rise to a subgraph. This is the control-flow graph of the program, from which an abstract graph is built. Nodes of the latter do not contain any bytecode instruction, but on the other hand, its arcs are decorated with *propagation rules*, expressing how the aliasing information at a program point flows into the aliasing information at each subsequent program point, on the basis of the bytecode instructions that occurred in the control-flow graph. These propagation rules have been derived and proved correct through abstract interpretation [12]. Hence, the aliasing information is propagated, by applying the propagation rules, until a fixpoint is reached. Since we want a definite analysis, the information at a program point is the intersection of the information that reaches that point along all possible incoming arcs. When the fixpoint is reached, the approximation at each program point is the result of the static analysis.

The generation of the control-flow graph is done through the traditional construction of the graph of basic blocks [3] adapted to Java bytecode, so we do not explain it in detail. On the other hand, the heart of our technique is the definition of the propagation rules, in particular of those dealing with bytecode instructions that might have side-effects on the heap, namely, field update and method calls. The fixpoint of the abstract graph can be computed with any technique. We use an iterative propagation algorithm.

1.2 Related Work

Alias analysis belongs to the large group of pointer analyses [19], and its task is to determine whether a memory location can be accessed in more than one way. There exist two types of alias analyses: possible (may) and definite (must). The former detects those pairs of variables that might point to the same memory location. There are very few tools performing this analysis on Java programs (e.g., `WALA` [2], `soot` [1], `JAAT` [30]). The later under-approximates the actual aliasing information and, to the best of our knowledge, the analysis introduced in this article is the first of this type dealing with Java bytecode programs and providing expressions aliased to variables. Similarly to our approach, the authors of [15] deal with definite aliasing, but their *must-aliasing* information is used for other goals and they do not deal with aliasing expressions.

The idea of abstract interpretation-based static analysis is not new, and since the paper introducing that technique [12] a lot of static analyses for both numerical and heap-related properties as well as a lot of static analysis tools based on abstract interpretation have been presented. Constraint-based approaches have been widely used for different types of program analyses. For instance, Gulwani et al. [18] introduce a constraint-based approach for discovering invariants involving linear inequalities, and generate weakest preconditions and strongest postconditions over the abstraction of linear arithmetic. Their constraints are Boolean combinations of quadratic inequalities over integer variables, and therefore their approach (although constraint-based) and goals are completely different from ours. Some other constraint-based techniques, successfully applied to the problem of discovering linear arithmetic invariants or inductive loop invariants for the verification of assertions, are for example [11,36,10].

Another very frequent application of constraints is for the problem of type inference [31, 5,9,42]. Usually, these techniques assume programs completely untyped, and then extract the constraints from the program text, represent them in the form of a directed graph, which is then solved by different algorithms similar to transitive closure. The abstract constraint graphs introduced in this article are similar to the trace graphs introduced by Palsberg and Schwartzbach in [31].

Nielson et al. [27, Chapter 3] presents a technique for constraint-based analysis of a simple functional language. Namely, the authors define a control flow analysis by developing a syntax directed specification of the problem followed by its constraint-based formulation and finally they show how these constraints can be solved. An overview of constraint-based program analyses is provided by Aiken in [4]. He underlines that this kind of analyses can be divided in two phases: constraint generation and constraint resolution. The former extracts from the program text a set of constraints related to the relevant information, while the latter solves these constraints. The author introduces set constraints, a widely used type of constraints, and shows how some classical problems such as standard dataflow equations, simple type inference and monomorphic closure analysis can be represented as particular instances of set constraint problems. Nevertheless, he does not consider any particular programming language. Moreover, the main goal of [4] is to propose different ways of solving the constraints. This article follows a similar direction, since the process of designing novel static analyses is divided in two phases : construction of the abstract constraint graph and its solution. For Java and Java bytecode, a similar approach has been used in [39,29]. However, the abstract domains and the propagation rules used there are completely different from those introduced in this paper.

A static analysis that over-approximates the set of fields that might be `null` at some point has been introduced in [38]. However, more complex expressions than just fields are not considered there. Our analysis is also related to the well-known *available expression analysis* [3] where, however, only variables of primitive type are considered. Hence, it is much easier to deal with side-effects there. Fields can be sometimes transformed into local variables before a static analysis is performed [6], but this requires a preliminary modification of the code and we want to deal with more general expressions than just fields.

The analysis introduced in this paper can also be related to the well-known technique of global value numbering [8,34,20,35,16,17], which is a classic analysis for finding must-equalities in programs, heavily used by compilers for many optimizations. It determines equivalent computations inside a program and then eliminates repetitions. Checking equivalence of program expressions is an undecidable problem, and the tools dealing with this problem just try to under-approximate the actual sets of equivalent expressions by considering equivalent operands connected by equal operations. This form of equivalence, where the operators are treated as uninterpreted functions, is also called Herbrand equivalence [35, 25,26], and global value numbering helps discovering it. There exist two main approaches in global value numbering. The first one discovers all possible Herbrand equivalences [20, 17], while the second one discovers only those Herbrand equivalences created from program sub-expressions [8]. We infer, for each program point p, and each variable v available at p a set of expressions among all possible expressions available at p whose value be equal to v, for any possible program execution. This is similar to the first approach mentioned above. On the other hand, our technique is applied to a complex programming language, Java bytecode, and is an inter-procedural static analysis, while the papers mentioned above deal with an imperative while language and are intra-procedural. Bytecode instructions of our target language might have side-effects on the heap, which affect the must-equalities and must be taken into account for correctness. This is not the case in the articles mentioned above.

Our analysis can be applied to support code motion techniques that exploit the availability of must-equality information. In particular, partial redundancy elimination [24,21] moves instructions in order to perform optimizations, such as for instance strength reduction of loops. Those analyses have been defined for integer variables only, so the complexity of side-effects on the heap is not considered. Exceptional flows are not considered either. Our definite aliasing analysis can provide the must-equalities that might support code motion also in the case of expressions that reference objects in the heap rather than just integers. Moreover, it can support code motion in Java bytecode, also when exceptional paths must be taken into account for the correctness of the code motion.

1.3 Organization of the Paper

The present work is an extended version of an already published conference paper [28]. The main differences are:

- we provide syntax and formal semantics of our target Java bytecode-like language which includes also arrays;
- we formally define our constraint-based approach, explain how the constraints are constructed and solved and discuss existence and uniqueness of their solution;
- we provide a full proof of correctness of our approach, including all necessary technical lemmas;
- we illustrate our definitions and lemmas with more detailed examples;
- we discuss the implementation of our analysis, as well as its complexity;
- we provide larger experiments.

The rest of the paper is organized as follows. Section 2 introduces the syntax and the operational semantics of the Java bytecode-like language that we consider in this article. Section 3 defines the notion of alias expressions, their non-standard evaluation and specifies which bytecode instructions might modify the value of these expressions. Section 4 introduces our abstract interpretation-based static analysis and proves it correct. Section 5 shows the application of our analysis to many real-life examples, its precision and the way it affects the other analyses performed by our static analyzer Julia. Section 6 concludes.

2 Operational Semantics

This section presents a formal operational semantics for Java bytecode, inspired by its standard informal semantics in [22]. This is the same semantics used in [39]. A similar formalization, but in denotational form, has been used in [32,37,40]. Another approach using a similar representation of bytecode, in an operational setting, is [7], 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 transformers, 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 [39] whose form highly simplifies the definition of the abstract interpretation and its proof of soundness.

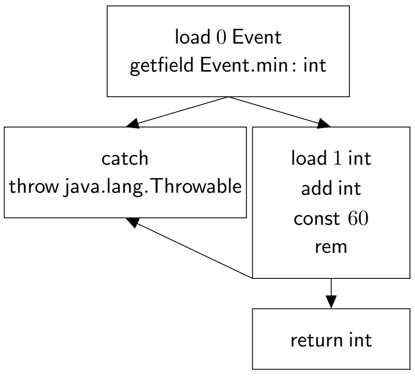
Java bytecode is the form of instructions executed by the Java Virtual Machine (JVM). Although it is a low-level language, it does support high-level concepts such as objects, dynamic dispatching, and garbage collection. Our formalization is at Java bytecode level for several reasons. First, it is much simpler than Java: there is a relatively small number of bytecode instructions, compared to varieties of source statements, and bytecode instructions

```
public class List{
 public Event head;
 public List tail;

 public List(){
   head = tail = null;
 }

 public List(Event head,
             List tail){
   this.head = head;
   this.tail = tail;
 }

 public Event getFirst(){
   return head;
 }

 public Event removeFirst(){
   Event result = head;
   if (tail != null) {
     head = tail.head;
     tail = tail.tail;
   } else {
     head = null;
   }
   return result;
 }
}
```

a)

```
public class Event{
 public int hr, min;
 ...
 public int delayMinBy(int offset){
   return (min + offset)%60;
 }

 public int setDelay(int offset){
     min = (min + offset)%60;
     return min;
 }
}
```

b)



c)

Fig. 2 Our running example: **a)**, **b)** - two simple classes `List` and `Event`; **c)** - our representation (CFG) of method `delayMinBy` of class `Event`

lack complexities like inner classes. Second, our implementation of reachability analysis is at bytecode level, bringing formalism, implementation and proofs closer. We require a formalization, since one of our goals is to prove our analysis sound.

2.1 Types

For simplicity, we assume that the only primitive type is int and that reference types are *classes*, containing *instance fields* and *instance methods*, and *arrays* of another type. Our implementation handles all Java primitive and reference types, e.g., classes with static fields and methods that, for simplicity, we do not consider in the present paper, as well as all the bytecode instructions except those dealing with multi-threading or reflection. For simplicity, our formalization considers only one primitive type (int), since all other primitive types may be handled in an analogous way. Interfaces are also missing from our formalization. We observe, however, that they are relevant at compilation time in Java while they have little to do with the 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 consequence of the formal complexity that they would introduce, since static fields are always in scope, at every program point. Static method calls would complicate the semantics, by duplicating the rules for method call, one for instance methods and one for static methods, and 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). We think that our concrete and abstract semantics would become too complex if static fields and methods were presented in formal terms in this article.

Definition 1 (Types) Let $\mathbb{K}$ be the set the set of *classes* of a program. Every class has at most one *direct superclass* and an arbitrary number of *direct subclasses*. Let $\mathbb{A}$ be the set of all the *array types* of the program. A *type* is an element of $\mathbb{T} = \{\mathsf{int}\} \cup \mathbb{K} \cup \mathbb{A}$. Given two types t and t' we say that t' is a *subtype* of t, and we denote it by $\mathsf{t}' \leq \mathsf{t}$, if one of the following conditions holds:

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

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 $\mathbb{F}(\kappa) = \{\kappa'.f : \mathsf{t}' \mid \kappa \leq \kappa'\}$ denote the fields defined in κ or in any of its superclasses. A class $\kappa \in \mathbb{K}$ has *instance methods* $\kappa.m(\vec{\mathsf{t}}) : \mathsf{t}$ (a method m, defined in κ, with parameters of type $\vec{\mathsf{t}}$, returning a value of type $\mathsf{t} \in \mathbb{T} \cup \{\mathsf{void}\}$), where κ, $\vec{\mathsf{t}}$, and t are often omitted. Constructors are methods with the special name init, which return void. Elements of an array type $\alpha = \mathsf{t}[\,]$ are of type t', such that $\mathsf{t}' \leq \mathsf{t}$.

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

$$\begin{tikzcd}[row sep=tiny] & b_1 \\ \boxed{\begin{array}{c}\mathsf{ins}@p\\ \mathit{rest}\end{array}} \arrow[ur] \arrow[dr] & \cdots \\ & b_m \end{tikzcd}$$

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 write just ins instead of ins $@p$.

Example 1 Consider the Java method `delayMinBy` and its corresponding graph of basic blocks of bytecode instructions given in Fig.2 **b)** and **c)**. The latter contains a branch since the getfield `min` bytecode might throw a `NullPointerException` which would be temporarily caught and then re-thrown to the caller of the method. Otherwise, the execution continues with a block that reads the other parameter (load 1), adds it to the value read from the field `min` and returns the result modulo 60. Each bytecode instruction except return and throw always has one or more immediate successors. On the other hand, return and throw are placed at the end of a method or constructor and have no successors. ■

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 run-time), 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 array of local variables and operand stack. Moreover, a standard algorithm [22] infers their static types. These static types are provided by the *type environment* map.

Definition 2 (Type environment) Each program point is enriched with a *type environment* τ i.e., a map from all the variables available at that point ($\mathsf{dom}(\tau)$) to their static types. We distinguish between *local variables* $L = \{l_0, \ldots\}$ and *stack elements* $S = \{s_0, \ldots\}$ i.e., $\mathsf{dom}(\tau) = L \cup S$.

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 4.

2.2 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 [22].

Definition 3 (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 paper) and where $\mathbb{L}$ is an infinite set of *memory locations*. The *default value* of a variable whose static type is int (respectively a reference type) is 0 (respectively $\texttt{null}$).

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

Definition 4 (Object representation) Given an object o, its type is maintained inside o in a special field $o.\kappa$ and we say that o is an *instance* of $o.\kappa$. Each object o contains its *internal environment* $o.\phi$ that maps every field $\kappa'.f{:}\mathsf{t}' \in \mathbb{F}(o.\kappa)$ into its value as provided in the object, denoted by $(o.\phi)(\kappa'.f : \mathsf{t}')$. Hence, the domain of $o.\phi$ is $\mathsf{dom}(o.\phi) = \mathbb{F}(o.\kappa)$ 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 paper is explained by the following definition.

Definition 5 (Array representation) Given an array a, its type is maintained inside a in a special field $a.\kappa$ and we say that a is an *instance* of $a.\kappa$. 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 $0 \le i < a.\texttt{length}$ 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 6 (State) A *state* σ over a type environment $\tau \in \mathcal{T}$ is a pair $\langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle, \mu\rangle$ 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.

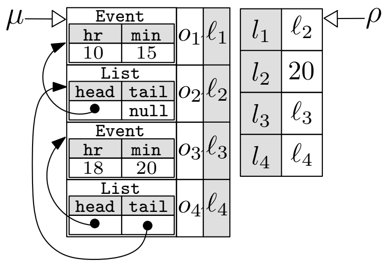


Fig. 3 A JVM state $\sigma = \langle\rho, \mu\rangle$

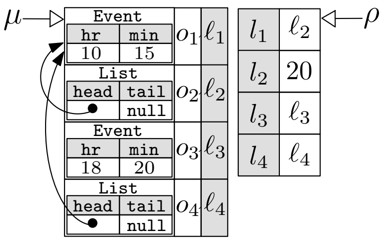


Fig. 4 A JVM state $\sigma_1 = \langle\rho, \mu_1\rangle$

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

Definition 7 (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 $\mathsf{t} = \mathsf{int}$ or
- $v = \mathtt{null}$ and $\mathsf{t} \in \mathbb{K} \cup \mathbb{A}$ or
- $v \in \mathbb{L}$, $\mathsf{t} \in \mathbb{K} \cup \mathbb{A}$ and $\mu(v).\kappa \le \mathsf{t}$.

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 $\kappa'.f\!:\!\mathsf{t}' \in \mathbb{F}(o.\kappa)$ 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 $a.\kappa = \mathsf{t}'[\,]$, the values in $\mathsf{rng}(a.\phi)$ are consistent with t'.

The Java Virtual Machine (JVM), as well as our formalization, supports exceptions. Therefore, we distinguish *normal* states Ξ arising during the normal execution of a piece of code, from *exceptional* states $\underline{\Xi}$ arising *just after* a bytecode that throws an exception. The operand stack of the states in $\underline{\Xi}$ has always 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 8 (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` i.e., $\tau'(s_0) \le$ `Throwable`.

Example 2 Consider a type environment $\tau = [l_1 \mapsto \mathtt{List};\ l_2 \mapsto \mathsf{int};\ l_3 \mapsto \mathtt{Event};\ l_4 \mapsto \mathtt{List}]$, where `List` and `Event` are classes defined in Fig. 2. In Fig. 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 $\ell_2 \in \mathbb{L}$, $20 \in \mathbb{Z}$, $\ell_3 \in \mathbb{L}$ 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 `Event`. 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. ■

2.3 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 [22]. We assume that we are analyzing a type-checked program, so that for instance field and method resolution always succeed. 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 [22]. Fig. 5 defines the semantics of bytecode instructions. We discuss it below.

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

Object Creation and Manipulation Instructions. These bytecode instructions create or access objects in memory: new κ creates a new instance of class κ whose fields hold the default values corresponding to their static types; getfield $\kappa.f$: t (respectively putfield $\kappa.f$: t) reads (respectively writes) a value from (respectively to) the field f belonging to the class κ and whose static type is t. Note that the creation of a new object initializes its fields to default values, that can be replaced programmatically, later, by the code of a constructor that gets called on the object just being created. That code will likely do extensive use of the putfield $\kappa.f$: t bytecode.

Array Creation and Manipulation Instructions. These bytecode instructions create or access arrays: arraynew t[ ] creates a new array of type t[ ] whose length is the value popped from the operand stack, initialize its elements to the default value for t 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 be a reference to an array of type 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 k-th element; *arraystore* α pops from the operand stack a value of type α, an integer k and a reference to an array of type α and writes the value into the k-th 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 the code if the condition is false.

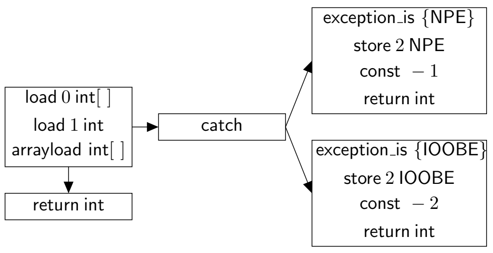
$$
\begin{array}{|rl|}
\hline
\mathit{const}\ x = & \lambda\langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle,\mu\rangle \,.\, \langle\langle \mathsf{l} \,\|\, x :: \mathsf{s}\rangle,\mu\rangle \\
\mathit{load}\ k\ \mathsf{t} = & \lambda\langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle,\mu\rangle \,.\, \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 \\
\hline
\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_2 * t_1 :: \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\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\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 \\
\hline
\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} \\
\hline
\mathit{arraynew}\ \alpha = & \lambda\langle\langle \mathsf{l} \,\|\, n :: \mathsf{s}\rangle,\mu\rangle \,.\, \begin{cases} \langle\langle \mathsf{l} \,\|\, \ell :: \mathsf{s}\rangle,\mu[\ell \mapsto a]\rangle & \text{if } n \geq 0 \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle,\mu[\ell \mapsto \mathit{nase}]\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{if } k \geq \mu(r).\mathtt{length} \text{ or } k < 0 \\ \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{if } k \geq \mu(r).\mathtt{length} \text{ or } k < 0 \\ \underline{\langle\langle \mathsf{l} \,\|\, \ell\rangle,\mu[\ell \mapsto \mathit{ase}]\rangle} & \text{if } v \in \mathbb{L} \text{ and } \mu(v).\kappa[\,] \not\leq \mu(r).\kappa \\ \langle\langle \mathsf{l} \,\|\, \mathsf{s}\rangle,\mu[(\mu(r).\phi)(k) \mapsto v]\rangle & \text{otherwise} \end{cases} \\
\hline
\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 \\
\hline
\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} \\
\hline
\mathit{return}\ \text{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 \text{void} \\
\hline
\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).\kappa \in K \\ \text{undefined} & \text{otherwise} \end{cases} \\
\hline
\end{array}
$$

Fig. 5 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 (with fields holding a default value) and a new array of type α (with elements holding a default value). Exceptions ae, $oome$, npe, $nase$, obe and ase are, respectively, new instances of the following: `ArithmeticException`, `OutOfMemoryError`, `NullPointerException`, `NegativeArraySizeException`, `ArrayIndexOutOfBoundsException` and `ArrayStoreException`

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. In actual Java bytecode, exception handlers are specified by a table that routes each exception type thrown inside a given program portion to a given exception handler. We have a different representation in this article, since we

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

a)



b)

Fig. 6 **a)** A simple Java method `get` handling two exceptions; **b)** Our representation of `get`, where NPE and IOOBE represent classes `NullPointerException` and `IndexOutOfBoundsException` respectively

only want instructions that behave as state transformers, also for exception handling. This simplifies the definition of the abstract interpretation. Hence, there is no exception table but the latter gets *compiled* into our graph of basic blocks. Namely, each program point where an exception might be raised gets linked to the exception handlers that might catch that exception. The catch bytecode starts an exception handler; it takes an exceptional state and transforms it into a normal one, subsequently used by the handler. After catch, bytecode exception_is K can be used to select an appropriate handler depending on the run-time class of the top of the stack, that is, of the exception that is raised: it filters those states whose top of the stack is an instance of a class in $K \subseteq \mathbb{K}$.

Example 3 In Fig. 6 a) we show a simple Java method `get` whose parameters are an array a of type int[ ] and an integer k, and which returns $a[k]$ if no exception occurs. This method handles two possible exceptions: `NullPointerException` (`NPE` for short) when a is `null` and `IndexOutOfBoundsException` (`IOOBE` for short) when k is greater or equal to a's length. These two exceptions are not subclasses one of another, i.e., neither `NPE` ≤ `IOOBE` nor `NPE` ≤ `IOOBE` holds. In Fig. 6 b) we show our representation of the `get` method. There are two separate blocks handling the two exceptions: one starting with exception_is NPE handling the case of `NPE` and the other starting with exception_is IOOBE handling the case of `IOOBE`. ■

Method calls and return. When a caller transfers control to a callee $\kappa.m(\vec{t}) : t$, the JVM runs an operation *makescope* $\kappa.m(\vec{t}) \colon t$ 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 i-th local variable of the callee is a copy of the $(\pi-1)-i$-th topmost element of the operand stack of the caller.

Definition 9 (makescope) Let $\kappa.m(\vec{t}) \colon t$ be a method and π the number of stack elements holding its actual parameters, including the implicit parameter this. We define a function

($makescope\ \kappa.m(\vec{\mathsf{t}})\colon \mathsf{t}$) : $\Sigma \to \Sigma$ as

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

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

Bytecode call $\kappa_1.m \ldots \kappa_n.m$ calls one of the callee 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 over-approximation 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 over-approximation can be computed by any *class analysis* [31]. The exact implementation of the method is later selected through a *makescope* instruction, that uses the method lookup rule of the programming language, as we will show soon with Fig. 7. If we assume that this lookup rule is deterministic, as in Java bytecode and Java, then only one of this targets will be selected at run-time, while for the others the *makescope* instruction at their beginning will be undefined. 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 Fig. 7 will show.

2.4 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 10 (Configuration) A *configuration* is a pair $\langle b \parallel \sigma\rangle$ of a block b and a state σ representing the fact that the JVM is about to execute b in state σ. An *activation stack* is a stack $c_1 :: c_2 :: \cdots :: c_n$ 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 11 (Operational Semantics) The (small step) operational semantics of a Java bytecode program P is a relation $a' \Rightarrow_P a''$ (P is usually omitted) providing the immediate successor activation stack a'' of an activation stack a'. It is defined by the rules in Fig. 7.

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

Rules (2) and (3) are for method calls. If a call occurs on a `null` receiver, no actual call happens in this case and rule (3) creates a new state whose operand stack contains only a reference to a `NullPointerException`. On the other hand, rule (2) calls a method on a non-`null` receiver: the call instructions are decorated with an over-approximation of the set of their possible run-time target methods. 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 look-up rules of the language, codified inside the *makescope* function. The latter is only defined when that implementation is selected at run-time; 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

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

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

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

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

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

$$\frac{1 \le w \le m}{\langle \boxed{}\!\begin{array}{l}\to b_1\\ \cdots\\ \to b_m\end{array} \parallel \sigma\rangle :: a \Rightarrow \langle b_w \parallel \sigma\rangle :: a} \tag{6}$$

Fig. 7 The transition rules of our semantics

and the call from the instructions to be executed at return time. Although this rule seems non-deterministic, only one thread of execution continues, since we assume that the look-up rules are deterministic, as in Java bytecode.

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

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

In the notation $\Rightarrow$, we often specify the rule in Fig. 7 used; for instance, we write $\overset{(1)}{\Rightarrow}$ for a derivation step through rule (1).

3 Alias Expressions

In this section, we define our expressions of interest (Definition 12), their *non-standard evaluation* (Definition 14), which might modify the content of some memory locations and we introduce the notion of *alias expression* (Definition 15). Moreover, we specify in which cases *a bytecode instruction might affect the value of an expression* (Definition 16), and when *the evaluation of an expression might modify a field* (Definition 17).

Definition 12 (Expressions) Given $\tau \in \mathcal{T}$, let $\mathcal{F}_\tau$ and $\mathcal{M}_\tau$ respectively denote the sets of the names of all possible fields and methods of all the objects available in Σ_τ. We define the set of expressions over $\mathsf{dom}(\tau)$:

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

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

Every expression has some important properties that can be determined statically. We formally define the notions of *sub-expression*, *depth*, *variables occurring in an expression* and *fields that an expression might read*.

Definition 13 For every type environment $\tau \in \mathcal{T}$, we define the following maps:

- $\mathsf{subExp} : \mathbb{E}_\tau \to \wp(\mathbb{E}_\tau)$ yielding the set of sub-expressions appearing in an expression;
- $\mathsf{depth} : \mathbb{E}_\tau \to \mathbb{N}$ mapping expressions to their depths;
- $\mathsf{variables} : \mathbb{E}_\tau \to \wp(\mathsf{dom}(\tau))$ yielding the set of variables occurring in an expression;
- $\mathsf{fields} : \mathbb{E}_\tau \to \wp(\mathcal{F}_\tau)$ yielding the fields that might be read through a method call executed during the evaluations of an expression.

These maps are defined inductively in Fig. 8.

Note that the definition of fields requires a preliminary computation of the fields possibly read by a method m, which might just be a transitive closure of the fields f for which a $\mathsf{getfield}$ occurs in m or in at least one method invoked by m. There exist some more precise approximations of this useful piece of information, e.g., that determined by our Julia tool. Anyway, in the absence of this approximation, we can always assume the least precise sound hypothesis: every method can read every field.

E	$\mathsf{depth}(\mathsf{E})$	$\mathsf{subExp}(\mathsf{E})$
n	0	$\{\mathsf{E}\}$
v		
$\mathsf{E}_1 \oplus \mathsf{E}_2$	$1 + \max_{i \in \{1,2\}}\{\mathsf{depth}(\mathsf{E}_i)\}$	$\{\mathsf{E}\} \cup \bigcup_{i=1}^{i=2} \mathsf{subExp}(\mathsf{E}_i)$
$\mathsf{E}_1[\mathsf{E}_2]$		
$\mathsf{E}.f$	$1 + \mathsf{depth}(\mathsf{E})$	$\{\mathsf{E}\} \cup \mathsf{subExp}(\mathsf{E})$
$\mathsf{E}.\mathtt{length}$		
$\mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_\pi)$	$1 + \max_{0 \le i \le \pi}\{\mathsf{depth}(\mathsf{E}_i)\}$	$\{\mathsf{E}\} \cup \bigcup_{i=0}^{\pi} \mathsf{subExp}(\mathsf{E}_i)$

E	$\mathsf{variables}(\mathsf{E})$	$\mathsf{fields}(\mathsf{E})$
n	$\varnothing$	$\varnothing$
v	$\{v\}$	
$\mathsf{E}_1 \oplus \mathsf{E}_2$	$\bigcup_{i=1}^{i=2} \mathsf{variables}(\mathsf{E}_i)$	$\bigcup_{i=1}^{i=2} \mathsf{fields}(\mathsf{E}_i)$
$\mathsf{E}_1[\mathsf{E}_2]$		
$\mathsf{E}.f$	$\mathsf{variables}(\mathsf{E})$	$\mathsf{fields}(\mathsf{E})$
$\mathsf{E}.\mathtt{length}$		
$\mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_\pi)$	$\bigcup_{i=0}^{\pi} \mathsf{variables}(\mathsf{E}_i)$	$\bigcup_{i=0}^{\pi} \mathsf{fields}(\mathsf{E}_i) \cup \{f \mid m \text{ might read } f\}$

Fig. 8 Auxiliary maps concerning expressions

Example 4 Consider the class `Event` introduced in Fig. 2 and let us assume that the static type of the local variable l_2 is `Event`. Then expression $\mathsf{E} = l_2$.`delayMinBy`(15) satisfies the following equalities:

- $\mathsf{depth}(l_2.\mathtt{delayMinBy}(15)) = 1 + \max\{\mathsf{depth}(l_2), \mathsf{depth}(15)\} = 1 + \max\{0, 0\} = 1$;
- $\mathsf{variables}(l_2.\mathtt{delayMinBy}(15)) = \mathsf{variables}(l_2) \cup \mathsf{variables}(15) = \{l_2\}$;
- $\mathsf{fields}(l_2.\mathtt{delayMinBy}(15)) = \mathsf{fields}(l_2) \cup \mathsf{fields}(15) \cup \{f \mid \mathtt{delayMinBy} \text{ might read } f\} = \{\mathtt{min}\}$.

The latter follows from the fact that `delayMinBy` contains only one getfield concerning the field `min` and no call invocations (Fig. 2). ■

In the following we show how the expressions introduced in Definition 12 are evaluated in an arbitrary state $\langle \rho, \mu \rangle$. It is worth noting that some of these expressions represent the result of a method invocation. Their evaluation, in general, might modify the initial memory μ, so we must be aware of the side-effects of the methods appearing in these expressions. We define the *non-standard evaluation* of an expression e in a state $\langle \rho, \mu \rangle$ as a pair $\langle w, \mu' \rangle$, where w is the computed value of e, while μ' is the updated memory obtained from μ after the evaluation of e.

Definition 14 (Non-standard evaluation of expressions) A *non-standard evaluation of expressions* in a state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ is a partial map $[\![\cdot]\!]^* : \mathbb{E}_\tau \to \Sigma_\tau \to \mathbb{V} \times \mathbb{M}$ defined as:

1. for every $n \in \mathbb{Z}$, $[\![n]\!]^*\sigma = \langle n, \mu \rangle$;
2. for every $v \in \mathsf{dom}(\tau)$, $[\![v]\!]^*\sigma = \langle \rho(v), \mu \rangle$;
3. $[\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\sigma$ is defined only if
   - $[\![\mathsf{E}_1]\!]^*\sigma = \langle n_1, \mu_1 \rangle$,
   - $[\![\mathsf{E}_2]\!]^*\langle \rho, \mu_1 \rangle = \langle n_2, \mu_2 \rangle$ and
   - $n_1, n_2 \in \mathbb{Z}$.

   In that case $[\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\sigma = \langle n_1 \oplus n_2, \mu_2 \rangle$, otherwise it is undefined;
4. $[\![\mathsf{E}.f]\!]^*\sigma$ is defined only if
   - $[\![\mathsf{E}]\!]^*\sigma = \langle \ell, \mu_1 \rangle$,
   - $\ell \in \mathbb{L}$,
   - $\mu_1(\ell).\kappa \in \mathbb{K}$ and
   - $f \in \mathbb{F}(\mu_1(\ell).\kappa)$.

   In that case $[\![\mathsf{E}.f]\!]^*\sigma = \langle (\mu_1(\ell).\phi)(f), \mu_1 \rangle$, otherwise it is undefined;
5. $[\![\mathsf{E}.\mathtt{length}]\!]^*\sigma$ is defined only if
   - $[\![\mathsf{E}]\!]^*\sigma = \langle \ell, \mu_1 \rangle$,
   - $\ell \in \mathbb{L}$ and
   - $\mu_1(\ell).\kappa \in \mathbb{A}$.

   In that case $[\![\mathsf{E}.\mathtt{length}]\!]^*\sigma = \langle \mu_1(\ell).\mathtt{length}, \mu_1 \rangle$, otherwise it is undefined;
6. $[\![\mathsf{E}_1[\mathsf{E}_2]]\!]^*\sigma$ is defined only if
   - $[\![\mathsf{E}_1]\!]^*\sigma = \langle \ell, \mu_1 \rangle$,
   - $[\![\mathsf{E}_2]\!]^*\langle \rho, \mu_1 \rangle = \langle n, \mu_2 \rangle$,
   - $\ell \in \mathbb{L}$,
   - $\mu_2(\ell).\kappa \in \mathbb{A}$,
   - $n \in \mathbb{Z}$ and
   - $0 \le n < \mu_2(\ell).\mathtt{length}$.

   In that case $[\![\mathsf{E}_1[\mathsf{E}_2]]\!]^*\sigma = \langle (\mu_2(\ell).\phi)(n), \mu_2 \rangle$, otherwise it is undefined;
7. in order to compute $[\![\mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_\pi)]\!]^*\sigma$, we determine $[\![\mathsf{E}_0]\!]^*\langle \rho, \mu \rangle = \langle w_0, \mu_0 \rangle$, and for each $1 \le i < \pi$, we evaluate E_{i+1} in the state $\langle \rho, \mu_i \rangle$: $[\![\mathsf{E}_{i+1}]\!]^*\langle \rho, \mu_i \rangle = \langle w_{i+1}, \mu_{i+1} \rangle$.

If $w_0 \in \mathbb{L}$ and $\mu_\pi(w_0).\kappa \in \mathbb{K}$, we run the method m on the object $\mu_\pi(w_0)$ with parameters $w_1, \ldots, w_\pi$ and if it terminates with no exception, the result of the evaluation is the pair composed of m's return value w and the memory μ' obtained from μ_π as a side-effect of m.

We write $[\![\mathsf{E}]\!]\sigma$ do denote the actual value of E in σ, without the updated memory.

The following example illustrates the non-standard evaluation of some simple expressions.

Example 5 Consider the state $\sigma = \langle\rho,\mu\rangle$ given in Fig. 3, where `Event` and `List` are the classes introduced in Fig. 2. Let us evaluate the following expressions in σ: l_3.`min`, l_4.`head.min`, l_3.`delayMinBy`(15) and l_4.`removeFirst`().`setDelay`(15).

$[\![l_3.\mathtt{min}]\!]^*\sigma$: By Definition 14 (case 2), we have $[\![l_3]\!]^*\langle\rho,\mu\rangle = \langle\rho(l_3),\mu\rangle = \langle\ell_3,\mu\rangle$, where $\ell_3 \in \mathbb{L}$. Moreover, $\mu(\ell_3).\kappa = \mathtt{Event} \in \mathbb{K}$ and $\mathtt{min} \in \mathbb{F}(\mathtt{Event})$. Hence, the conditions imposed by case 4 are satisfied and

$$[\![l_3.\mathtt{min}]\!]^*\langle\rho,\mu\rangle = \langle(\mu(\ell_3).\phi)(\mathtt{min}),\mu\rangle = \langle 20,\mu\rangle,$$

while $[\![l_3.\mathtt{min}]\!]\langle\rho,\mu\rangle = 20$.

$[\![l_4.\mathtt{head.min}]\!]^*\sigma$: By Definition 14 (case 2), we have $[\![l_4]\!]^*\langle\rho,\mu\rangle = \langle\rho(l_4),\mu\rangle = \langle\ell_4,\mu\rangle$, where $\ell_4 \in \mathbb{L}$. Moreover, $\mu(\ell_4).\kappa = \mathtt{List} \in \mathbb{K}$ and $\mathtt{head} \in \mathbb{F}(\mathtt{List})$. Hence, the conditions imposed by case 4 are satisfied and

$$[\![l_4.\mathtt{head}]\!]^*\langle\rho,\mu\rangle = \langle(\mu(\ell_4).\phi)(\mathtt{head}),\mu\rangle = \langle\ell_3,\mu\rangle,$$

while $[\![l_4.\mathtt{head}]\!]\langle\rho,\mu\rangle = \ell_3$. Similarly, $\mu(\ell_3).\kappa = \mathtt{Event} \in \mathbb{K}$ and $\mathtt{min} \in \mathbb{F}(\mathtt{Event})$, hence:

$$[\![l_4.\mathtt{head.min}]\!]^*\langle\rho,\mu\rangle = \langle(\mu(\ell_3).\phi)(\mathtt{min}),\mu\rangle = \langle 20,\mu\rangle.$$

while $[\![l_4.\mathtt{head.min}]\!]\langle\rho,\mu\rangle = 20$.

$[\![l_3.\mathtt{delayMinBy(15)}]\!]^*\sigma$: We have already shown that $[\![l_3]\!]^*\langle\rho,\mu\rangle = \langle\ell_3,\mu\rangle$, where $\ell_3 \in \mathbb{L}$ and $\mu(\ell_3).\kappa \in \mathbb{K}$. By Definition 14 (case 1), we have $[\![15]\!]^*\langle\rho,\mu\rangle = \langle 15,\mu\rangle$. Since the resulting memory did not change, we run the method `delayMinBy` on the object $\mu(\ell_3) = o_3$ with parameter 15 (case 5 of Definition 14). This method has no side effects (Fig. 2) and returns $((o_3.\phi)(\mathtt{min}) + 15)\%60 = 20 + 15 = 35$. Therefore,

$$[\![l_3.\mathtt{delayMinBy(15)}]\!]^*\langle\rho,\mu\rangle = \langle 35,\mu\rangle,$$

while $[\![l_3.\mathtt{delayMinBy(15)}]\!]\langle\rho,\mu\rangle = 35$.

$[\![l_4.\mathtt{removeFirst().setDelay(15)}]\!]^*\sigma$: We first determine $[\![l_4.\mathtt{removeFirst()}]\!]^*\sigma$. Similarly to the previous two cases, we have $[\![l_4]\!]^*\langle\rho,\mu\rangle = \langle\rho(l_4),\mu\rangle = \langle\ell_4,\mu\rangle$, where $\ell_4 \in \mathbb{L}$ and $\mu(\ell_4).\kappa = \mathtt{List} \in \mathbb{K}$. Then we run the method `removeFirst` on the object $\mu(\ell_4) = o_4$, which returns the value of its field `head`, i.e., $(o_4.\phi)(\mathtt{head}) = \ell_3 \in \mathbb{L}$, which is then removed from the list. Therefore, $[\![l_4.\mathtt{removeFirst()}]\!]^*\langle\rho,\mu\rangle = \langle\ell_3,\mu_1\rangle$, where μ_1 is the updated memory depicted in Fig. 4. Since $[\![l_3]\!]^*\langle\rho,\mu_1\rangle = \langle\rho(l_3),\mu_1\rangle = \langle\ell_3,\mu_1\rangle$, with $\ell_3 \in \mathbb{L}$, $\mu_1(\ell_3).\kappa = \mathtt{Event} \in \mathbb{K}$ and $[\![15]\!]^*\langle\rho,\mu_1\rangle = \langle 15,\mu\rangle$, we can run method `setDelay` on the object $\mu_1(\ell_3) = o_3$, which updates the value of the field `min` of the latter and returns that updated value, i.e., $((o_3.\phi)(\mathtt{min}) + 15)\%60 = 20 + 15 = 35$. Thus, we obtain

$$[\![l_4.\mathtt{removeFirst().setDelay(15)}]\!]^*\langle\rho,\mu\rangle = \langle 35,\mu_1[(\mu(\ell_3).\phi)(\mathtt{min}) \mapsto 35]\rangle,$$

while $[\![l_4.\mathtt{removeFirst().setDelay(15)}]\!]\langle\rho,\mu\rangle = 35$.

■

Finally, we define the notion of *alias expression*.

Definition 15 (Alias Expression) We say that an expression $\mathsf{E} \in \mathbb{E}_\tau$ is an *alias expression* of a variable $v \in \mathsf{dom}(\tau)$ in a state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ if and only if $[\![\mathsf{E}]\!]\sigma = \rho(v)$.

Example 6 Consider again the state $\langle \rho, \mu \rangle$ given in Fig. 3. The value of the local variable 2 in that state is $\rho(l_2) = 20$. Moreover, in Example 5 we showed that $[\![l_3.\mathtt{min}]\!]\langle \rho, \mu \rangle = 20$ and $[\![l_4.\mathtt{head.min}]\!]\langle \rho, \mu \rangle = 20$. Thus,

$$[\![l_3.\mathtt{min}]\!]\langle \rho, \mu \rangle = [\![l_4.\mathtt{head.min}]\!]\langle \rho, \mu \rangle = \rho(l_2) = 20$$

and, by Definition 15, we can state that both $l_3.\mathtt{min}$ and $l_4.\mathtt{head.min}$ are alias expressions of l_2 in $\langle \rho, \mu \rangle$. Similarly,

$$[\![l_4.\mathtt{head}]\!]\langle \rho, \mu \rangle = \rho(l_3) = \ell_3,$$

and we can state that $l_4.\mathtt{head}$ is an alias expression of l_3 in $\langle \rho, \mu \rangle$. ■

We specify when an execution of a bytecode instruction might affect the value of an expression. The following definition requires additional information about the fields that might be updated and about the static types of the arrays that might be updated by an execution of a method. These pieces of information can be computed statically, for example, by the side-effects analysis of Julia. It is worth noting that when this information is not available, our analysis is still sound, although less precise: we may assume that every field and every array of any array type might be updated.

Definition 16 (canBeAffected) Let τ and τ' be the static type information at and immediately after a bytecode instruction ins. Suppose that $\mathsf{dom}(\tau)$ contains i local variables and j stack elements. In Fig. 9, we define a map $\mathsf{canBeAffected}(\cdot, \mathsf{ins}) : \mathbb{E}_\tau \to \{\mathit{true}, \mathit{false}\}$ which, for every expression $\mathsf{E} \in \mathbb{E}_\tau$, determines whether E might be affected by an execution of ins.

That is, instructions that remove some variables from the stack (store, add, sub, mul, div, rem, putfield, arrayload, arraystore, ifeq, ifne, return and throw) affect the evaluation of all the expressions in which these variables appear. For instance, the execution of ifne t modifies the value of all the expressions containing the topmost stack element s_{j-1}. Instructions that write into a variable (store, add, sub, mul, div, rem, inc, getfield, arraylength and arrayload) might affect the evaluation of the expressions containing that variable. For instance, the execution of store k t might modify the value of all the expressions containing the local variable l_k, since this instruction writes a new value into that variable. Instruction putfield f might modify the evaluation of all the expressions that might read f. Instruction arraystore t[ ] might modify the evaluation of all the expressions that might read an array element whose type is compatible with t. Finally, call $m_1 \ldots m_k$ might modify the evaluation of all the expressions that might read a field f possibly modified by an m_w and of all the expressions which might read an element of an array of type t′[ ] if there exists a dynamic target m_w that writes into an array of type t[ ], where t′ and t are compatible types. For example, putfield `min` and call `setDelay` might modify the value of the expressions that we evaluated in Example 5: $l_3.\mathtt{min}$, $l_3.\mathtt{delayMinBy}(15)$ and $l_4.\mathtt{removeFirst}().\mathtt{setDelay}(15)$.

On the other hand, the evaluation of an expression in a state might update the memory component of that state by modifying the value of some fields. In the following we specify whether any evaluation of an expression might modify some fields of interest.

ins	canBeAffected(E, ins) $=$ *true* if and only if
const v	never
load k t	
store k t	variables(E) $\cap \{l_k, s_{j-1}\} \neq \varnothing$
add	variables(E) $\cap \{s_{j-1}, s_{j-2}\} \neq \varnothing$
sub	
mul	
div	
rem	
inc k x	$l_k \in$ variables(E)
new κ	never
getfield $\kappa.f$:t	$s_{j-1} \in$ variables(E)
putfield $\kappa.f$:t	variables(E) $\cap \{s_{j-1}, s_{j-2}\} \neq \varnothing \ \vee\ \kappa.f$:t $\in$ fields(E)
arraynew t[ ]	$s_{j-1} \in$ variables(E)
arraylength t[ ]	$s_{j-1} \in$ variables(E)
arrayload t[ ]	variables(E) $\cap \{s_{j-1}, s_{j-2}\} \neq \varnothing$
arraystore t[ ]	variables(E) $\cap \{s_{j-1}, s_{j-2}, s_{j-3}\} \neq \varnothing \vee$ [there exists an evaluation of E which might read an element of an array of type t′[ ], where t′ $\in$ compatible(t)]
dup t	never
ifeq t ifne t	$s_{j-1} \in$ variables(E)

return t
return void
	variables(E) $\cap\, S \neq \varnothing$

throw κ	
catch	never
exception_is K	
call $m_1, \ldots, m_n$	there exists an execution of a dynamic target m_w, where $1 \le w \le n$, 1. [which might modify a field from fields(E)] or 2. [which might write into an element of an array of type t[ ] and there exists an evaluation of E which might read an element of an array of type t′[ ], where t′ $\in$ compatible(t)]

Fig. 9 Definition of a map canBeAffected$(\cdot, \text{ins}) : \mathbb{E}_\tau \to \{\textit{true}, \textit{false}\}$

Definition 17 (mightMdf) Function mightMdf specifies whether a field belonging to a set of fields $F \subseteq \mathcal{F}_\tau$ might be modified during the evaluation of an expression E:

$$
\begin{aligned}
\text{mightModify}(n, F) &= \textit{false}\\
\text{mightModify}(v, F) &= \textit{false}\\
\text{mightModify}(\mathsf{E}_1 \oplus \mathsf{E}_2, F) &= \text{mightModify}(\mathsf{E}_1, F) \vee \text{mightModify}(\mathsf{E}_2, F)\\
\text{mightModify}(\mathsf{E}.f, F) &= \text{mightModify}(\mathsf{E}, F)\\
\text{mightModify}(\mathsf{E}.\mathtt{length}, F) &= \text{mightModify}(\mathsf{E}, F)\\
\text{mightModify}(\mathsf{E}_1[\mathsf{E}_2], F) &= \text{mightModify}(\mathsf{E}_1, F) \vee \text{mightModify}(\mathsf{E}_2, F)\\
\text{mightModify}(\mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_\pi), F) &= \bigvee_{i=0}^{\pi} \text{mightModify}(\mathsf{E}_i, F) \ \vee \ [\text{an execution of } m\\
&\qquad \text{might modify a field from } F],
\end{aligned}
$$

where $n \in \mathbb{Z}$, $v \in \text{dom}(\tau)$, $f \in \mathcal{F}_\tau$ and $m \in \mathcal{M}_\tau$.

Namely, evaluations of constants and variables do not modify any field. Evaluations of $\mathsf{E}_1 \oplus \mathsf{E}_2$ and $\mathsf{E}_1[\mathsf{E}_2]$ modify a field from F if there exists an evaluation of E_1 or E_2 modifying a field from F. Similarly, evaluations of $\mathsf{E}.f$ and $\mathsf{E}.\mathtt{length}$ modify a field from F if there exists an evaluation of E modifying a field from F. Evaluations of $\mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_\pi)$ might modify a field from F if there is an evaluation of any of E_is modifying a field from F or if the execution of m might modify a field from F.

Example 7 Consider one more time the class `Event` introduced in Fig. 2. Since the method `setDelay`() writes into the field `min` of class `Event`, we have

$$\mathsf{mightModify}(l_4.\mathtt{removeFirst}().\mathtt{setDelay}(15), \{\mathtt{Event.min}{:}\,\mathsf{int}\}) = \mathit{true}.$$

■

4 Definite Expression Aliasing Analysis

In this section we define our definite expression aliasing analysis and prove its soundness. Namely, we introduce a static analysis which determines for every variable at each program point a set of expressions that are definitely aliased to that variable, for every possible execution of the program of interest. Our analysis deals with the exceptional flows inside the program under analysis and the side-effects induced by calls to non-pure methods. Our analysis is based on the construction of a graph whose arcs are decorated with propagation rules for aliasing information. The use of a graph for static analysis is not novel at all. The important point, here, is the use of this traditional technique for the definition of propagation rules that deal with the side-effects of methods and of field updates, as well as the exceptional paths of execution.

The crucial notion for the operational semantics of the target language introduced in Section 2 is the notion of *state*, representing a system configuration. The set of all possible states that might be related to a given program point is called the *concrete domain*, and it is denoted by C. Let P be a program under analysis, composed of a set of `.class` files, and let L be the set of libraries that P uses. Suppose that P's classes as well as libraries in L are archived in a `.jar` file, representing the input of our analysis. The actions performed by our approach are listed below.

– This analysis is abstract interpretation-based [12,13]. More precisely, we extract from the concrete states (Definition 6) only those pieces of information representing the actual aliasing information contained in those states, i.e., we define the *abstract domain* ALIAS, whose elements, *abstract states*, are simpler than their concrete counterparts and easier to deal with (Definition 18). Then we show how our abstract and concrete states are related by defining a *concretization map* (Definition 19). The latter explains the actual meaning of the abstract states.
– From the `.jar` archive, an extended control flow graph (eCFG) is extracted. It contains a node for each bytecode instruction available in P and L, some special nodes which deal with the side-effects of non-pure methods, as well as with exceptional and non-exceptional method ends, and different types of arcs which connect those nodes. There are some simple arcs connecting one source node with one sink node, but there are also some special arcs, composed of two source nodes and one sink node: their main purpose is to handle the side-effects of the methods in both their exceptional and their non-exceptional executions; they represent one of the contributions of the present paper.

Section 4.2 explains different parts of these graphs. It is worth noting that this step does not depend on any particular property of interest: the same graph can be reused as underlying structure for the definition of other static analyses dealing with some other properties of interest.
- For each arc present in the eCFG, we define a propagation rule $\Pi : \textsc{Alias} \rightarrow \textsc{Alias}$ representing the behavior of the bytecode instruction corresponding to the source node of the arc with respect to the abstract domain A, and therefore with respect to our property of interest. This step is property-dependent, i.e., different properties of interest give rise to different propagation rules. These propagation rules are introduced by Definitions 21-30. Our approach annotates each eCFG's arc with the corresponding propagation rule, obtaining the *abstract constraints graph* (ACG).
- From the annotated graph a *system of constraints* is extracted, representing the actual definition of our constraint-based static analysis. Its solution is the approximation provided by that static analysis. The extraction of constraints from the ACG is explained in Section 4.3.

In Section 4.4 we show different properties that the propagation rules mentioned above have. These properties allow us to show one of the main contributions of the present paper, the *soundness* of our definite expression aliasing analysis.

4.1 Concrete and Abstract Domains

In Section 3 we explained when an expression E is aliased to a variable v in a state σ. This notion strictly depends on the state σ. We want to determine that property statically, and the most natural way for representing the fact that a variable must be aliased to some expressions is to assign to each variable available at a program point, a set of expressions that always have the same value as that variable itself. We followed this idea and formally defined the abstract domain $\textsc{Alias}$.

Definition 18 (Concrete and Abstract Domain) The concrete and abstract domains over $\tau \in \mathcal{T}$ are $\mathsf{C}_\tau = \langle \wp(\Sigma_\tau), \subseteq, \cup, \cap, \Sigma_\tau, \varnothing \rangle$ and $\textsc{Alias}_\tau = \langle \mathcal{A}_\tau, \sqsubseteq, \sqcup, \sqcap, \top_\tau, \bot_\tau \rangle$, where

- $\mathcal{A}_\tau = (\wp(\mathbb{E}_\tau))^{|\tau|}$;
- for every $A^1 = \langle \mathsf{A}^1_0, \ldots, \mathsf{A}^1_{|\tau|-1} \rangle \in \mathcal{A}_\tau$ and $A^2 = \langle \mathsf{A}^2_0, \ldots, \mathsf{A}^2_{|\tau|-1} \rangle \in \mathcal{A}_\tau$,
$$A^1 \sqsubseteq A^2 \Leftrightarrow \forall 0 \le i < |\tau|, \mathsf{A}^1_i \supseteq \mathsf{A}^2_i;$$
- for every $A^1 = \langle \mathsf{A}^1_0, \ldots, \mathsf{A}^1_{|\tau|-1} \rangle \in \mathcal{A}_\tau$ and $A^2 = \langle \mathsf{A}^2_0, \ldots, \mathsf{A}^2_{|\tau|-1} \rangle \in \mathcal{A}_\tau$,
$$A^1 \sqcup A^2 = \langle \mathsf{A}^1_0 \cap \mathsf{A}^2_0, \ldots, \mathsf{A}^1_{|\tau|-1} \cap \mathsf{A}^2_{|\tau|-1} \rangle;$$
- for every $A^1 = \langle \mathsf{A}^1_0, \ldots, \mathsf{A}^1_{|\tau|-1} \rangle \in \mathcal{A}_\tau$ and $A^2 = \langle \mathsf{A}^2_0, \ldots, \mathsf{A}^2_{|\tau|-1} \rangle \in \mathcal{A}_\tau$,
$$A^1 \sqcap A^2 = \langle \mathsf{A}^1_0 \cup \mathsf{A}^2_0, \ldots, \mathsf{A}^1_{|\tau|-1} \cup \mathsf{A}^2_{|\tau|-1} \rangle;$$
- $\top_\tau = \varnothing^{|\tau|}$;
- $\bot_\tau = (\mathbb{E}_\tau)^{|\tau|}$.

For a fixed number $d \in \mathbb{N}$, we write $\textsc{Alias}_\tau^d$ to denote a restriction of $\textsc{Alias}_\tau$, where the depth of the expressions is at most d, i.e.,

$$\langle \mathsf{A}_0, \dots, \mathsf{A}_{|\tau|-1} \rangle \in \textsc{Alias}_\tau^d \Leftrightarrow \forall 0 \le i < |\tau|. \forall \mathsf{E} \in \mathsf{A}_i.\mathsf{depth}(\mathsf{E}) \le d.$$

$\textsc{Alias}_\tau^d$'s top and bottom elements are denoted by $\top_\tau^d$ and $\bot_\tau^d$ respectively.

It is worth noting that, for every τ, $\textsc{Alias}_\tau$ is infinite, while for a fixed $d \in \mathbb{N}$, $\textsc{Alias}_\tau^d$ is finite. Concrete states σ corresponding to an abstract element $\langle \mathsf{A}_0, \dots, \mathsf{A}_{|\tau|-1} \rangle$ must satisfy the aliasing information represented by the latter, i.e., for each $0 \le r < |\tau|$, the value of all the expressions from the set A_r in σ must coincide with the value of v_r in σ (*definite* aliasing). Our concretization map formalizes this intuition.

Definition 19 (Concretization map) Consider a type environment $\tau \in \mathcal{T}$ and an abstract state $A = \langle \mathsf{A}_0, \dots, \mathsf{A}_{|\tau|-1} \rangle \in \textsc{Alias}_\tau$. We define $\gamma_\tau : \textsc{Alias}_\tau \to \mathsf{C}_\tau$, the concretization map of our abstract domain $\textsc{Alias}_\tau$ as follows:

$$\gamma(A) = \{\langle \rho, \mu \rangle \in \Sigma_\tau \mid \forall 0 \le r < |\tau|. \forall \mathsf{E} \in \mathsf{A}_r. [\![\mathsf{E}]\!] \langle \rho, \mu \rangle = \rho(v_r)\}.$$

Example 8 Consider a type environment $\tau \in \mathcal{T}$ with $\mathsf{dom}(\tau) = \{l_1, \dots, l_4\}$ and the state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ given in Fig. 3. We define the following abstract states from $\textsc{Alias}_\tau$:

$$\begin{array}{llllll}
A^1 = \langle & \overbrace{\varnothing}^{l_1}, & \overbrace{\{l_3.\mathtt{min}\}}^{l_2}, & \overbrace{\{l_4.\mathtt{head}\}}^{l_3}, & \overbrace{\varnothing}^{l_4} & \rangle \in \textsc{Alias}_\tau^1 \\
A^2 = \langle & \varnothing, & \{l_3.\mathtt{min}, l_4.\mathtt{head.min}\}, & \{l_4.\mathtt{head}\}, & \varnothing & \rangle \in \textsc{Alias}_\tau^2 \\
A^3 = \langle & \{l_3\}, & \{l_3.\mathtt{min}, l_4.\mathtt{head.min}\}, & \varnothing, & \varnothing & \rangle \in \textsc{Alias}_\tau^2.
\end{array}$$

By Definition 18, we can state that the aliasing information contained in A^2 is more precise than that in A^1, since $A^2 \sqsubseteq A^1$. On the other hand, we cannot compare A^1 and A^3, since the approximation of the aliasing information related to l_2 of A^1 is less precise than that of A^3, but the approximation of the aliasing information related to l_3 of A^1 is more precise than that of A^3.

State σ satisfies the aliasing information contained in both A^1 and A^2: in Example 6 we have shown that in σ variable l_2 is aliased to $l_3.\mathtt{min}$ and $l_4.\mathtt{head.min}$, while variable l_3 is aliased to $l_4.\mathtt{head}$. Hence, $\sigma \in \gamma_\tau(A^2) \subseteq \gamma_\tau(A^1)$. According to A^3, l_1 is aliased to l_3, which is not the case in σ: $(\rho(l_1) = \ell_2 \neq \ell_3 = \rho(l_3)$, which entails $\sigma \notin \gamma_\tau(A^3)$. ■

We recall some well-known notions from lattice theory. A sequence $\{\vec{\mathsf{A}_i}\}_{i \in \mathbb{N}}$ of elements in $\mathcal{A}_\tau$ is an *ascending chain* if $n \le m \Rightarrow \mathsf{A}_n \sqsubseteq \mathsf{A}_m$. That sequence *eventually stabilizes* iff $\exists n_0 \in \mathbb{N}. \forall n \in \mathbb{N}. n \ge n_0 \Rightarrow \mathsf{A}_n = \mathsf{A}_{n_0}$. The following lemma concerning $\textsc{Alias}_\tau^d$ guarantees the computability of the unique least solution of our static analysis by a finite Kleene iteration. For the detailed proof see Appendix B.1.1. It is worth noting that $\textsc{Alias}_\tau$ does not satisfy that property.

Lemma 1 *Given a type environment $\tau \in \mathcal{T}$ and an integer $d \in \mathbb{N}$, every ascending chain of elements in $\textsc{Alias}_\tau^d$ eventually stabilizes.*

In order to show that $\textsc{Alias}$ is actually an abstract domain in terms of abstract interpretation, we must show that the concretization map γ that we introduced in Definition 19 gives rise to a Galois connection. Once we know that we deal with a Galois connection, all results of abstract interpretation can be used and in particular the definition of soundness of the abstract operations. The detailed proof of Lemma 2 can be found in Appendix B.1.2.

Lemma 2 *Given a type environment $\tau \in \mathcal{T}$, and the function $\gamma_\tau : \textsc{Alias}_\tau \to \mathsf{C}_\tau$, there exists a function $\alpha_\tau : \mathsf{C}_\tau \to \textsc{Alias}_\tau$ such that $\langle \mathsf{C}_\tau, \alpha_\tau, \gamma_\tau, \textsc{Alias}_\tau \rangle$ is a Galois connection.*

4.2 The Abstract Constraint Graph

We introduce here the notion of *extended control flow graph* (eCFG), i.e., a control flow graph extracted from a `.jar` archive composed of all classes belonging to the program under analysis, as well as of all classes from the auxiliary libraries that the program uses. Similarly to the traditional control flow graph (Section 2), eCFG is composed of a set of nodes corresponding to different bytecode instructions belonging to the program, and a set of arcs that connect those nodes. Differently from the traditional control flow graph, eCFG contains some special nodes and special arcs that are not present in the former. This section formally introduces both traditional and special nodes and arcs. After the eCFG is constructed, its arcs are annotated by different propagation rules defined in this section, and the resulting graph is called *abstract constraints graph* (ACG).

Definition 20 (eCFG) Let P be the program under analysis enriched with all the methods from the libraries that it uses, already in the form of a CFG of basic blocks for each method or constructor (Section 2). The *extended control flow graph* (eCFG) for P is a directed graph $\langle V, E\rangle$ (nodes, arcs) where:

1. V contains a node $\boxed{\mathsf{ins}}$ for each bytecode instruction ins in P;
2. for each method or constructor m in P, V contains nodes $\boxed{\mathsf{exit}@m}$ and $\boxed{\mathsf{exception}@m}$, representing the normal and the exceptional ends of m;
3. each node contains an abstract element representing an approximation of the information related to the property of interest at that point;
4. E contains directed arcs with one $(1-1)$ or two $(2-1)$ sources and always one sink. Each arc has a *propagation rule* i.e., a function over the abstract domain, from the approximation(s) contained in its source(s) to that contained in its sink. We distinguish the following types of arcs:
   - **Sequential arcs:** if ins is a bytecode instruction in P, distinct from call, immediately followed by a bytecode instruction ins′, distinct from catch, then a $1-1$ sequential arc is built from $\boxed{\mathsf{ins}}$ to $\boxed{\mathsf{ins}'}$;
   - **Final arcs:** for each return t and throw κ instructions occurring in a method or a constructor m of P, there are $1-1$ final arcs from $\boxed{\mathsf{return\ t}}$ to $\boxed{\mathsf{exit}@m}$ and from $\boxed{\mathsf{throw}\ \kappa}$ to $\boxed{\mathsf{exception}@m}$, respectively;
   - **Exceptional arcs:** for each bytecode instruction ins throwing an exception, immediately followed by a catch, an $1-1$ exceptional arc is built from $\boxed{\mathsf{ins}}$ to $\boxed{\mathsf{catch}}$;
   - **Parameter passing arcs:** for each call $m_1 \ldots m_q$ occurring in P with π parameters (including the implicit parameter `this`) we build, for each $1 \le w \le q$, a $1-1$ parameter passing arc from $\boxed{\mathsf{call}\ m_1 \ldots m_q}$ to the node corresponding to the first bytecode instruction of the method m_w;
   - **Return value arcs:** for each call ins_C = call $m_1 \ldots m_q$ to a method with π parameters (including the implicit parameter `this`) returning a value of type $\mathsf{t} \neq \mathsf{void}$, and each subsequent bytecode instruction ins_N distinct from catch, we build, for each $1 \le w \le q$, a $2-1$ return value arc from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit}@m_w}$ (2 sources, in that order) to $\boxed{\mathsf{ins}_N}$;
   - **Side-effects arcs:** for each call ins_C = call $m_1 \ldots m_q$ to a method with π parameters (including the implicit parameter `this`), and each subsequent bytecode instruction ins_N, we build, for each $1 \le w \le q$, a $2-1$ side-effects arc from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit}@m_w}$ (2 sources, in that order) to $\boxed{\mathsf{ins}_N}$, if ins_N is not a catch and a $2-1$ side-effect arc from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exception}@m_w}$ (2 sources, in that order) to $\boxed{\mathsf{catch}}$.

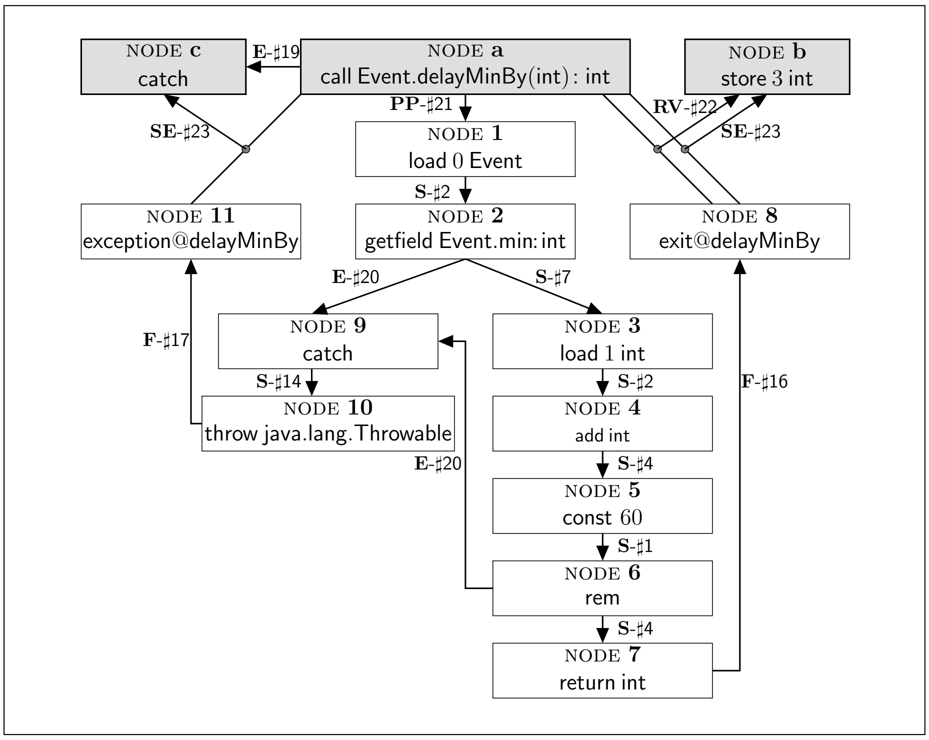


Fig. 10 The eCFG for the method `delayMinBy` in Fig. 2

The sequential arcs correspond to the non-exceptional executions of all the bytecode instructions except call, return and throw. The final arcs connect the nodes corresponding to the last bytecode instruction of each method or constructor m (i.e., return or throw) to the special nodes $\boxed{\text{exit@}m}$, in the case of return, and $\boxed{\text{exception@}m}$, in the case of throw. The exceptional arcs represent the exceptional executions of the bytecode instructions that might launch an exception, i.e., div, rem, new κ, getfield $\kappa.f$: t, putfield $\kappa.f$: t, arraynew α, arraylength α, arrayload α, arraystore α, throw and call, and they connect the nodes corresponding to these instructions with that related to the catch instruction at the beginning of their exceptional handlers. The parameter passing arcs link every node corresponding to a method call to that corresponding to the first bytecode instruction of the method(s) that might be called there. There exists a return value arc for each dynamic target m of a call ins_C returning a value. These arcs have two sources, $\boxed{\text{ins}_C}$ and $\boxed{\text{exit@}m_w}$, and they propagate the approximations present at these nodes to the node corresponding to the bytecode instruction following ins_C. Moreover, these arcs might enrich the resulting approximation with some additional abstract elements due to the returned value of m. The execution of method m might modify the memory where m is executed and this might affect the approximation at node $\boxed{\text{ins}_C}$ corresponding to the method call ins_C. The side-effects arcs deal with these phenomena, i.e., they are $2-1$ arcs connecting $\boxed{\text{ins}_C}$ and $\boxed{\text{exit@}m}$ (respectively $\boxed{\text{exception@}m}$) with the node corresponding to the bytecode instruction (respectively catch) which follows ins_C, for each dynamic target m of the call, and propagate the approximation at $\boxed{\text{ins}_C}$ modified by the side-effects of m's execution.

Example 9 In Fig. 10 we give the eCFG of the method `delayMinBy` introduced in Fig. 2. Nodes **a**, **b** and **c** belong to the caller of this method and exemplify the arcs related to the call and return bytecodes. Each arc is decorated with an abbreviation of the following form: **T**-$\sharp n$, where **T** denotes arc's type and may have the following values: **S**, **F**, **E**, **PP**, **RV** and **SE** representing sequential, final, exceptional, parameter passing, return value and side-effects arcs, respectively. On the other hand, $\sharp n$ represents the identification of the corresponding propagation rule (Definitions 21-30). ■

In the following we define different propagation rules which are used for the creation of the ACGs. These propagation rules represent the abstract semantics of our target language over the abstract domain Alias. We assume the presence of a *side-effects* approximation. Namely, we suppose that, for each method or constructor m available in the program under analysis, or in any of the libraries that the program may use, there exist the following pieces of information computed statically:

- a set of *fields that might be read* during any possible execution of m;
- a set of *fields that might be updated* during any possible execution of m;
- a set of *array types of all possible arrays whose elements might be read* during any possible execution of m and
- a set of *array types of all possible arrays whose elements might be updated* during any possible execution of m.

These pieces of information can be computed statically, and our tool Julia is able to provide them. Our analysis works correctly even when these approximations are not available: we can always assume that each method or constructor might read and modify every field and elements of arrays of every possible array type. In that case the definite expression aliasing information we determine would be less precise, but still sound.

According to Definition 20, we distinguish between simple ($1-1$) arcs, having one source and one sink node, and multi ($2-1$) arcs, which have two source and one sink node. We assume for all $1-1$ arcs that τ and τ' are the static type information at and immediately after the execution of a bytecode instruction ins, respectively. Moreover, we assume that τ contains j stack elements and i local variables. We write $\mathsf{noStackElements}(\mathsf{E})$ to denote that an expression E contains no stack elements, i.e., $\mathsf{variables}(\mathsf{E}) \cap \{s_0, \dots, s_{j-1}\} = \varnothing$. In the following we define the propagation rules related to the definite expression aliasing analysis.

Definition 21 (Sequential arcs) Each sequential arcs of the eCFG connecting nodes $\boxed{\mathsf{ins}}$ and $\boxed{\mathsf{ins}'}$ is annotated with a propagation rule $\lambda\langle \mathsf{A}_0, \dots, \mathsf{A}_{|\tau|-1}\rangle.\langle \mathsf{A}'_0, \dots, \mathsf{A}'_{|\tau'|-1}\rangle$, where, for each $0 \le r < |\tau'|$, A'_r is defined as follows:

<u>Rule #1</u>: If $\mathsf{ins} = \mathsf{const}\ v$, then

$$\mathsf{A}'_r = \begin{cases} \mathsf{A}_r & \text{if } r \neq |\tau| \\ \{v\} & \text{if } r = |\tau|. \end{cases}$$

<u>Rule #2</u>: If $\mathsf{ins} = \mathsf{load}\ k\ \mathsf{t}$, then

$$\mathsf{A}'_r = \begin{cases} \mathsf{A}_r \cup \mathsf{A}_r[s_j/l_k] & \text{if } r \notin \{k, |\tau|\} \\ \mathsf{A}_k \cup \{s_j\} & \text{if } r = k \\ \mathsf{A}_k \cup \{l_k\} & \text{if } r = |\tau|. \end{cases}$$

<u>RULE #3</u>: If $\mathsf{ins} = \mathsf{store}\ k\ \mathsf{t}$, then

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \text{if } r \neq k \\ \{\mathsf{E} \in \mathsf{A}_{|\tau|-1} \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \text{if } r = k. \end{cases}$$

<u>RULE #4</u>: If $\mathsf{ins} \in \{\mathsf{add}, \mathsf{sub}, \mathsf{mul}, \mathsf{div}, \mathsf{rem}\}$, then

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \text{if } r \neq |\tau|-2 \\ \{\mathsf{E}_1 \oplus \mathsf{E}_2 \mid \mathsf{E}_1 \in \mathsf{A}_{|\tau|-2} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_1, \mathsf{ins}) \wedge & \\ \qquad \mathsf{E}_2 \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_2, \mathsf{ins}) & \text{if } r = |\tau|-2, \end{cases}$$

where $\oplus$ is $+$, $-$, $\times$, div, % when ins is add, sub, mul, div, rem respectively.

<u>RULE #5</u>: If $\mathsf{ins} = \mathsf{inc}\ k\ x$, then

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E}[l_k - x/l_k] \mid \mathsf{E} \in \mathsf{A}_r\} & \text{if } r \neq k \\ \{\mathsf{E} + x \mid \mathsf{E} \in \mathsf{A}_r,\ l_k \notin \mathsf{variables}(\mathsf{E})\} & \text{if } r = k. \end{cases}$$

<u>RULE #6</u>: If $\mathsf{ins} = \mathsf{new}\ \kappa$, then

$$\mathsf{A}'_r = \begin{cases} \mathsf{A}_r & \text{if } r \neq |\tau| \\ \varnothing & \text{if } r = |\tau|. \end{cases}$$

<u>RULE #7</u>: If $\mathsf{ins} = \mathsf{getfield}\ \kappa.f{:}\mathsf{t}$, then

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \text{if } r \neq |\tau|-1 \\ \{\mathsf{E}.f \mid \mathsf{E} \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}) \wedge \neg\mathsf{mightModify}(\mathsf{E}, \{f\})\} & \text{if } r = |\tau|-1. \end{cases}$$

<u>RULE #8</u>: If $\mathsf{ins} = \mathsf{putfield}\ \kappa.f{:}\mathsf{t}$, then

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}.$$

<u>RULE #9</u>: If $\mathsf{ins} = \mathsf{arraynew}\ \mathsf{t}[\,]$, then

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \text{if } r \neq |\tau|-1 \\ \varnothing & \text{if } r = |\tau|-1. \end{cases}$$

<u>RULE #10</u>: If $\mathsf{ins} = \mathsf{arraylength}\ \mathsf{t}[\,]$, then

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \text{if } r \neq |\tau|-1 \\ \{\mathsf{E}.\mathtt{length} \mid \mathsf{E} \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \text{if } r = |\tau|-1. \end{cases}$$

<u>RULE #11</u>: If $\mathsf{ins} = \mathsf{arrayload}\ \mathsf{t}[\,]$, then

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \text{if } r \neq |\tau|-2 \\ \{\mathsf{E}_1[\mathsf{E}_2] \mid \mathsf{E}_1 \in \mathsf{A}_{|\tau|-2} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_1, \mathsf{ins}) \wedge & \\ \qquad \mathsf{E}_2 \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_2, \mathsf{ins}) \wedge & \\ \qquad [\mathsf{E}_1 \text{ and } \mathsf{E}_2 \text{ do not invoke any method}]\} & \text{if } r = |\tau|-2. \end{cases}$$

<u>RULE #12</u>: If $\mathsf{ins} = \mathsf{arraystore}\ \mathsf{t}[\,]$, then

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}.$$

Rule #13: If $\mathsf{ins} = \mathsf{dup}\ \mathsf{t}$, then

$$\mathsf{A}'_r = \begin{cases} \mathsf{A}_r \cup \mathsf{A}_r[s_j/s_{j-1}] & \text{if } r < |\tau|-1 \\ \mathsf{A}_{|\tau|-1} \cup \{s_j\} & \text{if } r = |\tau|-1 \\ \mathsf{A}_{|\tau|-1} \cup \{s_{j-1}\} & \text{if } r = |\tau|. \end{cases}$$

Rule #14: If $\mathsf{ins} \in \{\mathsf{ifeq}\ \mathsf{t}, \mathsf{ifne}\ \mathsf{t}, \mathsf{catch}, \mathsf{exception\_is}\ K\}$, then

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}.$$

Let us now explain, in more detail, the propagation rules introduced in Definition 21. The sequential arcs link an instruction ins to its immediate successor ins′ propagating, for every variable v at ins′, all those expressions E aliased to v at ins that cannot be affected by ins itself, i.e., such that $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ holds. However, some new alias expressions might be added to the initial approximation as well. We discuss the rules introduced above:

const v - in this case, a new variable $v_{|\tau|} = s_j$ is pushed onto the operand stack, and its value is v, while everything else stays unchanged. Therefore, for each variable available in $\mathsf{dom}(\tau)$, we just propagate its current approximation, while the approximation related to $v_{|\tau|}$ is $\{v\}$.

load k t - In this case a new variable (s_j) is pushed onto the operand stack and its value is equal to that of l_k. Therefore, for each variable $v_r \in \mathsf{dom}(\tau)$, we propagate all the alias expressions already present in A_r and by using the fact that $l_k = s_j$, we also add all those alias expression from A_r obtained by replacing all the occurrences of l_k with s_j. Obviously, s_j and l_k become alias expressions of s_j and l_k respectively.

store k t - In this case the topmost variable is popped from the operand stack (s_{j-1}) and its value is assigned to l_k. Therefore, all the alias expressions involving l_k and s_{j-1} in the initial approximations A_r, for any $r \neq k$, should be removed from the final ones (by Definition 16, $\mathsf{canBeAffected}(\mathsf{E}, \mathsf{store}\ k\ \mathsf{t}) = \mathit{true}$ if and only if l_k or s_{j-1} occurs in E). On the other hand, the final approximation related to l_k contains all the alias expressions $\mathsf{E} \in \mathsf{A}_{|\tau|-1}$ belonging to the initial approximation related to s_{j-1} which are not modified by the store k t, i.e, such that $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{store}\ k\ \mathsf{t})$ holds.

add, sub, mul, div, rem - in this case two topmost stack elements (memorized in s_{j-1} and s_{j-2}) of integer type are popped from the operand stack and the result of an opportune arithmetic operations applied to these two values is pushed back onto the operand stack (memorized in s_{j-2}). Therefore, for each variable v_r from $\mathsf{dom}(\tau')$, except s_{j-2}, we propagate all those alias expressions belonging to their initial approximations which might not be affected by this bytecode instruction, i.e., the ones not containing any occurrence of s_{j-1} and s_{j-2}. On the other hand, the expressions definitely aliased to the new topmost stack element s_{j-2} are of form $\mathsf{E}_1 \oplus \mathsf{E}_2$, where $\oplus$ is the arithmetic operation corresponding to this bytecode instruction, while E_1 and E_2 are expressions definitely aliased to s_{j-1} and s_{j-2} before the bytecode instruction is executed and which are not affected by this bytecode instruction.

inc k x - in this case the value memorized in the local variable l_k is incremented by x. Therefore, the final approximation related to each variable except l_k is composed of all the expressions available in the initial one where all the occurrences of l_k are replaced with $l_k - x$. In the case of l_k, we state that it becomes a definite alias of its old aliases (where l_k did not occur) plus x.

new κ - In this case a new object is created and the location it is bound to is pushed onto the operand stack, in s_j. Therefore, for each variable, except s_j, its initial approximation is kept. On the other hand, since s_j holds a fresh location, we soundly assume there is no expressions aliased to s_j.

getfield $\kappa.f$:t - In this case the location memorized in the topmost operand stack element s_{j-1} is replaced with the value of the field f of the object corresponding to the former. Therefore, for each variable, except s_{j-1}, its final approximation contains all the alias expressions from the initial one which are not modified by this bytecode instruction, i.e., the ones with no occurrence of s_{j-1} (Definition 16). On the other hand, the final approximation related to s_{j-1} contains the expressions $\mathsf{E}.f$ where E is aliased to s_{j-1} before the bytecode instruction is executed, it cannot be modified by the latter ($\neg$canBeAffected(E, ins), i.e., E contains no occurrences of s_{j-1}) and no evaluation of E might modify the field f ($\neg$mightModify(E, $\{f\}$)).

putfield $\kappa.f$:t - In this case the value memorized in the topmost operand stack element s_{j-1} is written in the field f of the object corresponding to the location memorized in the second topmost operand stack element s_{j-2}, and both s_{j-1} and s_{j-2} are popped from the operand stack. Hence, for each variable, we propagate all the alias expressions E belonging to its initial approximation which cannot be modified by this bytecode instruction, i.e., such that there is no occurrence of s_{j-2} and s_{j-1} in E and such that no evaluation of E might read the field f (Definition 16).

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

arraylength α - In this case the topmost operand stack element s_{j-1} containing a reference to an array is replaced with the length of that array. Therefore, for each variable, except s_{j-1}, its final approximation contains all the alias expressions from the initial one which are not modified by this bytecode instruction, i.e., the ones in which s_{j-1} does not appear (Definition 16). On the other hand, the expressions aliased to s_{j-1} are of the form `E.length`, where E is an alias expression of the old topmost stack element, which is not modified by this bytecode instruction.

arrayload α - In this case the k-th element of the array corresponding to the location memorized in the second topmost operand stack element s_{j-2}, where k is the topmost operand stack element, is written onto the top of the stack. Previously, both s_{j-1} and s_{j-2} are popped from the stack. The propagation rule and its explanation are analogous to the case of arithmetic operations.

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

dup t - In this case, the new topmost stack element s_j is a copy of the former topmost stack element s_{j-1}, hence they are trivially aliased to each other, and all the alias expressions of s_{j-1} at ins become the alias expressions of s_j at ins′. More precisely, we let $\mathsf{A}'_{|\tau|-1} = \mathsf{A}_{|\tau|-1} \cup \{s_j\}$ and $\mathsf{A}'_{|\tau|} = \mathsf{A}_{|\tau|-1} \cup \{s_{j-1}\}$. The approximations of the aliasing expressions of all other variables are enriched by all the expressions containing s_{j-1} already present in those approximations where occurrences of s_{j-1} are replaced by s_j: $\mathsf{A}'_r = \mathsf{A}_r \cup \mathsf{A}_r[s_j/s_{j-1}]$.

otherwise - catch and exception_is K do not modify the initial state, and therefore do not change the definite expression aliasing information available at that point, while ifne t and ifeq t just pop the topmost operand stack element, and therefore do not modify the aliasing expressions of any other variable which contains no occurrence of s_{j-1}.

Example 10 In Fig. 10 we introduced the ACG of the method `delayMinBy` from Fig. 2. Its arcs are decorated by the strings of the following form: **T**-$\sharp n$. The meaning of **T** has already been explained in Example 9, while $\sharp n$ are identifications of the propagation rules.

In the following examples for each node x we let τ_x, i_x and j_x denote the type environment, number of local variables and number of stack elements at x respectively. We let $A^x = \langle \mathsf{A}_0^x, \ldots, \mathsf{A}_{(i_x+j_x-1)}^x \rangle$ denote an approximation of the actual aliasing information at x, where each A_r^x denotes a set of expressions definitely aliased to the local variable l_r if $0 \le r < i_x$ or to a stack element s_{r-i_x} if $0 \le r - i_x < j_x$. Moreover, we assume that $i_\mathbf{a} = 3$, $j_\mathbf{a} = 2$ and that the call at node **a** occurs in a context with

$$\mathsf{A}_0^\mathbf{a} = \varnothing, \mathsf{A}_1^\mathbf{a} = \varnothing, \mathsf{A}_2^\mathbf{a} = \{v_1.\mathsf{getFirst}(), v_3\}, \mathsf{A}_3^\mathbf{a} = \{v_1.\mathsf{getFirst}(), v_2\} \text{ and } \mathsf{A}_4^\mathbf{a} = \{15\}. \quad (1)$$

■

The following example illustrates an application of some of the propagation rules introduced by Definition 21.

Example 11 Consider, for instance, nodes **2**, **3**, **4** and **5** in Fig. 10, and suppose that $i_\mathbf{2} = 2$ and $j_\mathbf{2} = 1$, i.e., at node **2** there are 2 local variables ($v_0 = l_0$ and $v_1 = l_1$) and 1 operand stack element ($v_2 = s_0$). Moreover, suppose that the variables v_0, v_1 and v_2 are respectively aliased to the following sets of expressions:

$$\mathsf{A}_0^\mathbf{2} = \{v_2\}, \quad \mathsf{A}_1^\mathbf{2} = \varnothing \quad \text{and } \mathsf{A}_2^\mathbf{2} = \{v_0\}.$$

Nodes **2** and **3** are linked by a sequential arc with propagation rule #7. It can be easily determined that $i_\mathbf{3} = 2$ and $j_\mathbf{3} = 1$. By Definition 16, at node **2**, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{getfield}\ \mathtt{min}) = v_2 \notin \mathsf{variables}(\mathsf{E})$. Moreover, according to Definition 17, $\mathsf{mightModify}(v_0, \{\mathtt{min}\}) = \mathit{false}$. Using these facts and Definition 21 (Rule #7) we obtain:

$$\begin{aligned}
\mathsf{A}_0^\mathbf{3} &= \{\mathsf{E} \in \mathsf{A}_0^\mathbf{2} \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{getfield}\ \mathtt{min})\} = \{\mathsf{E} \in \{v_2\} \mid v_2 \notin \mathsf{variables}(\mathsf{E})\} = \varnothing \\
\mathsf{A}_1^\mathbf{3} &= \{\mathsf{E} \in \mathsf{A}_1^\mathbf{2} \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{getfield}\ \mathtt{min})\} = \{\mathsf{E} \in \varnothing \mid v_2 \notin \mathsf{variables}(\mathsf{E})\} = \varnothing \\
\mathsf{A}_2^\mathbf{3} &= \{\mathsf{E}.\mathtt{min} \mid \mathsf{E} \in \mathsf{A}_2^\mathbf{2} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{getfield}\ \mathtt{min}) \wedge \neg\mathsf{mightModify}(\mathsf{E}, \{\mathtt{min}\})\} \\
&= \{\mathsf{E}.\mathtt{min} \mid \mathsf{E} \in \{v_0\} \wedge v_2 \notin \mathsf{variables}(\mathsf{E}) \wedge \neg\mathsf{mightModify}(\mathsf{E}, \{\mathtt{min}\})\} = \{v_0.\mathtt{min}\}.
\end{aligned}$$

Thus, $\Pi^{\#7}(A^\mathbf{2}) = \langle \varnothing, \varnothing, \{v_0.\mathtt{min}\} \rangle$.

Nodes **3** and **4** are linked by a sequential arc with propagation rule #2, and at node **4** we have $i_\mathbf{4} = j_\mathbf{4} = 2$. If we assume that $A^\mathbf{3} = \Pi^{\#7}(A^\mathbf{2})$, then, by Definition 21(Rule #2):

$$\begin{aligned}
\mathsf{A}_0^\mathbf{4} &= \mathsf{A}_0^\mathbf{3} \cup \mathsf{A}_0^\mathbf{3}[v_3/v_1] = \varnothing \\
\mathsf{A}_1^\mathbf{4} &= \mathsf{A}_1^\mathbf{3} \cup \{v_3\} = \varnothing \cup \{v_3\} = \{v_3\} \\
\mathsf{A}_2^\mathbf{4} &= \mathsf{A}_2^\mathbf{3} \cup \mathsf{A}_2^\mathbf{3}[v_3/v_1] = \{v_0.\mathtt{min}\} \cup \{v_0.\mathtt{min}\}[v_3/v_1] = \{v_0.\mathtt{min}\} \\
\mathsf{A}_3^\mathbf{4} &= \mathsf{A}_1^\mathbf{3} \cup \{v_1\} = \varnothing \cup \{v_1\} = \{v_1\}
\end{aligned}$$

Thus, $\Pi^{\#2}(A^\mathbf{3}) = \langle \varnothing, \{v_3\}, \{v_0.\mathtt{min}\}, \{v_1\} \rangle$.

Nodes **4** and **5** are linked by a sequential arc with propagation rule #4, and at node **5** we have $i_\mathbf{5} = 2$ and $j_\mathbf{5} = 1$. Note that at node **4**, by Definition 16, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{add}) = \mathsf{variables}(\mathsf{E}) \cap \{v_2, v_3\} = \varnothing$. If we assume that $A^\mathbf{4} = \Pi^{\#2}(A^\mathbf{3})$, then, by Definition 21 (Rule #4), we have:

$$\begin{aligned}
\mathsf{A}^5_0 &= \{\mathsf{E} \in \mathsf{A}^4_0 \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{add})\} = \{\mathsf{E} \in \varnothing \mid v_2, v_3 \notin \mathsf{variables}(\mathsf{E})\} = \varnothing \\
\mathsf{A}^5_1 &= \{\mathsf{E} \in \mathsf{A}^4_1 \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{add})\} = \{\mathsf{E} \in \{v_3\} \mid v_2, v_3 \notin \mathsf{variables}(\mathsf{E})\} = \varnothing \\
\mathsf{A}^5_2 &= \{\mathsf{E}_1 + \mathsf{E}_2 \mid \mathsf{E}_1 \in \mathsf{A}^4_2 \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_1, \mathsf{add}) \wedge \mathsf{E}_2 \in \mathsf{A}^4_3 \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_2, \mathsf{add})\} \\
&= \{\mathsf{E}_1 + \mathsf{E}_2 \mid \mathsf{E}_1 \in \{v_0.\mathtt{min}\} \wedge \mathsf{E}_2 \in \{v_1\} \wedge \mathsf{variables}(\mathsf{E}_1) \cap \{v_2, v_3\} = \mathsf{variables}(\mathsf{E}_2) \cap \{v_2, v_3\} = \varnothing\} \\
&= \{v_0.\mathtt{min} + v_1\}
\end{aligned}$$

Thus, $A^5 = \langle \varnothing, \varnothing, \{v_0.\mathtt{min} + v_1\}\rangle$. ∎

Definition 22 (Final arcs) Each final arc in the eCFG connecting nodes $\boxed{\mathsf{ins}}$ and $\boxed{\mathsf{ins}'}$ is annotated with a propagation rule $\lambda\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau|-1}\rangle.\langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{|\tau'|-1}\rangle$, where, for each $0 \le r < |\tau'|$, A'_r is defined as follows:

Rule #15: If $\mathsf{ins} = \mathsf{return}$ void, then

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\}.$$

Rule #16: If $\mathsf{ins} = \mathsf{return}$ t, then

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\} & \text{if } r \neq i \\ \{\mathsf{E} \in \mathsf{A}_{|\tau|-1} \mid \mathsf{noStackElements}(\mathsf{E})\} & \text{if } r = i. \end{cases}$$

Rule #17: If $\mathsf{ins} = \mathsf{throw}\ \kappa$, then

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\} & \text{if } r \neq i \\ \varnothing & \text{if } r = i, \end{cases}$$

where $\mathsf{noStackElements}(\mathsf{E})$ is true if and only if $\mathsf{variables}(\mathsf{E}) \cap \{s_0, \ldots, s_{j-1}\} = \varnothing$, i.e., if E contains no operand stack elements.

The **final arcs** introduced in Definition 22 feed nodes $\boxed{\mathsf{exit}@m}$ and $\boxed{\mathsf{exception}@m}$ for each method or constructor m. They propagate, for each local variable l_k available at $\boxed{\mathsf{exit}@m}$ (respectively $\boxed{\mathsf{exception}@m}$), all those expressions aliased to l_k at a $\boxed{\mathsf{return}}$ (respectively $\boxed{\mathsf{throw}}$) where no stack variable occurs. In the case of return t, with t ≠ void, the alias expressions of the only stack element at $\boxed{\mathsf{exit}@m}$ (i.e., $v_i = s_0$) are alias expressions of the topmost stack element at $\boxed{\mathsf{return\ t}}$ (s_{j-1}) with no stack elements. In the case of throw κ, we conservatively assume that no expression is aliased to the only stack element at $\boxed{\mathsf{exception}@m}$ ($v_i = s_0$).

Example 12 Consider nodes **7** and **8** in Fig. 10, which are linked by a final arc with propagation rule #16. It can be easily determined that at node **7**, $i_7 = 2$ and $j_7 = 1$, therefore, the only stack element there is $v_2 = s_0$, and $\mathsf{noStackElements}(\mathsf{E})$ holds if and only if $v_2 \notin \mathsf{variables}(\mathsf{E})$. Moreover, at node **8** we have $i_8 = 2$ and $j_8 = 1$. If we assume that $A^7 = \langle \varnothing, \varnothing, \{(v_0.\mathtt{min} + v_1)\%60\}\rangle$, then, by Definition 22 (Rule #16), we have:

$$\begin{aligned}
\mathsf{A}^8_0 &= \{\mathsf{E} \in \mathsf{A}^5_0 \mid \mathsf{noStackElements}(\mathsf{E})\} = \{\mathsf{E} \in \varnothing \mid \mathsf{noStackElements}(\mathsf{E})\} = \varnothing \\
\mathsf{A}^8_1 &= \{\mathsf{E} \in \mathsf{A}^5_1 \mid \mathsf{noStackElements}(\mathsf{E})\} = \{\mathsf{E} \in \varnothing \mid \mathsf{noStackElements}(\mathsf{E})\} = \varnothing \\
\mathsf{A}^8_2 &= \{\mathsf{E} \in \mathsf{A}^5_2 \mid \mathsf{noStackElements}(\mathsf{E})\} = \{\mathsf{E} \in \mathsf{A}^7_2 \mid v_2 \notin \mathsf{variables}(\mathsf{E})\} = \{(v_0.\mathtt{min} + v_1)\%60\}
\end{aligned}$$

Thus, $\Pi^{\#16}(A^7) = \langle \varnothing, \varnothing, \{(v_0.\mathtt{min} + v_1)\%60\}\rangle$. ∎

Definition 23 (Exceptional arcs) Each exceptional arc in the eCFG connecting nodes $\boxed{\mathsf{ins}}$ and $\boxed{\mathsf{ins}'}$ is annotated with a propagation rule $\lambda\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau|-1}\rangle.\langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{|\tau'|-1}\rangle$, where, for each $0 \leq r < |\tau'|$, A'_r is defined as follows:

<u>Rule #18</u>: If $\mathsf{ins} = \mathsf{throw}\ \kappa$, then:

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements(E)}\} & \text{if } r \neq i \\ \varnothing & \text{if } r = i, \end{cases}$$

where $\mathsf{noStackElements(E)}$ is true if and only if $\mathsf{variables(E)} \cap \{s_0, \ldots, s_{j-1}\} = \varnothing$, i.e., if E contains no operand stack elements.

<u>Rule #19</u>: If $\mathsf{ins} = \mathsf{call}\ m_1 \ldots m_n$, see Rule #18.

<u>Rule #20</u>: If ins is one of the following bytecode instructions: div, rem, $\mathsf{new}\kappa$, $\mathsf{getfield}\kappa.f\!:\!\mathsf{t}$, $\mathsf{putfield}\ \kappa.f\!:\!\mathsf{t}$, $\mathsf{arraynew}\ \alpha$, $\mathsf{arraylength}\ \alpha$, $\mathsf{arrayload}\ \alpha$ or $\mathsf{arraystore}\ \alpha$, see Rule #18.

The exceptional arcs link every instruction that might throw an exception to the catch at the beginning of their exception handler(s). They propagate alias expressions of local variables analogously to the final arcs. For the only stack element ($v_i = s_0$), holding the thrown exception, there is no alias expression ($\mathsf{A}_i = \varnothing$).

Example 13 Consider nodes **2** and **9** in Fig. 10, which are linked by an exceptional arc with propagation rule #20. Recall that, at node **2**, $i_5 = 2$ and $j_2 = 1$, therefore, the only stack element there is $v_2 = s_0$ and $\mathsf{noStackElements(E)}$ holds if and only if $v_2 \notin \mathsf{variables(E)}$. By Definition 23 and the hypotheses about A^2_0, A^2_1 and A^2_2 given in Example 11 we obtain:

$$\begin{aligned} \mathsf{A}^7_0 &= \{\mathsf{E} \in \mathsf{A}^2_0 \mid \mathsf{noStackElements(E)}\} = \{\mathsf{E} \in \varnothing \mid \mathsf{noStackElements(E)}\} = \varnothing \\ \mathsf{A}^7_1 &= \{\mathsf{E} \in \mathsf{A}^2_1 \mid \mathsf{noStackElements(E)}\} = \{\mathsf{E} \in \varnothing \mid \mathsf{noStackElements(E)}\} = \varnothing \\ \mathsf{A}^6_2 &= \varnothing \end{aligned}$$

Thus, $\Pi^{\#20}(A^2) = \langle \varnothing, \varnothing, \varnothing \rangle$. ■

Definition 24 (Parameter passing arcs) For each $\mathsf{call}\ m_1 \ldots m_q$ with π parameters (including the implicit parameter `this`), for each $1 \leq w \leq q$ we build an $1-1$ *parameter passing arc* from $\boxed{\mathsf{call}\ m_1 \ldots m_q}$ to the node corresponding to the first bytecode of m_w, with the propagation rule (Rule #21) $\lambda\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau|-1}\rangle.\langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{\pi-1}\rangle$, where, for each $0 \leq r < \pi$, $\mathsf{A}'_r = \varnothing$.

The above definition states that the approximation at the beginning of a method contains no aliasing information. This might be improved to introduce the possibility of letting abstract information flow from callers into callees. Our choice is sound but potentially imprecise. We have chosen that direction since, in practice, it is highly unlikely that all calls to a method will always provide the same parameter aliased to an alias expression, always the same for all possible calls. Moreover, for the analysis of libraries, the starting approximation of public methods can only be considered empty or the results would be unsound.

In the following we give some auxiliary definitions, that are necessary for the definition of the propagation rules for return value and side-effects arcs. We start with a map $\mathsf{noParameters}()$, that specifies whether there exists an actual argument of a method call among the variables appearing in an expression.

Definition 25 ($\mathsf{noParameters}$) Consider a type environment $\tau \in \mathcal{T}$ related to a program point with a method call $\mathsf{call}\ m_1 \ldots m_n$, and suppose that this method has π actual arguments (including the implicit parameter `this`). For every expression $\mathsf{E} \in \mathbb{E}_\tau$, we define a map $\mathsf{noParameters} : \mathbb{E}_\tau \to \{\mathit{true}, \mathit{false}\}$ as:

$$\mathsf{noParameters}(\mathsf{E}) = \mathsf{variables}(\mathsf{E}) \cap \{v_{|\tau|-\pi}, \ldots, v_{|\tau|-1}\} = \varnothing$$

(we recall that variables $v_{|\tau|-\pi}, \ldots, v_{|\tau|-1}$ correspond to π topmost operand stack elements).

The following definition specifies when the executions of a method are safe for an alias expression available at the point where the method is invoked.

Definition 26 ($\mathsf{safeExecution}$) Consider a type environment $\tau \in \mathcal{T}$ related to a program point with a method call $\mathsf{ins}_C = \mathsf{call}\ m_1 \ldots m_n$, and suppose that this method has π actual arguments (including the implicit parameter `this`). For every expression $\mathsf{E} \in \mathbb{E}_\tau$, we define a map $\mathsf{safeExecution}(\cdot, \mathsf{ins}_C) : \mathbb{E}_\tau \to \{\mathit{true}, \mathit{false}\}$ as:

$$\mathsf{safeExecution}(\mathsf{E}, \mathsf{ins}_C) = \mathsf{noParameters}(\mathsf{E}) \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}_C).$$

Namely, we say that *the executions of* ins_C *are safe for an expression* E, if all possible executions of all the dynamic targets m_i of ins_C cannot affect E ($\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}_C)$ holds) and if no actual parameter of ins_C appears in E (i.e., $\mathsf{noParameters}(\mathsf{E})$ holds). The former requires that every field that might be read by E must not be modified by any execution of any dynamic target m_i of ins_C, and that no execution of any dynamic target m_i of ins_C might write into an array whose elements might be read by E (Definition 16). The latter is required since the actual parameters of ins_C disappear from the operand stack after ins_C is executed.

The following definition will be useful soon to find alias expressions R of the return value of a method call that can be rephrased in the callee as alias expressions E that only use variables of the callee. For that, we require that the local variables $l_{k_0}, \ldots, l_{k_w}$ occurring in R are all formal parameters of the callee and that the corresponding actual parameters of the caller are aliased to some alias expressions $\mathsf{E}_{k_0}, \ldots, \mathsf{E}_{k_w}$. Moreover, E and the E_{k_i}, for $i = 0, \ldots, w$, must not be affected by the execution of the call.

Definition 27 ($\mathsf{safeAlias}$) Consider a type environment $\tau \in \mathcal{T}$ related to the program point of $\mathsf{ins}_C = \mathsf{call}\ m_1 \ldots m_n$, and suppose that the callees have π actual arguments (including the implicit parameter `this`). Let $\mathsf{E} \in \mathbb{E}_\tau$ have the form $\mathsf{E} = \mathsf{R}[\mathsf{E}_{k_0}/l_{k_0}, \ldots, \mathsf{E}_{k_w}/l_{k_w}]$ with $k_i < \pi$ for all i, where such l_{k_i} are all the local variables occurring in R. For every approximation $A = \langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau|-1} \rangle \in \textsc{Alias}_\tau$, we define a map $\mathsf{safeAlias}(\cdot, A, \mathsf{ins}_C) : \mathbb{E}_\tau \to \{\mathit{true}, \mathit{false}\}$ that is false whenever E has not the above form, and is otherwise defined as

$$\begin{aligned}\mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_C) = {} & \textstyle\bigwedge_{i=0}^{w}(\mathsf{E}_{k_i} \in \mathsf{A}_{|\tau|-\pi+k_i}) \wedge \bigwedge_{i=0}^{w} \mathsf{safeExecution}(\mathsf{E}_{k_i}, \mathsf{ins}_C) \wedge \\ & [\text{no evaluation of } \mathsf{E} \text{ might modify any field from } \mathsf{fields}(\mathsf{E}) \\ & \text{or any array element of type t if } \mathsf{E} \text{ also might read} \\ & \text{an array element of type t}' \text{ where t}' \le \text{t or t} \le \text{t}'].\end{aligned}$$

Finally, we specify when an alias expression of the returned value of a method available at the non-exceptional end of that method is safe at that point.

Definition 28 ($\mathsf{safeReturn}$) Consider a type environment $\tau \in \mathcal{T}$ related to a non-exceptional end of a method m, and suppose that this method has π formal arguments (including the implicit parameter `this`). For every expression $\mathsf{R} \in \mathbb{E}_\tau$, we define a map $\mathsf{safeReturn}(\cdot, m) : \mathbb{E}_\tau \to \{\mathit{true}, \mathit{false}\}$ as:

$$\mathsf{safeReturn}(\mathsf{R}, m) = \mathsf{variables}(\mathsf{R}) \subseteq \{l_0, \ldots, l_{\pi-1}\} \wedge \forall l_k \in \mathsf{variables}(\mathsf{R}), l_k \text{ is not modified by } m$$

(we recall that the formal arguments of a callee are held in the local variables $l_0, \ldots, l_{\pi-1}$).

We say that an *alias expression* R *of a return value at a non-exceptional end of a callee* m *is safe at that point* (i.e., safeReturn(R, m) holds) if only local variables holding the formal arguments of m ($l_0, \ldots, l_{\pi-1}$) appear in R and none of them might be modified by m. The latter condition requires that for each $l_k \in$ variables(R), no store k t nor inc k x occurs in m.

We can finally define the propagation rules for the return value and side-effects arcs of the ACGs for the definite expression aliasing analysis. These arcs have two sources, since they transfer local information that is not potentially affected by the call over the call. Namely, they use as source the approximation at the end of the callee, but also the approximation at the point of call, inside the caller.

Definition 29 (Return value arcs) For each ins_C = call $m_1 \ldots m_q$ to a method with π actual arguments (including the implicit parameter `this`) returning a value of type t ≠ void, and each subsequent bytecode instruction ins_N distinct from catch, we build, for each $1 \leq w \leq q$, a 2−1 *return value arc* from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit}@m_w}$ (2 sources, in that order) to $\boxed{\mathsf{ins}_N}$. Suppose that the static type information at $\boxed{\mathsf{ins}_C}$, $\boxed{\mathsf{exit}@m_w}$ and $\boxed{\mathsf{ins}_N}$ are τ_C, τ_E and τ_N, respectively. Given $A = \langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau_C|-\pi}, \ldots, \mathsf{A}_{|\tau_C|-1} \rangle \in \textsc{Alias}_{\tau_C}$ and $R = \langle \mathsf{R}_0, \ldots, \mathsf{R}_{|\tau_E|-1} \rangle \in \textsc{Alias}_{\tau_E}$, the propagation rule of these arcs is defined as $\lambda A, R.\langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{|\tau_C|-\pi} \rangle$, where for each $0 \leq r \leq |\tau_C| - \pi$, A'_r is defined by the Rule #22:

$$\mathsf{A}'_r = \begin{cases} \mathsf{A}_r & \text{if } r \neq |\tau_C|-\pi \\ \{\mathsf{E} = \mathsf{R}[\mathsf{E}_0, \ldots, \mathsf{E}_{\pi-1}/l_0, \ldots, l_{\pi-1}] | \mathsf{R} \in \mathsf{R}_{|\tau_E|-1} \wedge \mathsf{safeReturn}(\mathsf{R}, m_w) \wedge \mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_C)\} & \\ \cup \{\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1, \ldots, \mathsf{E}_{\pi-1}) \mid \mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_C)\} & \text{if } r = |\tau_C|-\pi. \end{cases}$$

Definition 30 (Side-effects arcs) For each ins_C = call $m_1 \ldots m_q$ to a method with π actual arguments (including the implicit parameter `this`), and each subsequent bytecode instruction ins_N, we build, for each $1 \leq w \leq q$, a 2−1 *side-effects arc* from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit}@m_w}$ (2 sources, in that order) to $\boxed{\mathsf{ins}_N}$, if ins_N is not a catch and a 2−1 *side-effects arc* from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exception}@m_w}$ (2 sources, in that order) to $\boxed{\mathsf{catch}}$. Suppose that the static type information at $\boxed{\mathsf{ins}_C}$, $\boxed{\mathsf{exit}@m_w}$ (or $\boxed{\mathsf{exception}@m_w}$) and $\boxed{\mathsf{ins}_N}$ are τ_C, τ_E and τ_N respectively. The propagation rule of these arcs is defined as:

$$\lambda \langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau_C|-\pi}, \ldots, \mathsf{A}_{|\tau_C|-1} \rangle, \langle \mathsf{R}_0, \ldots, \mathsf{R}_{|\tau_E|-1} \rangle.\langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{|\tau_N|-1} \rangle,$$

where for each $0 \leq r < |\tau_N|$, A'_r is defined by Rule #23:

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{safeExecution}(\mathsf{E}, \mathsf{ins}_C)\} & \text{if } r \neq |\tau_C|-\pi \\ \mathbb{E}_{\tau_N} & \text{if } r = |\tau_C|-\pi. \end{cases}$$

There exists a return value arc for each dynamic target m_w of a call ins_C returning a value. Rule #22 considers $\langle \mathsf{A}_0, \ldots, \mathsf{A}_{|\tau_C|-1} \rangle$ and $\langle \mathsf{R}_0, \ldots, \mathsf{R}_{|\tau_E|-1} \rangle$, approximations at $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit}@m_w}$, and builds the alias expressions related to the returned value $v_{|\tau_C|-\pi} = v_{|\tau_N|-1}$ at the node corresponding to the instruction which follows the call, $\boxed{\mathsf{ins}_N}$. An alias expression $\mathsf{R} \in \mathsf{R}_{|\tau_E|-1}$ of the computed value $v_{|\tau_E|-1}$ at $\boxed{\mathsf{exit}@m_w}$ can be turned into an alias expression of $v_{|\tau_C|-\pi}$ at $\boxed{\mathsf{ins}_N}$ if

1. R is safe at m_w and
2. every occurrence of a formal parameter l_k in R is replaced by an alias expression $\mathsf{E}_k \in \mathsf{A}_{|\tau_C|-\pi+k}$ of the corresponding actual parameter $v_{|\tau_C|-\pi+k}$ at $\boxed{\mathsf{ins}_C}$, which is safe w.r.t. ins_C.

Moreover, $\mathsf{E} = \mathsf{E}_0.m_w(\mathsf{E}_1, \ldots, \mathsf{E}_{\pi-1})$ can be an alias of $v_{|\tau_C|-\pi}$ at $\boxed{\mathsf{ins}_N}$ if it is safe w.r.t. ins_C.

The side-effects arcs consider the alias expressions E of the variables v_r different from the actual parameters $(v_{|\tau_C|-\pi}, \ldots, v_{|\tau_C|-1})$ of the method at $\boxed{\mathsf{ins}_C}$ and insert them among the alias expressions of v_r also at $\boxed{\mathsf{ins}_N}$ if they are safe w.r.t. ins_C.

Example 14 Nodes **a** and **8** are linked to node **b** through a return value and a side-effects arc with propagation rules #22 and #23. We recall that $i_{\mathbf{a}} = 3$, $j_{\mathbf{a}} = \pi = 2$ (Example 10), $i_{\mathbf{6}} = 2$, $j_{\mathbf{6}} = 1$ (Example 12) and the static type of the returned value of `delayMinBy` is not void. Therefore, $i_{\mathbf{b}}$ and $j_{\mathbf{b}}$ have to be 3 and 1 respectively. Let ins_C denote the call to `delayMinBy` at node **a** and let us first consider the return value arc and let us assume that $A^{\mathbf{a}} = \langle \mathsf{A}^{\mathbf{a}}_0, \mathsf{A}^{\mathbf{a}}_1, \mathsf{A}^{\mathbf{a}}_2, \mathsf{A}^{\mathbf{a}}_3, \mathsf{A}^{\mathbf{a}}_4 \rangle$ (Equation 1) and $A^{\mathbf{8}} = \Pi^{\#16}(A^{\mathbf{7}})$ (Example 12). Application of the propagation rule #22 (Definition 29), on the pair $(A^{\mathbf{a}}, A^{\mathbf{8}})$ gives: $\mathsf{A}^{\mathbf{b}}_r = \mathsf{A}^{\mathbf{a}}_r$ for $0 \le r \le 2$ and

$$\begin{aligned}\mathsf{A}^{\mathbf{b}}_3 = \{&\mathsf{E} = \mathsf{R}[\mathsf{E}_0, \mathsf{E}_1/l_0, l_1] | \mathsf{R} \in \mathsf{A}^{\mathbf{8}}_2 \wedge \mathsf{safeReturn}(\mathsf{R}, \texttt{delayMinBy}) \wedge \mathsf{safeAlias}(\mathsf{E}, A^{\mathbf{a}}, \mathsf{ins}_C)\} \\ &\cup \{\mathsf{E} = \mathsf{E}_0.\texttt{delayMinBy}(\mathsf{E}_1) \mid \mathsf{safeAlias}(\mathsf{E}, A^{\mathbf{a}}, \mathsf{ins}_C)\}.\end{aligned} \tag{2}$$

Consider the alias expression $\mathsf{R} = (v_0.\texttt{min} + v_1)\%60$, aliased to v_2 at node **8**, i.e.,

$$\mathsf{R} \in \mathsf{A}^{\mathbf{8}}_2. \tag{3}$$

It is clear that only formal parameters of `delayMinBy` (l_0 and l_1) appear in R. Moreover, it is possible to statically determine that `delayMinBy` does not modify l_0 and l_1 since that method contains no store nor inc instructions (Fig. 2). Thus,

$$\mathsf{safeReturn}(\mathsf{R}, \texttt{delayMinBy}) = \mathit{true}. \tag{4}$$

It is worth noting that the formal parameters l_0 and l_1 (node **1**) correspond to the actual parameters $v_3 = s_0$ and $v_4 = s_1$ (node **a**) of `delayMinBy`. By (2), an alias expression E of $v_3 = s_0$ at node **b** (holding the returned value of `delayMinBy`) can be obtained by substituting all occurrences of l_0 and l_1 in R by alias expressions E_0 and E_1 of s_0 and s_1 at node **a** respectively, only if E is safe w.r.t. ins_C, i.e., if $\mathsf{safeAlias}(\mathsf{E}, \mathsf{A}^{\mathbf{a}}, \mathsf{ins}_C)$ holds. By the hypotheses introduced in Example 10, the alias expressions of s_0 at **a** are v_2 and $v_1.\texttt{getFirst}()$, while the only alias expression of s_1 at **a** is 15. These expressions contain no actual parameter of the call at node **a**. Let us show that $\mathsf{E} = \mathsf{R}[v_1.\mathsf{getFirst}(), 15/l_0, l_1] = (v_1.\texttt{getFirst}().\texttt{min} + 15)\%60$ is safe w.r.t. ins_C. First of all we have:

$$v_1.\texttt{getFirst}() \in \mathsf{A}^{\mathbf{a}}_3 \ \text{ and } \ 15 \in \mathsf{A}^{\mathbf{a}}_4. \tag{5}$$

It is clear that no execution of `delayMin` might modify the evaluation of 15, since the latter is a constant. Thus,

$$\mathsf{safeExecution}(15, \mathsf{ins}_C) = \mathit{true}. \tag{6}$$

On the other hand, `delayMinBy` does not modify any field (Fig. 2), and therefore, it never modifies any field that might be read during any evaluation of $v_1.\mathsf{getFirst}()$, which implies that $\neg\mathsf{canBeAffected}(v_1.\mathsf{getFirst}(), \mathsf{ins}_C)$ holds, and therefore

$$\mathsf{safeExecution}(v_1.\mathsf{getFirst}(), \mathsf{ins}_C) = \mathit{true}. \tag{7}$$

Finally, `getFirst` reads no array element and, by Definition 17, modifies no fields:

$$\begin{aligned}&\mathsf{mightModify}(v_1.\texttt{getFirst}().\texttt{min} + 15, \mathsf{fields}(v_1.\texttt{getFirst}().\texttt{min} + 15)) \\ &= \mathsf{mightModify}(v_1.\texttt{getFirst}().\texttt{min}, F) \vee \mathsf{mightModify}(15, F) \\ &= \mathsf{mightModify}(v_1.\texttt{getFirst}(), F) \vee \mathit{false} \\ &= \mathit{false},\end{aligned} \tag{8}$$

where $F = \{\texttt{List.head:Object}, \texttt{Event.min:int}\}$. From Equations 6, 7 and 8 we obtain $\mathsf{safeAlias}(\mathsf{E}, \mathsf{A}^{\mathbf{a}}, \mathsf{ins}_C) = \mathit{true}$, which, together with (3) and (4) implies that E is an alias expression of v_3 at node **b**. We can similarly show that also the alias expression $\mathsf{R}[v_2, 15/l_0, l_1] = (v_2.\texttt{min} + 15)\%60$ is an alias expression of v_3 at **b**. It can be easily shown that also $v_1.\texttt{getFirst}().\texttt{delayMinBy}(15)$ and $v_2.\texttt{delayMinBy}(15)$ are safe w.r.t. ins_C. Namely,

- $v_1.\texttt{getFirst}() \in \mathsf{A}^{\mathbf{a}}_3$ and $15 \in \mathsf{A}^{\mathbf{a}}_4$;
- $\mathsf{safeAlias}(v_1.\texttt{getFirst}(), A^{\mathbf{a}}, \mathsf{ins}_C)$ holds (see (7));
- $\mathsf{safeAlias}(15, A^{\mathbf{a}}, \mathsf{ins}_C)$ holds (see (6);
- `getFirst` reads no array elements and by (3) modifies no fields that might be read by $v_1.\texttt{getFirst}()$ and 15.

Hence, $\mathsf{safeAlias}(v_1.\texttt{getFirst.delayMinBy}(15), A^{\mathbf{a}}, \mathsf{ins}_C)$ holds. It can be similarly shown that also $\mathsf{safeAlias}(v_2.\texttt{min} + 15, A^{\mathbf{a}}, \mathsf{ins}_C)$ holds. Therefore, Rule #22 gives rise to the following aliasing information:

$$\begin{aligned}
\mathsf{A}^{\mathbf{b}'}_0 &= \mathsf{A}^{\mathbf{b}'}_1 = \varnothing\\
\mathsf{A}^{\mathbf{b}'}_2 &= \{v_3, v_1.\mathsf{getFirst}()\}\\
\mathsf{A}^{\mathbf{b}'}_3 &= \{(v_1.\texttt{getFirst}().\texttt{min}+15)\%60, v_1.\texttt{getFirst}().\texttt{delayMinBy}(15),\\
&\quad (v_2.\texttt{min}+15)\%60, v_2.\texttt{delayMinBy}(15)\}.
\end{aligned}$$

Moreover, rule #23 states that an alias expression E of any local variable l_k at node **b** is an alias expressions of the same variable at node **a** if E is safe w.r.t. the executions of `delayMinBy` (i.e., if $\mathsf{safeExecution}(\mathsf{E}, \mathsf{ins}_C)$), while any expression can be an alias of v_3 at node **b**. It is obvious that ins_C is not safe for $v_3 \in \mathsf{A}^{\mathbf{a}}_2$ (since $\mathsf{noParameters}(v_3) = \mathit{false}$), while it is safe for $v_1.\mathsf{getFirst}()$, as we have already shown above (see (7)). Therefore, rule #23 gives rise to the following aliasing information:

$$\begin{aligned}
\mathsf{A}^{\mathbf{b}''}_0 &= \mathsf{A}^{\mathbf{b}''}_1 = \varnothing\\
\mathsf{A}^{\mathbf{b}''}_2 &= \{v_1.\mathsf{getFirst}()\}\\
\mathsf{A}^{\mathbf{b}''}_3 &= \mathbb{E}_{\tau_{\mathbf{b}}}
\end{aligned}$$

Hence, there are two arcs leading to node **b**, and bringing aliasing approximations $A^{\mathbf{b}'}$ and $A^{\mathbf{b}''}$. By Definition 18 we obtain the aliasing information at node **b** as:

$$\begin{aligned}
A^{\mathbf{b}} &= A^{\mathbf{b}'} \sqcup A^{\mathbf{b}''}\\
&= \langle \mathsf{A}^{\mathbf{b}'}_0 \cap \mathsf{A}^{\mathbf{b}''}_0, \mathsf{A}^{\mathbf{b}'}_1 \cap \mathsf{A}^{\mathbf{b}''}_1, \mathsf{A}^{\mathbf{b}'}_2 \cap \mathsf{A}^{\mathbf{b}''}_2, \mathsf{A}^{\mathbf{b}'}_3 \cap \mathsf{A}^{\mathbf{b}''}_3 \rangle\\
&= \langle \varnothing, \varnothing, \mathsf{A}^{\mathbf{b}''}_2, \mathsf{A}^{\mathbf{b}'}_3 \rangle.
\end{aligned}$$

■

4.3 Solution of the Constraints

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

Definition 31 (Constraints) Let N be a node of an ACG and A_N the approximation of the information contained in that node. Suppose that there are k arcs whose sink is N and for each $1 \leq i \leq k$, let Π^i and $\mathsf{approx}(i)$ respectively denote the propagation rule and the approximation of the property of interest at the source(s) of the i^{th} arc. These arcs give rise to the following constraints:

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

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

$$\bigsqcup_{i=1}^{k} \Pi^i(\mathsf{approx}(i)) \sqsubseteq A_N. \tag{9}$$

Definitions 18 and 31 entail that if k arcs reach the same node **n**, bringing to it k approximations, i.e., k sets of alias expressions for each variable at **n**, then the approximation of the actual aliasing information for each variable v at **n** should be included ($\subseteq$) in the intersection of the k sets related to v.

Example 15 Fig. 11 shows the constraints extracted from the ACG obtained by annotating the eCFG introduced in Example 9. These constraints concern the method `delayMinBy` only, and not the whole program under analysis. ■

$\Pi^{\#21}(\mathsf{A}^{\mathbf{a}}) \sqsubseteq \mathsf{A}^{\mathbf{1}}$	$\Pi^{\#1}\left(\mathsf{A}^{\mathbf{5}}\right) \sqsubseteq \mathsf{A}^{\mathbf{6}}$	$\Pi^{\#20}\left(\mathsf{A}^{\mathbf{2}}\right) \sqcup \Pi^{\#20}\left(\mathsf{A}^{\mathbf{6}}\right) \sqsubseteq \mathsf{A}^{\mathbf{9}}$
$\Pi^{\#2}\left(\mathsf{A}^{\mathbf{1}}\right) \sqsubseteq \mathsf{A}^{\mathbf{2}}$	$\Pi^{\#4}\left(\mathsf{A}^{\mathbf{6}}\right) \sqsubseteq \mathsf{A}^{\mathbf{7}}$	$\Pi^{\#22}\left(\mathsf{A}^{\mathbf{a}}, \mathsf{A}^{\mathbf{8}}\right) \sqcup \Pi^{\#23}\left(\mathsf{A}^{\mathbf{a}}, \mathsf{A}^{\mathbf{8}}\right) \sqsubseteq \mathsf{A}^{\mathbf{b}}$
$\Pi^{\#7}\left(\mathsf{A}^{\mathbf{2}}\right) \sqsubseteq \mathsf{A}^{\mathbf{3}}$	$\Pi^{\#16}\left(\mathsf{A}^{\mathbf{7}}\right) \sqsubseteq \mathsf{A}^{\mathbf{8}}$	$\Pi^{\#19}(\mathsf{A}^{\mathbf{a}}) \sqcup \Pi^{\#23}\left(\mathsf{A}^{\mathbf{a}}, \mathsf{A}^{\mathbf{11}}\right) \sqsubseteq \mathsf{A}^{\mathbf{c}}$
$\Pi^{\#2}\left(\mathsf{A}^{\mathbf{3}}\right) \sqsubseteq \mathsf{A}^{\mathbf{4}}$	$\Pi^{\#14}\left(\mathsf{A}^{\mathbf{9}}\right) \sqsubseteq \mathsf{A}^{\mathbf{10}}$	$\top^{\mathbf{a}} \sqsubseteq \mathsf{A}^{\mathbf{a}}$
$\Pi^{\#4}\left(\mathsf{A}^{\mathbf{4}}\right) \sqsubseteq \mathsf{A}^{\mathbf{5}}$	$\Pi^{\#17}\left(\mathsf{A}^{\mathbf{10}}\right) \sqsubseteq \mathsf{A}^{\mathbf{11}}$	

Fig. 11 The constraints extracted from the ACG given in Fig. 10

In the following we briefly discuss the existence and uniqueness of the solution of the system of constraints extracted from the ACG. This section is inspired by [27, Chapter 1.3].

Definition 32 Suppose that there are x nodes in the ACG under analysis, and for each $1 \leq n \leq x$, let τ_n and A_n be the static type information and the approximation concerned with the n^{th} node and let $d \in \mathbb{N}$ be a fixed expressions' depth. We let $\mathsf{EA}^d = (\textsc{Alias}^d_{\tau_1} \times \ldots \times \textsc{Alias}^d_{\tau_x})$ denote the set of tuples whose n^{th} element represents the approximation contained in the n^{th} node. Let $\vec{EA} = \langle A_1, \ldots, A_x \rangle$ and $\vec{EA}' = \langle A'_1, \ldots, A'_x \rangle$ be two arbitrary elements of EA^d. Consider the following ordering:

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

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

$$\vec{\bot}^d = \langle \bot^d_{\tau_1}, \ldots, \bot^d_{\tau_x} \rangle \qquad \vec{\top}^d = \langle \top^d_{\tau_1}, \ldots, \top^d_{\tau_x} \rangle.$$

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

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

Definition 33 Consider a function F operating over EA^d:

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

where $F_n\colon \mathsf{EA}^d \to \textsc{Alias}^n_{\tau_n}$ represents the constraint associated to the n^{th} node (Equation 9).

The following theorem shows how the least solution of the equation system $F(\vec{EA}) = \vec{EA}$ can be computed. Its proof is given in Appendix B.2.1.

Theorem 1 *The least solution of the equation system $F(\vec{EA}) = \vec{EA}$ exists and can be characterized as*

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

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

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

Moreover, the constraint system $F(\vec{EA}) \vec{\sqsubseteq} \vec{EA}$ and the equation system $\vec{EA} = F(\vec{EA})$ have the same least solution.

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

Definition 34 (Definite Expression Aliasing Analysis) The *solution* of an ACG is the least assignment of an abstract element $A_n \in \textsc{Alias}^d_{\tau_n}$ to each node n of the ACG, $\vec{EA} = \langle A_1, \dots, A_x\rangle \in \mathsf{EA}^d$, that satisfies the constraints extracted from the ACG and an initial constraint at the beginning of the program, i.e., such that $F(\vec{EA}) \vec{\sqsubseteq} \vec{EA}$ and $A_1 = \top$ hold, where we assume that A_1 is the node corresponding to the first statement of the `main` method of the program under analysis.

4.4 Soundness of our Approach

In this section we provide a theorem stating that our analysis sound. More precisely, suppose that we statically analyze a program using our Definite Expression Aliasing Analysis, and then we execute this program with an arbitrary input until one particular instruction. We show that the program state obtained at that point is included in the concretization of the abstract aliasing information concerning that point that we computed statically. This shows that our propagation rules correctly simulate the semantics of programs with respect to the definite aliasing information. The detailed proof of the theorem can be found in Appendix B.3.7.

Theorem 2 *Let $\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi\rangle \Rightarrow^* \left\langle \boxed{\begin{array}{c}\mathsf{ins}\\ \mathit{rest}\end{array}} \begin{array}{l}\to b_1\\ \cdots\\ \to b_m\end{array} \parallel \sigma\right\rangle :: a$ be an 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 $\xi \in \Sigma_{\tau_0}$, 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. Let $A \in \textsc{Alias}_\tau$ be the reachability approximation at ins, as computed by our reachability analysis starting from A_0. Then, $\sigma \in \gamma_\tau(A)$ holds.*

5 Implementation

We have implemented our definite expression aliasing analysis inside the Julia analyzer for Java bytecode (`http://www.juliasoft.com`) and we have analyzed some real-life benchmark programs. These benchmarks are reported in Fig. 12. Some are Android applications, that we analyze after being exported in Java bytecode format from the Eclipse IDE used for development. Hence we do not currently analyze their Dalvik bytecode. The benchmarks are analyzed together with most of the libraries that they use. In particular, in table 12 we report the libraries included in the analysis of each benchmark. Of course, the standard Java library (`java.*`) and the Android library (`android.*`, for Android benchmarks only) are always included and we do not report it in the figure. The fact that a library is included does not mean that *all* its code is analyzed: only the portion that is actually used by the benchmark is analyzed, and this is extracted through a traditional class analysis for object-oriented code [31]. The Android benchmarks are Mileage, OpenSudoku, Solitaire and Tilt-Mazes[1]; ChimeTimer, Dazzle, OnWatch and Tricorder[2]; TxWthr[3]; VoiceRecognition, Cube-LiveWallpaper, AccelerometerPlayActivity, SkeletonApp, AbdTest, Snake, BackupRestore, SoftKeyboard, MultiResolution, LunarLander, TestAppv2, TicTacToe, Spinner, TippyTipper, JetBoy, SampleSyncAdapter, NotePad, HoneycombGallery, Real3D, GestureBuilder, BluetoothChat, SearchableDictionary, ContactManager, Home and Wiktionary, that are all sample programs from the Android 3.1 distribution by Google[4]. The Java programs are JFlex, a lexical analyzers generator[5]; Plume, a library by Michael D. Ernst[6]; Nti, a non-termination analyzer by Étienne Payet[7]; Lisimplex, a numerical simplex implementation by Ricardo Gobbo[8]; avrora, an AVR simulation and analysis framework[9]; luindex, an indexer of documents, h2, a database benchmark, and sunflow, a ray tracer, from the DaCapo benchmark suite[10]; hadoop-common, a software for distributed computing[11]; and our julia analyzer itself[12].

Experiments have been performed on a Linux quad-core Intel Xeon machine running at 3.10GHz, with 12 gigabytes of RAM.

5.1 Results w.r.t. the expression aliasing analysis

By only considering the analysis introduced in this article, Fig. 12 shows that it is quite fast and scales to large software with a cost in time that has never exploded in our experiments. All analyses could be completed with less than one gigabyte of RAM. In the same figure, the last column on the right reports the average size of the set of expression aliases for each variable. We do not consider the tautological and trivial alias of a variable with itself. That

[1] `http://f-droid.org/repository/browse/`
[2] `http://moonblink.googlecode.com/svn/trunk/`
[3] `http://typoweather.googlecode.com/svn/trunk/`
[4] `http://developer.android.com/tools/samples/index.html`
[5] `http://jflex.de`
[6] `http://code.google.com/p/plume-lib`
[7] `http://personnel.univ-reunion.fr/epayet/Research/NTI/NTI.html`
[8] `http://sourceforge.net/projects/lisimplex`
[9] `http://compilers.cs.ucla.edu/avrora/`
[10] `http://www.dacapobench.org`
[11] `http://hadoop.apache.org`
[12] `http://www.juliasoft.com`

ID	NAME	LIBRARIES	LINES	METHODS	RUNTIME	SIZE
1	nti		13915	1653	0.51	0.17
2	lisimplex		25564	2729	1.28	0.15
3	avrora		38165	5006	3.29	0.12
4	JFlex		41365	4286	1.96	0.28
5	plume		44028	4646	2.63	0.12
6	VoiceRecognition		44974	5094	2.40	0.03
7	CubeLiveWallpaper		45891	5197	2.22	0.27
8	AccelerometerPlayActivity		47913	5394	2.34	0.19
9	SkeletonApp		57399	6371	3.49	0.32
10	AbdTest		58020	6402	6.14	0.14
11	Snake		58606	6473	3.11	0.26
12	BackupRestore		58706	6471	3.23	0.24
13	SoftKeyboard		58819	6535	4.31	0.35
14	MultiResolution		58917	6542	3.03	0.86
15	LunarLander		59122	6519	3.15	0.2
16	TestAppv2		59889	6587	3.14	0.35
17	TicTacToe		59943	6657	3.13	0.37
18	Spinner		61912	6759	3.32	0.42
19	luindex	lucene-core, lucene-demos	62050	6409	3.18	0.18
20	Solitaire		63507	6988	3.46	0.32
21	TippyTipper		65310	7322	3.36	0.89
22	JetBoy		65874	7189	3.73	0.22
23	SampleSyncAdapter		66646	7348	3.47	0.08
24	NotePad		67066	7372	3.56	0.28
25	sunflow	janino	72061	9130	6.18	0.18
26	HoneycombGallery		72352	7879	5.31	0.32
27	Real3D		75001	8179	4.37	0.15
28	TxWthr		75434	8232	43.45	0.35
29	Dazzle	hermitandroid	78344	8681	40.34	0.28
30	GestureBuilder		85213	9093	5.06	0.51
31	BluetoothChat		85290	9119	5.07	0.45
32	SearchableDictionary		88034	9392	5.24	0.24
33	ContactManager		88110	9465	5.35	0.48
34	Home		88256	9489	5.22	0.3
35	TiltMazes		90419	9641	5.35	0.39
36	ChimeTimer	hermitandroid	90465	9743	6.09	0.26
37	Mileage		104647	11188	6.07	0.16
38	Tricorder	hermitandroid	105475	11140	6.57	0.34
39	julia	bcel	106117	12495	15.42	0.29
40	Wiktionary		109140	11762	7.54	0.27
41	OnWatch	hermitandroid	114391	11928	7.01	0.26
42	hadoop-common	org.w3c.*, javax.security.* guava, protobuf-java, jets3t	118706	14812	5.91	0.21
43	OpenSudoku		120164	13002	6.59	0.22
44	h2	junit3, derbyTesting, tpcc	183398	17276	14.99	0.17

Fig. 12 The benchmarks that we have analyzed and the cost and precision of their expression aliasing analysis. For each benchmark we report the name, the number of non-comment non-blank source code lines that get analyzed, the number of methods that get analyzed, the time (in seconds) required for our expression aliasing analysis and the average size of the approximation of each variable (number of alias expressions per variables). The latter does not include the tautological aliasing of a variable with itself, which is trivial and irrelevant.

column shows that that size is small, which possibly accounts for the lack of computational explosion during the analysis. This is important, since a formal study of the worst-case complexity of our analysis leads to an exponential cost, in theory, as shown below.

5.2 Theoretical Computational Complexity

A worst-case complexity for our definite aliasing analysis can be estimated as follows. The number N of alias expressions (Definition 12) over a maximal number v of variables in scope, a number n of constants (those used in the program text), a number f of field names and a number m of method names is finite as soon as we fix a maximal depth k for the alias expressions themselves (Definition 18). More precisely, N is polynomial in v, n, f and m for a fixed k and exponential in k: $N \in O(v \cdot n \cdot f \cdot m \cdot 2^k)$. Note that we consider here the maximal number v of variables in scope at each given program point, which is usually small, and not the total number of program variables, that can be very large instead. The approximation of each program variable is a set of alias expressions; hence that set can only decrease N times, for each variable, during the analysis. There are at most v such sets at each program point, one for each variable in scope there. Hence, the chain of possible approximations at a given program point is long at most $v \cdot N$. If l is the length of the program, that is, the number of its program points, the number of iterations is consequently bounded from above by $l \cdot v \cdot N$. That is, our expression aliasing analysis requires $O(l \cdot v \cdot N)$ iterations until stabilization and its computational cost is, hence, polynomial in v, n, f and m for a fixed k and exponential in k: $O(v \cdot n \cdot f \cdot m \cdot 2^k)$. In our implementation, we have fixed $k = 4$.

It is interesting to observe that, in practice, very few iterations are needed to converge to a fixpoint. This is because many alias expressions cannot be generated because they would not type-check. Moreover, on average, each variable is approximated by very small alias sets (Fig. 12): in most cases, those sets are empty or singletons. For this reason, the theoretical computational result that we have just shown is a very pessimistic bound of the actual cost of the analysis.

5.3 Implementation Optimizations

An abstract Java class `Alias` is used to represent the alias expressions of Definition 12. Constants and variables are concrete and non-recursive subclasses of `Alias`. Other alias expressions are concrete and recursive subclasses of `Alias`; for instance, the alias expression `E`.f is represented by a subclass `FieldOf` of `Alias` that refers to the field f and to another alias expression, that is `E`. The reduction of the memory footprint of this representation for alias expressions is possible and important, since identical or at least similar alias expressions are often used at different program points. Namely, our representation of alias expressions is *interned*, that is, we never generate two Java objects that stand for the same alias expression. This allows a maximal sharing of data structures and reduces the memory cost of our representation (*compositum pattern*). We achieve this internment through a map that binds each alias expression e to the unique representative of all alias expressions equal to e. Since Julia is a parallel analyzer, race conditions must be avoided in the access to that map. To that purpose, we use a `java.util.concurrent.ConcurrentHashMap` and its handy `putIfAbsent()` method for checking the presence and putting new alias expressions in the table, atomically. Beyond the reduction of the memory cost, internment has the advantage that equality of alias expressions can be tested by faster == tests rather than `equals()` calls.

Sets of alias expressions are hence sets of unique objects. There are many such sets during the analysis, for each variable in scope at each given program point. Also in this case, a compact representation is needed. We have achieved this by using bitsets, represented through arrays of `longs` (each `long` contains 64 bits). A fixed enumeration of the alias expressions relates a given bit position to the expression that it represents. The enumeration is built on demand as soon as new alias expressions are created by the analysis. For each bit set to 1, the alias expression in that enumeration position is assumed to belong to the bitset.

5.4 Benefits for other analyses

Section 5.1 has shown that the aliasing expression analysis can be computed in a few seconds also for large applications. It remains to show that its results are useful for other, subsequent analyses, that exploit the availability of definite aliasing information. To that purpose, we have used our analysis to support Julia's nullness and termination analyses. In particular, we use our analysis at the then branch of each comparison `if (v!=null)` to infer that the definite aliases of `v` are non-`null` there, and at each assignment `w.f=exp` to infer that expressions `E.f` are non-`null` when `exp` is non-`null` and when `E` is a definite alias of `w` whose evaluation does not read nor write `f`. Moreover, we use it to infer symbolic upper or lower bounds of variables whenever we have a comparison such as `x<y`: all definite alias expressions of `y` (respectively `x`) are upper (respectively lower) bounds for `x` (respectively `y`). This is important for the termination analysis of Julia.

Note that our nullness and termination analyses are sound, that is, there are no false negatives (as long as reflection is not used or only used in a limited way: for instance, inflation of XML views in Android is supported in Julia [33]); but there are false positives of course. We have identified actual bugs in the benchmarks, among the places where Julia signals a possible warning. However, we cannot check by hand hundreds of warnings, on third-party code. Nevertheless, the nullness analysis of Julia is currently the most precise available on the market [38] and scales to very large software, as our experiments here show. The termination analysis of Julia scales almost in the same way, as shown below, and we have never seen any other report of a termination analysis that scales to that size of programs. Nevertheless, this article is not about nullness nor termination nor their precision. We use those analyses only to support the thesis that definite aliasing analysis is useful to support other analyses.

Fig. 13 reports the times for nullness and termination analysis. These are total times, that is, they include everything: from the parsing of the `.jar` files, to ours and other supporting analyses, to the presentation of the warnings to the user. In that figure, we have copied the times for the expression aliasing analysis alone, to highlight the fact that they are only a small fraction of the total times for nullness and termination analysis. When the expression aliasing analysis is turned off, times are in general smaller, also because there is fewer information to exploit for nullness and termination proofs. For instance, when that information is missing, it is sometimes the case that a symbolic upper bound for a loop variable is missed, which results in the immediate failure of the termination proof but in coarser results. The termination proof for h2 went into out of memory, so we do not report times for it in the figure.

Figures 14 and 15 report the precision of the same nullness and termination analyses. They show how fewer warnings are issued by Julia after those analyses, if our aliasing expression analysis is turned on. Fig. 14 shows that the number of warnings for nullness analysis is in general halved; sometimes, there are no more warnings, thanks to the support of the expression aliasing analysis (as for benchmarks 6, 14 and 17); only in three cases there are no benefits (benchmarks 9, 11 and 16). The situation is similar for termination analysis (Fig. 15), but the gain in precision is less evident here.

ID	EXP	NULLNESS		TERMINATION	
	ALIASING	WITH	WITHOUT	WITH	WITHOUT
1	0.51	12.64	11.73	19.66	13.96
2	1.28	27.81	24.81	17.13	15.37
3	3.29	78.50	62.93	115.21	93.90
4	1.96	54.83	48.75	79.61	63.34
5	2.63	74.98	78.15	86.21	73.63
6	2.40	50.88	45.61	27.60	24.92
7	2.22	51.21	47.25	29.76	26.86
8	2.34	57.95	54.69	31.61	31.82
9	3.49	73.89	68.12	47.01	40.88
10	6.14	138.66	119.75	52.81	76.14
11	3.11	87.92	86.48	51.80	42.70
12	3.23	77.11	72.82	49.51	45.76
13	4.31	71.08	66.16	42.05	37.72
14	3.03	78.25	71.93	47.67	43.65
15	3.15	94.59	83.16	51.00	47.77
16	3.14	75.35	70.58	49.91	45.71
17	3.13	80.64	72.09	50.52	46.76
18	3.32	91.22	85.76	49.58	48.84
19	3.18	125.16	112.97	354.19	248.65
20	3.46	107.98	100.43	96.55	89.63
21	3.36	96.42	91.39	54.63	53.76
22	3.73	99.40	92.58	56.91	51.60
23	3.47	98.28	92.58	62.96	55.58
24	3.56	98.20	92.88	56.94	51.24
25	6.18	296.79	269.18	816.06	904.79
26	5.31	109.28	105.20	61.11	59.40
27	4.37	121.69	113.95	72.17	61.36
28	43.45	166.83	109.68	95.87	65.18
29	40.34	188.86	130.80	105.50	69.22
30	5.06	155.79	138.68	80.66	76.33
31	5.07	183.82	161.70	82.77	74.16
32	5.24	167.97	146.02	85.87	76.55
33	5.35	182.31	178.20	86.33	78.64
34	5.22	191.19	170.41	87.26	75.08
35	5.35	167.06	163.04	88.13	79.55
36	6.09	188.15	173.07	88.42	84.29
37	6.07	254.84	257.70	136.21	121.86
38	6.57	247.31	233.59	132.87	114.89
39	15.42	617.00	600.70	1090.51	1630.15
40	7.54	258.14	242.33	126.09	104.66
41	7.01	307.38	277.47	151.55	136.46
42	5.91	623.97	638.86	638.47	561.45
43	6.59	341.38	322.75	164.98	148.97
44	14.99	1017.42	1007.89	N.A.	N.A.

Fig. 13 Total time (in seconds) for the nullness and termination analysis of our benchmarks, each computed in two versions: with our expression aliasing analysis and without our expression aliasing analysis. We also report the cost of just the expression aliasing analysis, for comparison. The cost of nullness and termination includes everything: from the parsing of the `jar` files to the presentation of the warnings to the user. Hence they also include the times of the expression aliasing analysis. The termination analysis of benchmark 44 (`h2`) could not be completed because of an out of memory during the termination proof (while our expression aliasing analysis could be completed in 14.99 seconds and less than one gigabyte of RAM).

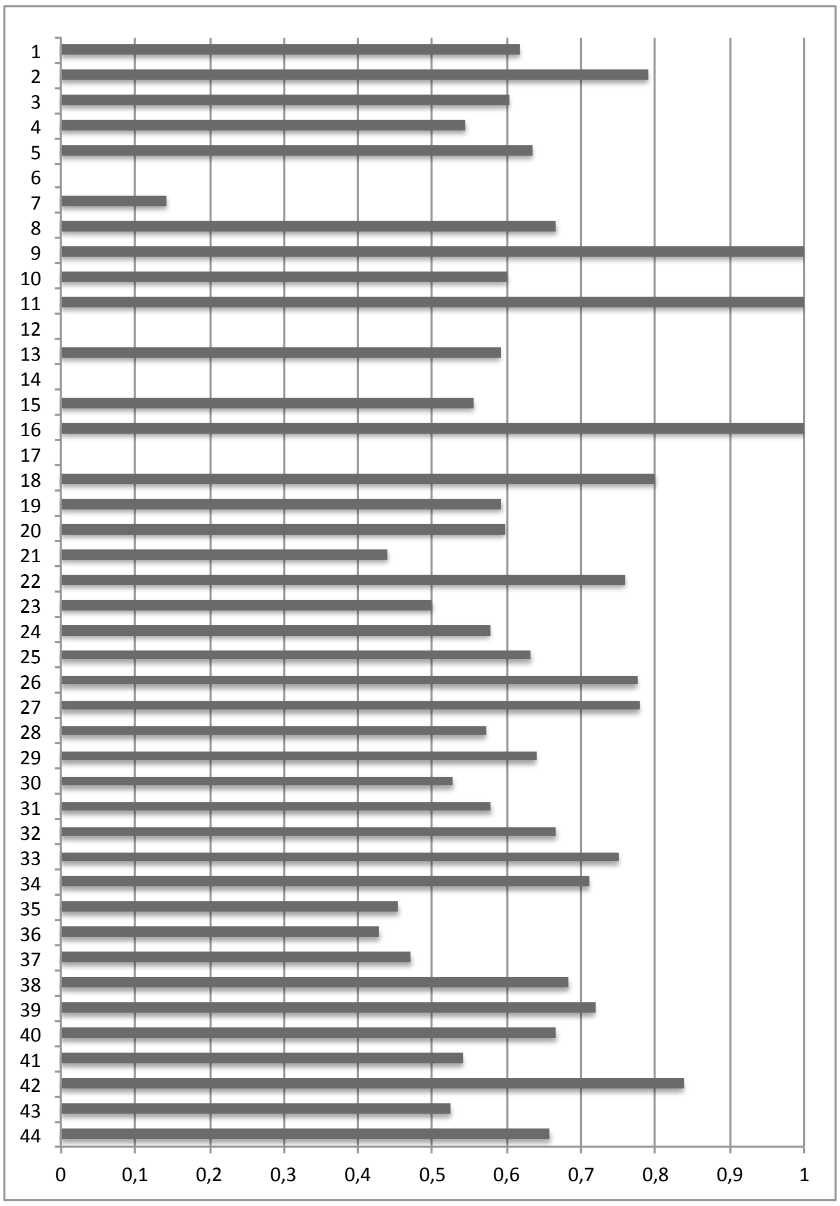


Fig. 14 The gain in precision due to our expression aliasing analysis, for the nullness analyses. For each benchmark, it shows how much is gained by the use of the expression aliasing information. For instance, the number of warnings for benchmark 1 (`nti`), using the aliasing information, is only 61% of the number of warnings for the same benchmark when no aliasing information is used.

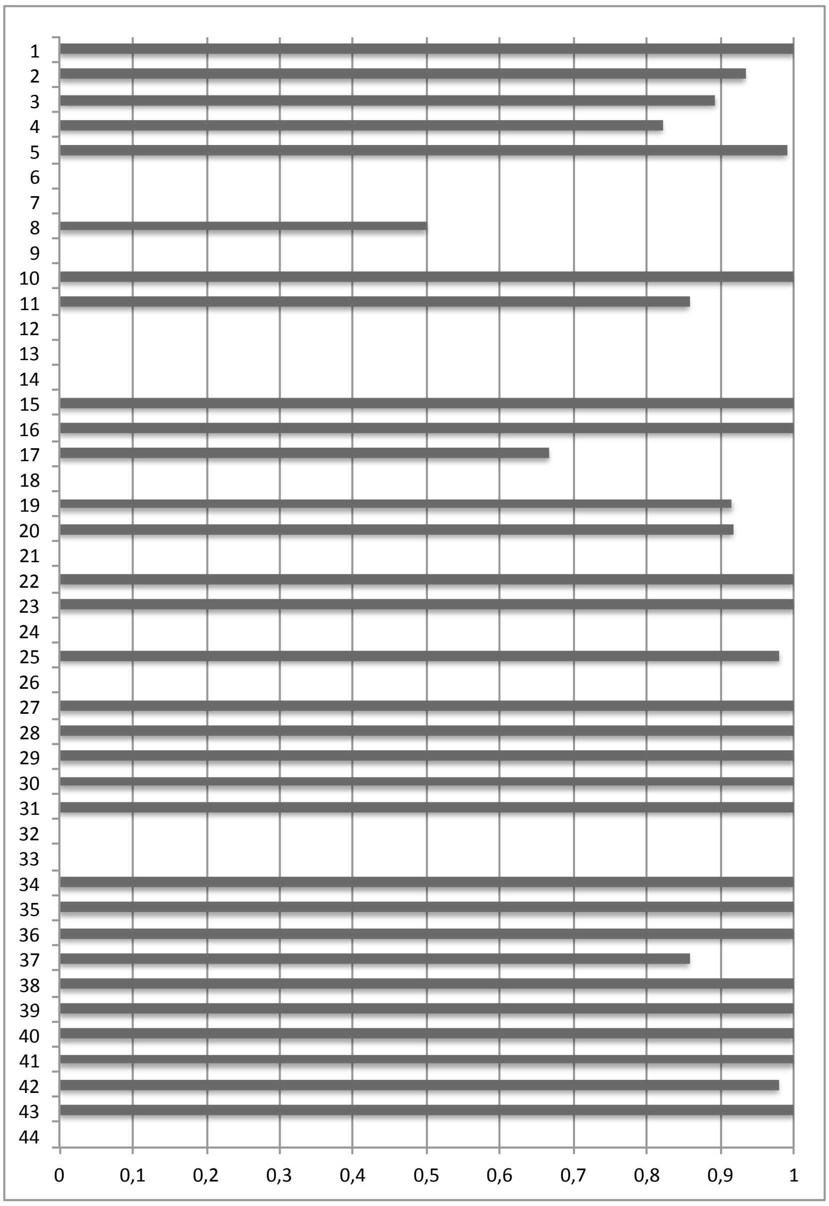


Fig. 15 The gain in precision due to our expression aliasing analysis, for the termination analyses. For each benchmark, it shows how much is gained by the use of the expression aliasing information. For instance, the number of warnings for benchmark 4 (`JFlex`), using the aliasing information, is only 82% of the number of warnings for the same benchmark when no aliasing information is used.

6 Conclusion

We have introduced a static analysis called *definite expression aliasing analysis*, which provides, for each program point p and each variable v, a set of expressions E such that the values of E and v at point p coincide, for every possible execution path. Our static analysis is based on abstract interpretation, that we use to formally prove its correctness. The implementation of our inter-procedural analysis inside the Julia static analyzer handles full Java bytecode with exceptions, but not multithreading nor reflection. We performed an experimental evaluation on some real-life benchmarks, that showed the scalability of our analysis and the benefits that it brings for subsequent nullness and termination analysis.

Our definite expression aliasing analysis is an instance of a more general constraint-based static analysis for Java bytecode, that we have already applied to infer a safe over-approximation of the program variables, fields, or array elements that, at run time, might hold partially initialized (raw) objects [39] and a safe over-approximation of the pairs of program variables of reference type $\langle a, b\rangle$ such that, at runtime, the location bound to a might reach the location bound to b [29]. The use of a constraint is similar to [39] and rather traditional in static analysis. But the difference and real complexity of this analysis lays in the propagation rules for the arcs of the constraint graph, that must take into account the side-effects on the heap. Moreover, this is the first example of a definite analysis and shows that the same graph-based technique can be used for possible as well as definite static analysis. In particular, to the best of our knowledge, this is the first definite aliasing analysis dealing with Java bytecode programs and with expressions (not only program variables) aliased to program variables.

References

1. Soot: A Java Optimization Framework - `http://www.sable.mcgill.ca/soot/`
2. WALA: T.J. Watson Libraries for Analysis - `http://wala.sourceforge.net/`
3. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1986)
4. Aiken, A.: Introduction to Set Constraint-Based Program Analysis. Science of Computer Programming **35**(2), 79–111 (1999)
5. Aiken, A., Wimmers, E.L.: Type Inclusion Constraints and Type Inference. In: Proceedings of Conference on Functional Programming Languages and Computer Architecture (FPCA 1993), pp. 31–41. ACM (1993)
6. Albert, E., Arenas, P., Genaim, S., Puebla, G., Ramírez, D.: From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In: Proceedings of the 17th Static Analysis Symposium (SAS 2010), *LNCS*, vol. 6337, pp. 100–116. Springer (2010)
7. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: Proceedings of the 16th European Symposium on Programming (ESOP 2007), pp. 157–172. Springer (2007)
8. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting Equality of Variables in Programs. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1988), pp. 1–11. ACM (1988)
9. Anderson, C., Giannini, P., Drossopoulou, S.: Towards Type Inference for JavaScript. In: Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005), *LNCS*, vol. 3586, pp. 428–452. Springer (2005)
10. Bradley, A.R., Manna, Z.: Verification Constraint Problems with Strengthening. In: Proceedings of the 3th International Colloquium on Theoretical Aspects of Computing (ICTAC 2006), *LNCS*, vol. 4281, pp. 35–49. Springer (2006)
11. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear Invariant Generation Using Non-linear Constraint Solving. In: Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003), *LNCS*, vol. 2725, pp. 420–432. Springer (2003)

12. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of the 4th Symposium on Principles of Programming Languages (POPL 1977), pp. 238–252. ACM (1977)
13. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Proceedings of the 6th Symposium on Principles of Programming Languages (POPL 1979), pp. 269–282. ACM (1979)
14. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press (1990)
15. Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective Typestate Verification in the Presence of Aliasing. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 133–144. ACM (2006)
16. Gulwani, S., Necula, G.C.: Global Value Numbering Using Random Interpretation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004), pp. 342–352. ACM (2004)
17. Gulwani, S., Necula, G.C.: A Polynomial-Time Algorithm for Global Value Numbering. Science of Computer Programming **64**(1), 97–114 (2007)
18. Gulwani, S., Srivastava, S., Venkatesan, R.: Program Analysis as Constraint Solving. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008), pp. 281–292. ACM (2008)
19. Hind, M.: Pointer Analysis: Haven't We Solved This Problem Yet? In: Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), pp. 54–61. ACM (2001)
20. Kildall, G.A.: A Unified Approach to Global Program Optimization. In: Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1973), pp. 194–206. ACM (1973)
21. Knoop, J., Rüthing, O., Steffen, B.: Lazy Code Motion. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1992), pp. 224–234. ACM (1992)
22. Lindholm, T., Yellin, F.: The Java^{TM} Virtual Machine Specification, second edn. Addison-Wesley (1999)
23. Logozzo, F., Fähndrich, M.: On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In: Proceedings of the 17th International Conference on Compiler Construction (CC 2008), *LNCS*, vol. 4959, pp. 197–212. Springer (2008)
24. Morel, E., Renvoise, C.: Global Optimization by Suppression of Partial Redundancies. Communications of the ACM **22**(2), 96–103 (1979)
25. Müller-Olm, M., Rüthing, O., Seidl, H.: Checking Herbrand Equalities and Beyond. In: Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005), pp. 79–96. Springer-Verlag, Berlin, Heidelberg (2005)
26. Müller-Olm, M., Seidl, H., Steffen, B.: Interprocedural Herbrand Equalities. In: Proceedings of the 14th European Symposium on Programming (ESOP 2005), pp. 31–45. Springer-Verlag, Berlin, Heidelberg (2005)
27. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, corrected edn. Springer (2004)
28. Nikolić, Đ., Spoto, F.: Definite Expression Aliasing Analysis for Java Bytecode. In: Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing (ICTAC 2012), *LNCS*, vol. 7521, pp. 74–89. Springer-Verlag Berlin Heidelberg (2012)
29. Nikolić, Đ., Spoto, F.: Reachability Analysis of Program Variables. In: Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR 2012), *LNAI*, vol. 7364, pp. 423–438. Springer-Verlag Berlin Heidelberg (2012)
30. Ohata, F., Inoue, K.: JAAT: Java Alias Analysis Tool for Program Maintenance Activities. In: Proceedings of the 9th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2006), pp. 232–244. IEEE (2006)
31. Palsberg, J., Schwartzbach, M.I.: Object-oriented Type Inference. In: Proceedings of Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 1991), *ACM SIGPLAN Notices*, vol. 26(11), pp. 146–161. ACM (1991)
32. Payet, É., Spoto, F.: Magic-Sets Transformation for the Analysis of Java Bytecode. In: Proceedings of the 14th International Symposium on Static Analysis (SAS 2007), *LNCS*, vol. 4634, pp. 452–467. Springer (2007)
33. Payet, É., Spoto, F.: Static Analysis of Android Programs. Information & Software Technology **54**(11), 1192–1201 (2012)
34. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global Value Numbers and Redundant Computations. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1988), pp. 12–27. ACM (1988)
35. Rüthing, O., Knoop, J., Steffen, B.: Detecting Equalities of Variables: Combining Efficiency with Precision. In: Proceedings of the 6th International Symposium on Static Analysis (SAS 1999), pp. 232–247. Springer-Verlag (1999)

36. Sankaranarayanan, S., Sipma, H., Manna, Z.: Constraint-Based Linear-Relations Analysis. In: Proceedings of the 11th International Symposium on Static Analysis (SAS 2004), vol. 3148, pp. 53–68. Springer (2004)
37. Spoto, F.: Nullness Analysis in Boolean Form. In: Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008), pp. 21–30. IEEE (2008)
38. Spoto, F.: Precise Null-pointer Analysis. Software and System Modeling **10**(2), 219–252 (2011)
39. Spoto, F., Ernst, M.D.: Inference of Field Initialization. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), pp. 231–240. ACM (2011)
40. Spoto, F., Mesnard, F., Payet, E.: A Termination Analyzer for Java Bytecode Based on Path-Length. ACM Transactions on Programming Languages and Systems **32**(3), 1–70 (2010)
41. Tarski, A.: A Lattice Theoretical Fixpoint Theorem and its Applications. Pacific Journal of Mathematics **5**, 285–310 (1955)
42. Wang, T., Smith, S.F.: Precise Constraint-Based Type Inference for Java. In: Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001), *LNCS*, vol. 2072, pp. 99–117. Springer (2001)

A Alias Expressions

In the following we enunciate and prove some technical lemmas. Lemma 3 states that if two environments ρ and ρ' coincide on all the variables appearing in an arbitrary expression E but a given variable a, and if, for a given constant n, $\rho'(a) \oplus n = \rho(a)$ holds, then, for an arbitrary memory μ, the evaluation of E in $\langle \rho, \mu \rangle$ coincides with the evaluation of E where all occurrences of a are replaced by $a \oplus n$ in $\langle \rho', \mu \rangle$.

Lemma 3 *Consider a type environment $\tau \in \mathcal{T}$, a variable $a \in \mathsf{dom}(\tau)$, a constant $n \in \mathbb{Z}$ and two environments ρ and ρ' such that $\rho'(a) \oplus n = \rho(a)$. Let $\mathsf{E} \in \mathbb{E}_\tau$ be an arbitrary expression. If for every variable $v \in \mathsf{variables}(\mathsf{E}) \smallsetminus \{a\}$, $\rho'(v) = \rho(v)$ holds, then for every memory μ,*

$$[\![\mathsf{E}[(a \oplus n)/a]]\!]^*\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]^*\langle \rho, \mu \rangle,$$

where $\mathsf{E}[(a \oplus n)/a]$ denotes the expression E with all the occurrences of a replaced with $a \oplus n$.

Proof. We proof this lemma by induction on the depth of E.
Base case: Let μ be an arbitrary memory. If $\mathsf{depth}(\mathsf{E}) = 0$, then $\mathsf{E} = x \in \mathbb{V}$ or $\mathsf{E} = v \in \mathsf{dom}(\tau)$. In the former case, by Definition 14,

$$[\![x]\!]^*\langle \rho, \mu \rangle = \langle x, \mu \rangle = [\![x]\!]^*\langle \rho', \mu \rangle = [\![x[(a \oplus n)/a]]\!]^*\langle \rho', \mu \rangle.$$

In the latter case, if $v \neq a$ then, by Definition 14,

$$[\![v]\!]^*\langle \rho, \mu \rangle = \langle \rho(v), \mu \rangle = \langle \rho'(v), \mu \rangle = [\![v]\!]^*\langle \rho', \mu \rangle = [\![v[(a \oplus n)/a]]\!]^*\langle \rho', \mu \rangle.$$

If $v = a$, by hypothesis, $\rho(a) = \rho'(a) \oplus n$ and by Definition 14, we obtain

$$[\![a]\!]^*\langle \rho, \mu \rangle = \langle \rho(a), \mu \rangle = \langle \rho'(a) \oplus n, \mu \rangle = [\![(a \oplus n)]\!]^*\langle \rho', \mu \rangle = [\![a[(a \oplus n)/a]]\!]^*\langle \rho', \mu \rangle.$$

Induction: Suppose that for every expression E' of depth at most k hypothesis holds, i.e., if for every variable $v \in \mathsf{variables}(\mathsf{E}') \smallsetminus \{a\}$, $\rho'(v) = \rho(v)$ holds, then for every memory μ, $[\![\mathsf{E}[(a \oplus n)/a]]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle$ holds. Let E be an expression such that $\mathsf{depth}(\mathsf{E}) = k + 1$. If there exists a variable $v \in \mathsf{variables}(\mathsf{E}) \smallsetminus \{a\}$ such that $\rho(v) \neq \rho(v')$, then the result trivially holds. Otherwise, $\forall v \in \mathsf{variables}(\mathsf{E}) \smallsetminus \{a\}. \rho'(v) = \rho(v)$ holds and we distinguish different possible forms of E:

– <u>If $\mathsf{E} = \mathsf{E}_1 \oplus \mathsf{E}_2$</u>, we have $k + 1 = \mathsf{depth}(\mathsf{E}) = 1 + \max\{\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2)\}$ (Definition 13), which entails $\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2) \leq k$. Since $\mathsf{variables}(\mathsf{E}) = \mathsf{variables}(\mathsf{E}_1) \cup \mathsf{variables}(\mathsf{E}_2)$, and for every $v \in \mathsf{variables}(\mathsf{E}) \smallsetminus \{a\}$, $\rho'(v) = \rho(v)$, we conclude that ρ and ρ' also agree on the values of all the variables different from a which appear in E_1 and E_2. Let μ be an arbitrary memory, then by inductive hypothesis on E_1 and E_2 we have:

$$\begin{array}{lcl} [\![\mathsf{E}_1]\!]^*\langle \rho, \mu \rangle &=& [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle \rho', \mu \rangle &=& \langle w_1, \mu_1 \rangle \\ [\![\mathsf{E}_2]\!]^*\langle \rho, \mu_1 \rangle &=& [\![\mathsf{E}_2[(a \oplus n)/a]]\!]^*\langle \rho', \mu_1 \rangle &=& \langle w_2, \mu_2 \rangle, \end{array}$$

where $w_1, w_2 \in \mathbb{Z}$. Therefore:

$$\begin{array}{rcll} [\![\mathsf{E}]\!]^*\langle \rho, \mu \rangle &=& [\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\langle \rho, \mu \rangle & \\ &=& \langle w_1 \oplus w_2, \mu_2 \rangle & \text{[By Definition 14]} \\ &=& [\![\mathsf{E}_1[(a \oplus n)/a] \oplus \mathsf{E}_2[(a \oplus n)/a]]\!]^*\langle \rho', \mu \rangle & \text{[By hypothesis and Def. 14]} \\ &=& [\![\mathsf{E}[(a \oplus n)/a]]\!]^*\langle \rho', \mu \rangle. & \end{array}$$

– <u>If $\mathsf{E} = \mathsf{E}_1.f$</u>, then $k+1 = \mathsf{depth}(\mathsf{E}) = 1 + \mathsf{depth}(\mathsf{E}_1)$ (Definition 13), which entails $\mathsf{depth}(\mathsf{E}_1) = k$. Since $\mathsf{variables}(\mathsf{E}) = \mathsf{variables}(\mathsf{E}_1)$, we have that for every $v \in \mathsf{variables}(\mathsf{E}_1) \smallsetminus \{a\}$, $\rho'(v) = \rho(v)$ and, by inductive hypothesis on E_1, $[\![\mathsf{E}_1]\!]^*\langle \rho, \mu \rangle = [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle \rho', \mu \rangle = \langle \ell, \mu_1 \rangle$, where $\ell \in \mathbb{L}$ and $\mu_1(\ell).\kappa \in \mathbb{K}$. Therefore:

$$\begin{array}{rcll} [\![\mathsf{E}]\!]^*\langle \rho, \mu \rangle &=& [\![\mathsf{E}_1.f]\!]^*\langle \rho, \mu \rangle & \\ &=& \langle (\mu_1(\ell).\phi)(f), \mu_1 \rangle & \text{[By Definition 14]} \\ &=& [\![\mathsf{E}_1[(a \oplus n)/a].f]\!]^*\langle \rho', \mu \rangle & \text{[By hypothesis and Definition 14]} \\ &=& [\![\mathsf{E}[(a \oplus n)/a]]\!]^*\langle \rho', \mu \rangle. & \end{array}$$

– If $\mathsf{E} = \mathsf{E}_1.\mathtt{length}$, then $k+1 = \mathsf{depth}(\mathsf{E}) = 1+\mathsf{depth}(\mathsf{E}_1)$ (Definition 13), which entails $\mathsf{depth}(\mathsf{E}_1) = k$. Since $\mathsf{variables}(\mathsf{E}) = \mathsf{variables}(\mathsf{E}_1)$, we have that for every $v \in \mathsf{variables}(\mathsf{E}_1) \smallsetminus \{a\}$, $\rho'(v) = \rho(v)$ and, by inductive hypothesis on E_1, $[\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle = [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle\rho',\mu\rangle = \langle\ell,\mu_1\rangle$, where $\ell \in \mathbb{L}$ and $\mu_1(\ell).\kappa \in \mathbb{A}$. Therefore:

$$\begin{aligned}
[\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_1.\mathtt{length}]\!]^*\langle\rho,\mu\rangle \\
&= \langle\mu_1(\ell).\mathtt{length},\mu_1\rangle && \text{[By Definition 14]} \\
&= [\![\mathsf{E}_1[(a \oplus n)/a].\mathtt{length}]\!]^*\langle\rho',\mu\rangle && \text{[By hypothesis and Definition 14]} \\
&= [\![\mathsf{E}[(a \oplus n)/a]]\!]^*\langle\rho',\mu\rangle.
\end{aligned}$$

– If $\mathsf{E} = \mathsf{E}_1[\mathsf{E}_2]$, we have $k + 1 = \mathsf{depth}(\mathsf{E}) = 1 + \max\{\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2)\}$ (Definition 13), which entails $\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2) \leq k$. Since $\mathsf{variables}(\mathsf{E}) = \mathsf{variables}(\mathsf{E}_1) \cup \mathsf{variables}(\mathsf{E}_2)$, and for every $v \in \mathsf{variables}(\mathsf{E}) \smallsetminus \{a\}$, $\rho'(v) = \rho(v)$, we conclude that ρ and ρ' also agree on the values of all the variables different from a which appear in E_1 and E_2. Let μ be an arbitrary memory, then by inductive hypothesis on E_1 and E_2 we have:

$$\begin{aligned}
[\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle\rho',\mu\rangle &&= \langle\ell,\mu_1\rangle \\
[\![\mathsf{E}_2]\!]^*\langle\rho,\mu_1\rangle &= [\![\mathsf{E}_2[(a \oplus n)/a]]\!]^*\langle\rho',\mu_1\rangle &&= \langle k,\mu_2\rangle,
\end{aligned}$$

where $\ell \in \mathbb{L}$, $\mu_2(\ell).\kappa \in \mathbb{A}$ and $k \in \mathbb{Z}$. Therefore:

$$\begin{aligned}
[\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_1[\mathsf{E}_2]]\!]^*\langle\rho,\mu\rangle \\
&= \langle(\mu_2(\ell).\phi)(k),\mu_2\rangle && \text{[By Definition 14]} \\
&= [\![\mathsf{E}_1[(a \oplus n)/a][\mathsf{E}_2[(a \oplus n)/a]]]\!]^*\langle\rho',\mu\rangle && \text{[By hypothesis and Def. 14]} \\
&= [\![\mathsf{E}[(a \oplus n)/a]]\!]^*\langle\rho',\mu\rangle.
\end{aligned}$$

– If $\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1,\ldots,\mathsf{E}_\pi)$, we have $k + 1 = \mathsf{depth}(\mathsf{E}) = 1 + \max_{0 \leq i \leq \pi}\{\mathsf{depth}(\mathsf{E}_i)\}$, hence $\mathsf{depth}(\mathsf{E}_i) \leq k$, for each $0 \leq i \leq \pi$. Since $\mathsf{variables}(\mathsf{E}) = \bigcup_{i=0}^{\pi} \mathsf{variables}(\mathsf{E}_i)$, and for every $v \in \mathsf{variables}(\mathsf{E}) \smallsetminus \{a\}$, $\rho'(v) = \rho(v)$, we conclude that ρ and ρ' agree on the values of all the variables different from a which appear in each E_i. Let μ be an arbitrary memory, then by inductive hypothesis on E_1 and E_2 we have:

$$\begin{aligned}
[\![\mathsf{E}_0]\!]^*\langle\rho,\mu\rangle &= [\![\mathsf{E}_0[(a \oplus n)/a]]\!]^*\langle\rho',\mu\rangle &&= \langle w_0,\mu_0\rangle \\
[\![\mathsf{E}_1]\!]^*\langle\rho,\mu_0\rangle &= [\![\mathsf{E}_1[(a \oplus n)/a]]\!]^*\langle\rho',\mu_0\rangle &&= \langle w_1,\mu_1\rangle \\
&\ \ \cdots \\
[\![\mathsf{E}_\pi]\!]^*\langle\rho,\mu_{\pi-1}\rangle &= [\![\mathsf{E}_\pi[(a \oplus n)/a]]\!]^*\langle\rho',\mu_{\pi-1}\rangle &&= \langle w_\pi,\mu_\pi\rangle
\end{aligned}$$

Hence, for each $1 \leq i \leq \pi$, evaluations of both E_i and $\mathsf{E}_i[(a \oplus n)/a]$ in $\langle\rho,\mu_{i-1}\rangle$ and $\langle\rho',\mu_{i-1}\rangle$ respectively give equal results $\langle w_i,\mu_i\rangle$ and, by Definition 14, it implies that evaluations of both E in $\langle\rho,\mu\rangle$ and $\mathsf{E}[(a \oplus n)/a]$ in $\langle\rho',\mu\rangle$ are equal and correspond to the value returned by the method m. Namely, in both cases, the execution of m is deterministic since we fixed the actual parameters (receiver $\mu_\pi(w_0)$ and parameters $w_1,\ldots,w_\pi$) and the memory (μ_π), hence in both cases it will produce the same return value. This value is enriched with the resulting memory μ' obtained from μ_π as a side-effect of m's execution.

$\square$

One particular case of Lemma 3 is when the constant n is 0. Namely, when ρ and ρ' coincide on all the variables appearing in E then the evaluations of the latter in $\langle\rho,\mu\rangle$ and $\langle\rho',\mu\rangle$ coincide too.

Corollary 1 *Consider a type environment $\tau \in \mathcal{T}$ and two environments ρ and ρ'. Let $\mathsf{E} \in \mathbb{E}_\tau$ be an arbitrary expression. If for every variable $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$ holds, then for every memory μ,*

$$[\![\mathsf{E}]\!]^*\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle.$$

Similarly, we prove another important result. Lemma 4 states that if a state assigns the same values to two fixed variables a and b, then the evaluation of an arbitrary expression in that state does not change if we replace all the occurrences of a with b.

Lemma 4 *Consider a type environment $\tau \in \mathcal{T}$, variables $a, b \in \mathsf{dom}(\tau)$ and an environment ρ such that $\rho(a) = \rho(b)$. Let $\mathsf{E} \in \mathbb{E}_\tau$ be an arbitrary expression. Then, for every memory μ,*

$$[\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle = [\![\mathsf{E}[b/a]]\!]^*\langle\rho,\mu\rangle,$$

where $\mathsf{E}[b/a]$ denotes the expression E with all the occurrences of a replaced with b.

Proof. We proof this lemma by induction on the depth of E.
Base case: Let μ be an arbitrary memory. If $\mathsf{depth}(\mathsf{E}) = 0$, then $\mathsf{E} = n \in \mathbb{V}$ or $\mathsf{E} = v \in \mathsf{dom}(\tau)$. In the former case, by Definition 14,

$$[\![n]\!]^*\langle\rho,\mu\rangle = \langle n,\mu\rangle = [\![n[b/a]]\!]^*\langle\rho,\mu\rangle.$$

In the latter case, if $v \neq a$ then, by Definition 14,

$$[\![v]\!]^*\langle\rho,\mu\rangle = \langle\rho(v),\mu\rangle = \langle\rho'(v),\mu\rangle = [\![v[b/a]]\!]^*\langle\rho,\mu\rangle.$$

If $v = a$, by hypothesis, $\rho(a) = \rho(b)$ and by Definition 14, we obtain

$$[\![a]\!]^*\langle\rho,\mu\rangle = \langle\rho(a),\mu\rangle = \langle\rho(b),\mu\rangle = [\![b]\!]^*\langle\rho,\mu\rangle = [\![a[b/a]]\!]^*\langle\rho,\mu\rangle.$$

Induction: Suppose that for every expression E' of depth at most k hypothesis holds, i.e., $[\![\mathsf{E}']\!]^*\langle\rho,\mu\rangle = [\![\mathsf{E}'[b/a]]\!]^*\langle\rho,\mu\rangle$, for every memory μ. Let E be an expression such that $\mathsf{depth}(\mathsf{E}) = k+1$. We show that $[\![\mathsf{E}]\!]^*\sigma = [\![\mathsf{E}[b/a]]\!]^*\sigma$. We distinguish different possible forms of E:

- $\underline{\text{If } \mathsf{E} = \mathsf{E}_1 \oplus \mathsf{E}_2}$, we have $k+1 = \mathsf{depth}(\mathsf{E}) = 1 + \max\{\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2)\}$ (Definition 13), which entails $\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2) \leq k$. Therefore, inductive hypothesis holds on both E_1 and E_2. More precisely, inductive hypothesis entails

$$\begin{array}{lll} [\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle & = [\![\mathsf{E}_1[(a\oplus n)/a]]\!]^*\langle\rho',\mu\rangle & = \langle w_1,\mu_1\rangle \\ [\![\mathsf{E}_2]\!]^*\langle\rho,\mu_1\rangle & = [\![\mathsf{E}_2[(a\oplus n)/a]]\!]^*\langle\rho',\mu_1\rangle & = \langle w_2,\mu_2\rangle, \end{array}$$

  where $w_1, w_2 \in \mathbb{Z}$. Therefore:

$$\begin{array}{lll} [\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle & = [\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\langle\rho,\mu\rangle & \\ & = \langle w_1 \oplus w_2, \mu_2\rangle & \text{[By Definition 14]} \\ & = [\![\mathsf{E}_1[b/a] \oplus \mathsf{E}_2[b/a]]\!]^*\langle\rho,\mu\rangle & \text{[By hypothesis and Definition 14]} \\ & = [\![\mathsf{E}[b/a]]\!]^*\langle\rho,\mu\rangle. & \end{array}$$

- $\underline{\text{If } \mathsf{E} = \mathsf{E}_1.f}$, we have $k+1 = \mathsf{depth}(\mathsf{E}) = 1 + \mathsf{depth}(\mathsf{E}_1)$, which entails $\mathsf{depth}(\mathsf{E}_1) = k$, and therefore, hypothesis holds on it, i.e.,

$$[\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle = [\![\mathsf{E}_1[b/a]]\!]^*\langle\rho,\mu\rangle = \langle\ell,\mu_1\rangle,$$

  where $\ell \in \mathbb{L}$ and $\mu_1(\ell).\kappa \in \mathbb{K}$. We have

$$\begin{array}{lll} [\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle & = [\![\mathsf{E}_1.f]\!]^*\langle\rho,\mu\rangle & \\ & = \langle(\mu_1(\ell).\phi)(f),\mu_1\rangle & \text{[By Definition 14]} \\ & = [\![\mathsf{E}_1[b/a].f]\!]^*\langle\rho',\mu\rangle & \text{[By hypothesis and Definition 14]} \\ & = [\![\mathsf{E}[b/a]]\!]^*\langle\rho',\mu\rangle. & \end{array}$$

- $\underline{\text{If } \mathsf{E} = \mathsf{E}_1.\mathtt{length}}$, we have $k+1 = \mathsf{depth}(\mathsf{E}) = 1 + \mathsf{depth}(\mathsf{E}_1)$, which entails $\mathsf{depth}(\mathsf{E}_1) = k$, and therefore, hypothesis holds on it, i.e.,

$$[\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle = [\![\mathsf{E}_1[b/a]]\!]^*\langle\rho,\mu\rangle = \langle\ell,\mu_1\rangle,$$

  where $\ell \in \mathbb{L}$ and $\mu_1(\ell).\kappa \in \mathbb{A}$. We have

$$\begin{array}{lll} [\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle & = [\![\mathsf{E}_1.\mathtt{length}]\!]^*\langle\rho,\mu\rangle & \\ & = \langle\mu_1(\ell).\mathtt{length},\mu_1\rangle & \text{[By Definition 14]} \\ & = [\![\mathsf{E}_1[b/a].\mathtt{length}]\!]^*\langle\rho',\mu\rangle & \text{[By hypothesis and Definition 14]} \\ & = [\![\mathsf{E}[b/a]]\!]^*\langle\rho',\mu\rangle. & \end{array}$$

- $\underline{\text{If } \mathsf{E} = \mathsf{E}_1[\mathsf{E}_2]}$, we have $k+1 = \mathsf{depth}(\mathsf{E}) = 1+\max\{\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2)\}$ (Definition 13), which entails $\mathsf{depth}(\mathsf{E}_1), \mathsf{depth}(\mathsf{E}_2) \leq k$. Therefore, inductive hypothesis holds on both E_1 and E_2. More precisely, inductive hypothesis entails

$$\begin{array}{lll} [\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle & = [\![\mathsf{E}_1[(a\oplus n)/a]]\!]^*\langle\rho',\mu\rangle & = \langle\ell,\mu_1\rangle \\ [\![\mathsf{E}_2]\!]^*\langle\rho,\mu_1\rangle & = [\![\mathsf{E}_2[(a\oplus n)/a]]\!]^*\langle\rho',\mu_1\rangle & = \langle k,\mu_2\rangle, \end{array}$$

  where $\ell \in \mathbb{L}$, $\mu_2(\ell).\kappa \in \mathbb{A}$ and $k \in \mathbb{Z}$. Therefore:

$$\begin{array}{lll} [\![\mathsf{E}]\!]^*\langle\rho,\mu\rangle & = [\![\mathsf{E}_1[\mathsf{E}_2]]\!]^*\langle\rho,\mu\rangle & \\ & = \langle(\mu_2(\ell).\phi)(k),\mu_2\rangle & \text{[By Definition 14]} \\ & = [\![\mathsf{E}_1[b/a][\mathsf{E}_2[b/a]]]\!]^*\langle\rho,\mu\rangle & \text{[By hypothesis and Definition 14]} \\ & = [\![\mathsf{E}[b/a]]\!]^*\langle\rho,\mu\rangle. & \end{array}$$

– If $\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1,\ldots,\mathsf{E}_\pi)$, we have $k+1 = \mathsf{depth}(\mathsf{E}) = 1 + \max_{0\le i\le\pi}\{\mathsf{depth}(\mathsf{E}_i)\}$, hence $\mathsf{depth}(\mathsf{E}_i) \le k$, for each $0 \le i \le \pi$. Thus, hypothesis holds on each E_i, which entails:

$$\begin{array}{rcl} [\![\mathsf{E}_0]\!]^*\langle\rho,\mu\rangle &=& [\![\mathsf{E}_0[b/a]]\!]^*\langle\rho,\mu\rangle \;=\; \langle w_0,\mu_0\rangle \\ [\![\mathsf{E}_1]\!]^*\langle\rho,\mu_0\rangle &=& [\![\mathsf{E}_1[b/a]]\!]^*\langle\rho,\mu_0\rangle \;=\; \langle w_1,\mu_1\rangle \\ && \ldots \\ [\![\mathsf{E}_\pi]\!]^*\langle\rho,\mu_{\pi-1}\rangle &=& [\![\mathsf{E}_\pi[b/a]]\!]^*\langle\rho,\mu_{\pi-1}\rangle = \langle w_\pi,\mu_\pi\rangle. \end{array}$$

Hence, for each $1 \le i \le \pi$, evaluation of both E_i and $\mathsf{E}_i[b/a]$ in $\langle\rho,\mu_{i-1}\rangle$ and $\langle\rho',\mu_{i-1}\rangle$ respectively gives equal result $\langle w_i,\mu_i\rangle$ and, by Definition 14, it implies that evaluations of both E and $\mathsf{E}[b/a]$ in $\langle\rho,\mu\rangle$ are equal and correspond to the value returned by the method m. Namely, in both cases, the execution of m is deterministic since we fixed the actual parameters (receiver $\mu_\pi(w_0)$ and parameters $w_1,\ldots,w_\pi$) and the memory (μ_π), hence in both cases it will produce the same return value. This value is enriched with the resulting memory μ' obtained from μ_π as a side-effect of m's execution.

□

B Definite Expression Aliasing Analysis

B.1 Concrete and Abstract Domains

B.1.1 Lemma 1 from Section 4.1

Given a type environment $\tau \in \mathcal{T}$ and an integer $d \in \mathbb{N}$, every ascending chain of elements in Alias_τ^d eventually stabilizes.

Proof. By Definition 18, $\bot_\tau^d$, i.e., the bottom element of Alias_τ^d is finite since it might contain only expressions whose depth is at most d, and variables in $\mathsf{dom}(\tau)$, field and method names are finite. Moreover, when $A^1 \sqsubseteq A^2$, it means that for each variable $v_r \in \mathsf{dom}(\tau)$, A_r^2 (the approximation A^2 assigns to v_r) is included in A_r^1 (the approximation A^1 assigns to v_r), i.e., $\mathsf{A}_r^2 \subseteq \mathsf{A}_r^1$. It means that greater elements in an ascending chain assign less alias expressions to each variable. It is worth noting that the least and the greatest elements an ascending chain might have are respectively $\bot_\tau^d$ (which is finite) and $\top_\tau^d = \varnothing^{|\tau|}$, which implies that this ascending chain eventually stabilizes, i.e., Alias_τ^d satisfies the ACC condition. □

Lemma 5 *For any type environment $\tau \in \mathcal{T}$, the function γ_τ is co-additive, i.e.,*

$$\gamma_\tau(\bigsqcap_{i\ge0} A^i) = \bigcap_{i\ge0}\gamma_\tau(A^i).$$

Proof. Consider a family of abstract elements $\{\langle\mathsf{A}_0^i,\ldots,\mathsf{A}_{n-1}^i\rangle\}_i$, where $n = |\tau|$. Then,

$$\begin{array}{rl} \gamma_\tau(\bigsqcap_i\langle\mathsf{A}_0^i,\ldots,\mathsf{A}_{n-1}^i\rangle) &= \gamma_\tau(\langle\bigcup_i \mathsf{A}_0^i,\ldots\bigcup_i \mathsf{A}_{n-1}^1\rangle) \\ &= \{\sigma\in\Sigma_\tau \mid \forall 0\le r<n.\forall\mathsf{E}\in\bigcup_i\mathsf{A}_r^i.[\![\mathsf{E}]\!]\sigma = \rho(v_r)\} \\ &= \{\sigma\in\Sigma_\tau \mid \forall 0\le r<n.\forall i.\forall\mathsf{E}\in\mathsf{A}_r^i.[\![\mathsf{E}]\!]\sigma = \rho(v_r)\} \\ &= \{\sigma\in\Sigma_\tau \mid \forall i.\forall 0\le r<n.\forall\mathsf{E}\in\mathsf{A}_r^i.[\![\mathsf{E}]\!]\sigma = \rho(v_r)\} \\ &= \bigcap_i\{\sigma\in\Sigma_\tau \mid \forall 0\le r<n.\forall\mathsf{E}\in\mathsf{A}_r^i.[\![\mathsf{E}]\!]\sigma = \rho(v_r)\} \\ &= \bigcap_i\gamma_\tau(\langle\mathsf{A}_0^i\ldots\mathsf{A}_{n-1}^i\rangle). \end{array}$$

□

B.1.2 Lemma 2 from Section 4.1

Given a type environment $\tau \in \mathcal{T}$, and the function $\gamma_\tau : \text{Alias}_\tau \to \mathsf{C}_\tau$, there exists a function $\alpha_\tau : \mathsf{C}_\tau \to \text{Alias}_\tau$ such that $\langle\mathsf{C}_\tau,\alpha_\tau,\gamma_\tau,\text{Alias}_\tau\rangle$ is a Galois connection.

Proof. Both C_τ and Alias_τ are complete lattices. Moreover, Lemma 5 shows that γ_τ is co-additive. Therefore, there exists the unique map α_τ, determined as:

$$\forall C\in\mathsf{C}_\tau.\alpha(C) = \bigcap\{A\in\text{Alias}_\tau | C\subseteq\gamma_\tau(A)\},$$

such that $\langle\mathsf{C}_\tau,\alpha_\tau,\gamma_\tau,\text{Alias}_\tau\rangle$ is a Galois connection [12]. Hence, Alias_τ is actually an abstract domain, in the sense of abstract interpretation. □

B.2 Solution of the Constraints

Lemma 6 $\langle \mathsf{EA}^d, \vec{\sqsubseteq}, \vec{\sqcup}, \vec{\sqcap}, \vec{\top}^d, \vec{\bot}^d \rangle$ *is a complete lattice.*

Proof. Follows directly from the fact that, for each n, $\text{Alias}^d_{\tau_n}$ is a complete lattice. □

Lemma 7 *Every ascending chain of elements in* EA^d *eventually stabilizes.*

Proof. By Lemma 1, given a type environment τ, every ascending chain of elements in Alias^d_τ eventually stabilizes. Moreover, every ascending chain of elements in EA^d represents a Cartesian product of x ascending chains of elements in $\text{Alias}^d_{\tau_1}, \dots, \text{Alias}^d_{\tau_x}$ which eventually stabilize, and x is a finite number. □

Lemma 8 F *is a monotonic function w.r.t.* $\vec{\sqsubseteq}$.

Proof. The propagation rules defined in Section 4.2 are monotonic w.r.t. $\sqsubseteq$, and this entails the monotonicity of the constraints defined by Equation 9. Consequently, F is a monotonic function w.r.t. $\vec{\sqsubseteq}$. □

B.2.1 Theorem 1 from Section 4.3

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

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

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

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

Moreover, the equation system $F(\vec{EA}) \vec{\sqsubseteq} \vec{EA}$ and the constraint system $\vec{EA} = F(\vec{EA})$ have the same least solution.

Proof. It is well-known that any monotonic function f over a partially ordered set satisfying the Ascending Chain Condition is also continuous [14]. By Lemma 8, F is monotonic, and by Lemma 7, EA^d satisfies the Ascending Chain Condition, hence F is continuous.

On the other hand, by Knaster-Tarski's fixpoint theorem [41], in a complete lattice $\langle L, \preccurlyeq, \curlyvee, \curlywedge, \top, \bot \rangle$, for any continuous function $f : L \to L$, the least fixpoint of f, *lfp(f)*, is equal to $\curlyvee_n f^n(\bot)$. Since $\langle \mathsf{EA}^d, \vec{\sqsubseteq}, \vec{\sqcup}, \vec{\sqcap}, \vec{\top}, \vec{\bot} \rangle$ is a complete lattice (Lemma 6) and F is continuous, its least fixpont can be computer as $\mathsf{lfp}(F) = \vec{\bigsqcup}_n F^n(\vec{\bot})$.

Suppose now that $\vec{EA}$ is a solution of the constraint system, i.e., $F(\vec{EA}) = \vec{EA}$. Then, starting from $\vec{\bot} \vec{\sqsubseteq} \vec{EA}$, by the monotonicity of F and mathematical induction, it can be shown that $F^n(\vec{\bot}) \vec{\sqsubseteq} \vec{EA}$. Since $F^n(\vec{\bot})$ is a solution of the constraint system, this shows that it is also the least solution of the constraint system. □

B.3 Soundness of our Approach

B.3.1 Soundness of Sequential Arcs

We show that in the case of the propagation rules of the sequential arcs, only non-exceptional concrete states belonging to the concretization of a correct approximation of the property of interest before a bytecode instruction is executed, are correctly propagated by the corresponding rule. That is because the sequential arcs simulate only those bytecode instructions that are defined on non-exceptional concrete states, and undefined on the exceptional ones. Lemma 9 shows that this property actually holds. As usual in abstract interpretation, this correctness result is proved by using the extension of the semantics *ins* of each bytecode to sets of states.

Lemma 9 *The propagation rules* Rule #1 - Rule #14 *introduced by Definition 21 are sound. More precisely, let us consider a sequential arc from a bytecode* ins *and its propagation rule Π. Assume that* ins *has static type information τ at its beginning and τ' immediately after its non-exceptional execution. Then, for every $A \in$ Alias$_\tau$ we have:*

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

(we recall that ins is the semantics of ins*, see Fig. 5).*

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contains i local variables $L = \{l_0, \dots, l_{i-1}\}$ and j stack elements $S = \{s_0, \dots, s_{j-1}\}$. For ease of representation, we let $\mathsf{dom}(\tau) = \{v_0, \dots, v_{n-1}\}$, where $n = |\tau|$, $v_r = l_r$ for $0 \le r < i$ and $v_r = s_{r-i}$ for $i \le r < n$, like we did in Definition 2. Moreover, let $\mathsf{dom}(\tau') = L' \cup S'$, where L' and S' are the local variables and stack elements of $\mathsf{dom}(\tau')$, and let $n' = |\tau'|$.

We choose an arbitrary abstract element $A = \langle \mathsf{A}_0, \dots, \mathsf{A}_{n-1} \rangle \in$ Alias$_\tau$, an arbitrary state $\sigma' = \langle \rho', \mu' \rangle \in ins(\gamma_\tau(A)) \cap \Xi_{\tau'}$, and we show that $\sigma' \in \gamma_{\tau'}(\Pi(A))$, i.e., (Definition 19) that

$$\text{for each } 0 \le r < n' \text{ and every } \mathsf{E} \in \mathsf{A}_r, [\![\mathsf{E}]\!]\sigma' = \rho'(v_r). \qquad (10)$$

Note that, by the choice of σ', there exists $\sigma = \langle \rho, \mu \rangle \in \gamma_\tau(A)$ such that $\sigma' = ins(\sigma)$ and such that, for each $0 \le r < n$ and every $\mathsf{E} \in \mathsf{A}_r$, $[\![\mathsf{E}]\!]\sigma = \rho(v_r)$.

$\underline{\mathsf{ins} = \mathsf{const}\ v}$. We have $L' = L$, $S' = S \cup \{s_j\}$. Moreover, for every $v \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\rho'(v) = \rho(v)$, while $\underline{\rho'(s_j) = x}$ and $\mu' = \mu$. According to Rule #1 of Definition 21, $\Pi(\langle \mathsf{A}_0, \dots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \dots, \mathsf{A}'_n \rangle$, where $\mathsf{A}'_r = \mathsf{A}_r$ for $r \neq n$ and $\mathsf{A}'_n = \{x\}$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

– if $0 \le r < n$, then $\mathsf{A}'_r = \mathsf{A}_r$, and therefore $\mathsf{E} \in \mathsf{A}_r$. By hypothesis,

$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

In this case, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for each $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. Finally, by Corollary 1, we have:

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$

– if $r = n$, then $\mathsf{A}'_n = \{x\}$ and $\mathsf{E} = x$. Therefore, $[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = x = \rho'(v_n)$.

$\underline{\mathsf{ins} = \mathsf{load}\ k\ \mathsf{t}}$. We have $L' = L$, $S' = S \cup \{s_j\}$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_j) = \rho(l_k)$. According to Rule #2 of Definition 21, $\Pi(\langle \mathsf{A}_0, \dots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \dots, \mathsf{A}'_n \rangle$, where $\mathsf{A}'_r = \mathsf{A}_r \cup \mathsf{A}_r[s_j/l_k]$ for $r \notin \{k, n\}$, $\mathsf{A}'_k = \mathsf{A}_k \cup \{s_j\}$ and $\mathsf{A}_n = \mathsf{A}_k \cup \{l_k\}$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

– if $r \notin \{k, n\}$, then $\mathsf{A}'_r = \mathsf{A}_r \cup \mathsf{A}_r[s_j/l_k]$. If $\mathsf{E} \in \mathsf{A}_r$, then, by hypothesis,

$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

In this case, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 1, we have:

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$

Otherwise, if $\mathsf{E} \in \mathsf{A}_r[s_j/l_k]$, then there exists $\mathsf{E}_1 \in \mathsf{A}_r$ such that $\mathsf{E} = \mathsf{E}_1[s_j/l_k]$. Note that, by hypothesis,

$$[\![\mathsf{E}_1]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

Since $\rho'(s_j) = \rho'(l_k)$, we have, by Lemma 4:

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}_1[s_j/l_k]]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}_1]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}_1]\!]\langle \rho', \mu \rangle.$$

On the other hand, $\mathsf{E}_1 \in \mathsf{A}_r$ entails $s_j \notin \mathsf{variables}(\mathsf{E}_1)$, and therefore for every $v \in \mathsf{variables}(\mathsf{E}_1)$, $\rho'(v) = \rho(v)$. By Corollary 1 we have:

$$[\![\mathsf{E}_1]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}_1]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$

Thus, $[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}_1]\!]\langle \rho', \mu \rangle = \rho'(v_r)$.

– if $r = k$, then $v_k = l_k$ and $\mathsf{A}'_k = \mathsf{A}_k \cup \{s_j\}$. If $\mathsf{E} \in \mathsf{A}_k$, then by hypothesis,

$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_k) = \rho(l_k) = \rho'(s_j) = \rho'(v_k).$$

Moreover, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 1, we have

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_k).$$

Otherwise, $\mathsf{E} = s_j$, and we have

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = \rho'(s_j) = \rho'(l_k) = \rho'(v_k).$$

– if $r = n$, then $v_n = s_j$ and $\mathsf{A}'_n = \mathsf{A}_k \cup \{l_k\}$. If $\mathsf{E} \in \mathsf{A}_k$, then by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_k) = \rho'(v_n).$$

Moreover,$s_j \notin$ variables(E), and therefore, for every $v \in$ variables(E), $\rho'(v) = \rho(v)$. By Corollary 1, we have:

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_n).$$

Otherwise, $\mathsf{E} = l_k$, and we have

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = \rho'(l_k) = \rho'(s_j) = \rho'(v_n).$$

ins = store k t. We have $L' = L$, $S' = S \smallsetminus \{s_{j-1}\}$, $\mu' = \mu$, and for every $v \in \mathsf{dom}(\tau') \smallsetminus \{l_k\}$, $\rho'(v) = \rho(v)$, while $\rho'(l_k) = \rho(s_{j-1})$. According to Rule #5 of Definition 21, $\Pi(\langle \mathsf{A}_0, \dots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \dots, \mathsf{A}'_{n-2}\rangle$, where $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})\}$ for $r \neq k$ and $\mathsf{A}'_k = \{\mathsf{E} \in \mathsf{A}_k \mid \neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})\}$. According to Definition 16, for every $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})$ holds if $l_k, s_{j-1} \notin$ variables(E). Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

– if $r \neq k$, then $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid l_k, s_{j-1} \notin \mathsf{variables}(\mathsf{E})\} \subseteq \mathsf{A}_r$. Since $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ we have, by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, $l_k, s_{j-1} \notin$ variables(E), and therefore, for every $v \in$ variables(E), $\rho'(v) = \rho(v)$. By Corollary 1, we have:

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

– if $r = k$, then $\mathsf{A}'_k = \{\mathsf{E} \in \mathsf{A}_{n-1} \mid l_k, s_{j-1} \notin \mathsf{variables}(\mathsf{E})\} \subseteq \mathsf{A}_{n-1}$. Since $\mathsf{E} \in \mathsf{A}'_{n-1} \subseteq \mathsf{A}_{n-1}$ we have, by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_{n-1}) = \rho(s_{j-1}) = \rho'(l_k) = \rho'(v_k).$$

Moreover, $l_k, s_{j-1} \notin$ variables(E), and therefore, for every $v \in$ variables(E), $\rho'(v) = \rho(v)$. By Corollary 1, we have:

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_k).$$

ins $\in$ {add, sub, mul, div, rem}. We have $L' = L$, $S' = S \smallsetminus \{s_{j-1}\}$, $\mu' = \mu$, and for every $v \in \mathsf{dom}(\tau') \smallsetminus \{s_{j-2}\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_{j-2}) = \rho(s_{j-2}) \oplus \rho(s_{j-1})$, where $\oplus$ is the arithmetic operation corresponding to ins. According to Rule #4 of Definition 21, $\Pi(\langle \mathsf{A}_0, \dots, \mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0, \dots, \mathsf{A}'_{n-2}\rangle$, where

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})\} & \text{if } r \neq |\tau|-2 \\ \{\mathsf{E}_1 \oplus \mathsf{E}_2 \mid \mathsf{E}_1 \in \mathsf{A}_{|\tau|-2} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_1,\mathsf{ins}) \wedge & \\ \qquad \mathsf{E}_2 \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_2,\mathsf{ins}) & \text{if } r = |\tau|-2, \end{cases}$$

According to Definition 16, for every expresion $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})$ holds if $s_{j-2}, s_{j-1} \notin$ variables(E). Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

– if $r < n-2$, then $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})\} \subseteq \mathsf{A}_r$. Since $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ we have, by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, $s_{j-2}, s_{j-1} \notin$ variables(E), and therefore, for every $v \in$ variables(E), $\rho'(v) = \rho(v)$. By Corollary 1, we have:

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

– if $r = n-2$, then $\mathsf{E} = \mathsf{E}_1 \oplus \mathsf{E}_2$, where $\mathsf{E}_1 \in \mathsf{A}_{n-2}$, $\mathsf{E}_2 \in \mathsf{A}_{n-1}$, $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E}_1)$ and $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E}_2)\}$. Since $\mathsf{E}_1 \in \mathsf{A}_{n-2}$ and $\mathsf{E}_2 \in \mathsf{A}_{n-1}$ we have, by hypothesis,

$$\begin{aligned} [\![\mathsf{E}_1]\!]\langle\rho,\mu\rangle &= \rho(v_{n-2}) = \rho(s_{j-2}) \\ [\![\mathsf{E}_2]\!]\langle\rho,\mu\rangle &= \rho(v_{n-1}) = \rho(s_{j-1}). \end{aligned}$$

Moreover, for $h \in \{1,2\}$, $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E}_h)$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E}_h)$, $\rho'(v) = \rho(v)$. Hence, by Corollary 1,

$$[\![\mathsf{E}_1]\!]^*\langle\rho',\mu\rangle = [\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle = \langle\rho(s_{j-2}),\mu_1\rangle$$

and

$$[\![\mathsf{E}_2]\!]^*\langle\rho',\mu_1\rangle = [\![\mathsf{E}_2]\!]^*\langle\rho,\mu_1\rangle = \langle\rho(s_{j-1}),\mu_2\rangle.$$

Hence,

$$[\![\mathsf{E}]\!]^*\langle\rho',\mu'\rangle = [\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\langle\rho',\mu'\rangle = [\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]^*\langle\rho',\mu\rangle = \langle\rho(s_{j-2}) \oplus \rho(s_{j-1}),\mu_2\rangle,$$

i.e.,

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = \rho(s_{j-2}) \oplus \rho(s_{j-1}) = \rho'(s_{j-2}) = \rho'(v_{n-2}).$$

<u>$\mathsf{ins} = \mathsf{inc}\ k\ x$</u>. We have $L' = L$, $S' = S$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau') \smallsetminus \{l_k\}$, $\rho'(v) = \rho(v)$, while $\rho'(l_k) = \rho(l_k) + x$. According to Rule #5 of Definition 21, $\Pi(\langle \mathsf{A}_0, \dots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \dots, \mathsf{A}'_{n-1} \rangle$, where $\mathsf{A}'_r = \{\mathsf{E}[l_k - x/l_k] \mid \mathsf{E} \in \mathsf{A}_r\}$ for $r \neq k$, and $\mathsf{A}'_k = \varnothing$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

– if $r \neq k$, then $\mathsf{A}'_r = \{\mathsf{E}[l_k - x/l_k] \mid \mathsf{E} \in \mathsf{A}_r\}$. If $l_k \notin \mathsf{variables}(\mathsf{E})$, then, $\mathsf{E}[l_k - x/l_k] = \mathsf{E} \in \mathsf{A}_r$ and, by hypothesis,
$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$
Moreover, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$ and, by Corollary 1, we have:
$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$
Otherwise, $l_k \in \mathsf{variables}(\mathsf{E})$, and we have that for any $v \in \mathsf{variables}(\mathsf{E}) \smallsetminus \{l_k\}$, $\rho'(v) = \rho(v)$. Hence, by Lemma 3,
$$[\![\mathsf{E}[l_k - x/l_k]]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle.$$
By hypothesis,
$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r),$$
and therefore
$$[\![\mathsf{E}[l_k - x/l_k]]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}[l_k - x/l_k]]\!]\langle \rho', \mu \rangle = \rho'(v_r).$$

– if $r = k$, then $\mathsf{A}'_k = \{\mathsf{E}{+}x \mid \mathsf{E} \in \mathsf{A}_k\}$. Consider an arbitrary expression $\mathsf{E}_1 \in \mathsf{A}'_k$. It has the following form: $\mathsf{E}_1 = \mathsf{E} + x$, where $\mathsf{E} \in \mathsf{A}_k$. The latter implies that $l_k \notin \mathsf{variables}(\mathsf{E})$, hence for any $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$ and, by Corollary 1, we have:
$$[\![\mathsf{E}]\!]^*\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]^*\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]^*\langle \rho, \mu \rangle = \langle \rho(v_k), \mu'' \rangle = \langle \rho(l_k), \mu'' \rangle,$$
where μ'' is the memory μ potentially updated by E's evaluation. Moreover, by Definition 14 we have $[\![x]\!]^*\langle \rho', \mu'' \rangle = \langle x, \mu'' \rangle$ and therefore
$$[\![\mathsf{E}_1]\!]^*\langle \rho', \mu' \rangle = [\![\mathsf{E} + x]\!]^*\langle \rho', \mu' \rangle = \langle \rho(l_k) + x, \mu'' \rangle.$$
Thus, we obtain $[\![\mathsf{E}_1]\!]\langle \rho', \mu' \rangle = \rho(l_k) + x = \rho'(l_k)$.

<u>$\mathsf{ins} = \mathsf{new}\ \kappa$</u>. We have $L' = L$, $S' = S \cup \{s_j\}$. For every $v \in \mathsf{dom}(\tau') \smallsetminus \{s_j\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_j) = \ell \in \mathbb{L}$, where ℓ is a fresh location, i.e., a location not reachable from any other location and which does not reach any other location. Moreover, $\mu' = \mu[\ell \mapsto o]$, where o is a new object of class κ. It is worth noting that, under these circumstances,
$$\forall \mathsf{E} \in \mathbb{E}_{\tau'}. [\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle. \tag{11}$$
According to Rule #6 of Definition 21, $\Pi(\langle \mathsf{A}_0, \dots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \dots, \mathsf{A}'_n \rangle$, where $\mathsf{A}'_r = \mathsf{A}_r$ for $r \neq n$, while $\mathsf{A}'_n = \varnothing$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

– if $0 \le r < n$, then $\mathsf{A}'_r = \mathsf{A}_r$, and therefore $\mathsf{E} \in \mathsf{A}_r$. By hypothesis,
$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$
In this case, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 1, we have
$$[\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$
Hence, the latter and (11) entail
$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = \rho'(v_r).$$

– if $r = n$, then $\mathsf{A}'_n = \varnothing$, and therefore $\forall \mathsf{E} \in \mathsf{A}'_n. [\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = \rho'(v_n)$ trivially holds.

<u>$\mathsf{ins} = \mathsf{getfield}\ \kappa.f{:}\mathsf{t}$</u>. We have $L' = L$, $S' = S$, $\mu' = \mu$, and for every $v \in \mathsf{dom}(\tau') \smallsetminus \{s_{j-1}\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_{j-1}) = (\mu\rho(s_{j-1}).\phi)(f)$. According to Rule #7 of Definition 21, $\Pi(\langle \mathsf{A}_0, \dots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \dots, \mathsf{A}'_{n-1} \rangle$, where for any $r \neq n - 1$,
$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\},$$
while
$$\mathsf{A}'_{n-1} = \{\mathsf{E}.f \mid \mathsf{E} \in \mathsf{A}_{n-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}) \wedge \neg\mathsf{mightModify}(\mathsf{E}, \{\kappa.f{:}\mathsf{t}\})\}.$$
According to Definition 16, for any $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ holds if $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $r \neq n-1$, then $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ and we have, by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

  Moreover, $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 1, we have

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

- if $r = n-1$, then since $\mathsf{E} \in \mathsf{A}'_{n-1}$, there exists $\mathsf{E}_1 \in \mathsf{A}_{n-1}$ such that $\mathsf{E} = \mathsf{E}_1.f$. By hypothesis,

$$[\![\mathsf{E}_1]\!]\langle\rho,\mu\rangle = \rho(v_{n-1}) = \rho(s_{j-1}).$$

  In addition, $s_{j-1} \notin \mathsf{variables}(\mathsf{E}_1)$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E}_1)$, $\rho'(v) = \rho(v)$. Hence, by Corollary 1, we have:

$$[\![\mathsf{E}_1]\!]^*\langle\rho',\mu\rangle = [\![\mathsf{E}_1]\!]^*\langle\rho,\mu\rangle = \langle\rho(s_{j-1}),\mu_1\rangle.$$

  Hence,

$$[\![\mathsf{E}_1]\!]^*\langle\rho',\mu'\rangle = [\![\mathsf{E}_1]\!]^*\langle\rho',\mu\rangle = \langle\rho(s_{j-1}),\mu_1\rangle.$$

  Moreover, $\neg\mathsf{mightModify}(\mathsf{E}_1,\{f\})$ guarantees that no evaluation of E_1 might modify the field f. In particular, evaluation of E_1 in $\langle\rho,\mu\rangle$ does not modify f, and therefore its value before E_1's evaluation, $(\mu\rho(s_{j-1}).\phi)(f)$, is equal to its value after E_1's evaluation, $(\mu_1\rho(s_{j-1}).\phi)(f)$. Hence,

$$\begin{aligned}[\![\mathsf{E}]\!]^*\langle\rho',\mu'\rangle &= [\![\mathsf{E}_1.f]\!]^*\langle\rho',\mu'\rangle\\ &= \langle(\mu_1\rho(s_{j-1}).\phi)(f),\mu_1\rangle\\ &= \langle(\mu\rho(s_{j-1}).\phi)(f),\mu_1\rangle\\ &= \langle\rho'(s_{j-1}),\mu_1\rangle,\end{aligned}$$

  which implies $[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = \rho'(s_{j-1}) = \rho'(v_r)$.

<u>ins = putfield $\kappa.f$:t</u>. We have $L' = L$, $S' = S \setminus \{s_{j-2}, s_{j-1}\}$, $\mu' = \mu[(\mu\rho(s_{j-2}).\phi)(f) \mapsto \rho(s_{j-1})]$ and $\rho' = \rho$. According to RULE #8 of Definition 21, $\Pi(\langle\mathsf{A}_0,\ldots,\mathsf{A}_{n-1}\rangle) = \langle\mathsf{A}'_0,\ldots,\mathsf{A}'_{n-3}\rangle$, where for each $0 \le r < n-2$,

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})\}.$$

According to Definition 16, for every expresion $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})$ holds if $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})$ and if no evaluation of E might read a field f. Consider an expression $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$, for an arbitrary $0 \le r < n-2$. By hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r).$$

Moreover, $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})$, hence for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho(v) = \rho'(v)$ and, by Corollary 1:

$$[\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

Since no evaluation of E, and in particular its evaluation in $\langle\rho',\mu\rangle$, might read any field f, E's value $[\![\mathsf{E}]\!]\langle\rho',\mu\rangle$ does not depend on a value of any field f in that state. In particular, $[\![\mathsf{E}]\!]\langle\rho',\mu\rangle$ does not depend on the value of the field f of the object memorized in location $\rho(s_{j-1})$, $(\mu\rho(s_{j-1}).\phi)(f)$. Since $\mu' = \mu[(\mu\rho(s_{j-2}).\phi)(f) \mapsto \rho(s_{j-1})]$, i.e., the only difference between memories μ and μ' is exactly the value of the field mentioned above, we conclude that E's values in $\langle\rho',\mu'\rangle$ and in $\langle\rho',\mu\rangle$ are equal, i.e.,

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = \rho'(v_r).$$

<u>ins = arraynew α</u>. We have $L' = L$, $S' = S$. For every $v \in \mathsf{dom}(\tau') \setminus \{s_{j-1}\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_{j-1}) = \ell \in \mathbb{L}$, where ℓ is a fresh location, i.e., a location not reachable from any other location and which does not reach any other location. Moreover, $\mu' = \mu[\ell \mapsto a]$, where o is a new aray of array type α. It is worth noting that, under these circumstances,

$$\forall\mathsf{E} \in \mathbb{E}_{\tau'}.[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle. \tag{12}$$

According to RULE #9 of Definition 21, $\Pi(\langle\mathsf{A}_0,\ldots,\mathsf{A}_{n-1}\rangle) = \langle\mathsf{A}'_0,\ldots,\mathsf{A}'_n\rangle$, where for each $r \neq n-1$, $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})\}$, while $\mathsf{A}'_{n-1} = \varnothing$. According to Definition 16, for every expresion $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})$ holds if $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

– if $0 \le r < n$, then $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid s_{j-1} \notin \mathsf{variables}(\mathsf{E})\} \subseteq \mathsf{A}_r$. Since $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ we have, by hypothesis,

$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By (12 and Corollary 1, we have:

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$

– if $r = n-1$, then $\mathsf{A}'_n = \varnothing$, and therefore $\forall \mathsf{E} \in \mathsf{A}'_n . [\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = \rho'(v_n)$ trivially holds.

<u>ins = arraylength α</u>. We have $L' = L$, $S' = S$, $\mu' = \mu$, and for every $v \in \mathsf{dom}(\tau') \smallsetminus \{s_{j-1}\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_{j-1}) = \mu\rho(s_{j-1}).\mathtt{length}$. According to Rule #10 of Definition 21, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{n-1} \rangle$, where for any $r \neq n-1$,

$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\},$$

while

$$\mathsf{A}'_{n-1} = \{\mathsf{E}.\mathtt{length} \mid \mathsf{E} \in \mathsf{A}_{n-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\}.$$

According to Definition 16, for any $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ holds if $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

– if $r \neq n-1$, then $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ and we have, by hypothesis,

$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, $s_{j-1} \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 1, we have

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$

– if $r = n-1$, then since $\mathsf{E} \in \mathsf{A}'_{n-1}$, there exists $\mathsf{E}_1 \in \mathsf{A}_{n-1}$ such that $\mathsf{E} = \mathsf{E}_1.\mathtt{length}$. By hypothesis,

$$[\![\mathsf{E}_1]\!]\langle \rho, \mu \rangle = \rho(v_{n-1}) = \rho(s_{j-1}).$$

In addition, $s_{j-1} \notin \mathsf{variables}(\mathsf{E}_1)$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E}_1)$, $\rho'(v) = \rho(v)$. Hence, by Corollary 1, we have:

$$[\![\mathsf{E}_1]\!]^*\langle \rho', \mu' \rangle = [\![\mathsf{E}_1]\!]^*\langle \rho', \mu \rangle = [\![\mathsf{E}_1]\!]^*\langle \rho, \mu \rangle = \langle \rho(s_{j-1}), \mu_1 \rangle.$$

Moreover, no evaluation of any expression might modify the length of already existing array, i.e., the length of the array $\mu\rho(s_{j-1})$ is equal before and after E_1's evaluation in $\langle \rho, \mu \rangle$: $\mu\rho(s_{j-1}).\mathtt{length} = \mu_1\rho(s_{j-1}).\mathtt{length}$. Hence,

$$\begin{aligned} [\![\mathsf{E}]\!]^*\langle \rho', \mu' \rangle &= [\![\mathsf{E}_1.\mathtt{length}]\!]^*\langle \rho', \mu' \rangle \\ &= \langle \mu_1\rho(s_{j-1}).\mathtt{length}, \mu_1 \rangle \\ &= \langle \mu\rho(s_{j-1}).\mathtt{length}, \mu_1 \rangle \\ &= \langle \rho'(s_{j-1}), \mu_1 \rangle, \end{aligned}$$

which implies $[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = \rho'(s_{j-1}) = \rho'(v_r)$.

<u>ins = arrayload α</u>. We have $L' = L$, $S' = S \smallsetminus \{s_{j-1}\}$, $\mu' = \mu$, and for every $v \in \mathsf{dom}(\tau') \smallsetminus \{s_{j-2}\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_{j-2}) = (\mu\rho(s_{j-2}).\phi)(\rho(s_{j-1}))$. According to Rule #11 of Definition 21, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{n-2} \rangle$, where

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})\} & \text{if } r \neq |\tau|-2 \\ \{\mathsf{E}_1[\mathsf{E}_2] \mid \mathsf{E}_1 \in \mathsf{A}_{|\tau|-2} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_1, \mathsf{ins}) \wedge & \\ \quad \mathsf{E}_2 \in \mathsf{A}_{|\tau|-1} \wedge \neg\mathsf{canBeAffected}(\mathsf{E}_2, \mathsf{ins}) \wedge & \\ \quad [\mathsf{E}_1 \text{ and } \mathsf{E}_2 \text{ do not invoke any method}] & \text{if } r = |\tau|-2. \end{cases}$$

According to Definition 16, for every expresion $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins})$ holds if $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

– if $r < n-2$, then $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})\} \subseteq \mathsf{A}_r$. Since $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ we have, by hypothesis,

$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 1, we have:

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$

- if $r = n-2$, then $\mathsf{E} = \mathsf{E}_1[\mathsf{E}_2]$, where $\mathsf{E}_1 \in \mathsf{A}_{n-2}$, $\mathsf{E}_2 \in \mathsf{A}_{n-1}$, $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E}_1)$, $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E}_2)\}$ and E_1 and E_2 do not invoke any method. Since $\mathsf{E}_1 \in \mathsf{A}_{n-2}$ and $\mathsf{E}_2 \in \mathsf{A}_{n-1}$ we have, by hypothesis,
$$\begin{aligned}\llbracket \mathsf{E}_1 \rrbracket\langle\rho,\mu\rangle = \rho(v_{n-2}) = \rho(s_{j-2})\\ \llbracket \mathsf{E}_2 \rrbracket\langle\rho,\mu\rangle = \rho(v_{n-1}) = \rho(s_{j-1}).\end{aligned}$$
Since $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E}_1)$, for every $v \in \mathsf{variables}(\mathsf{E}_1)$, $\rho'(v) = \rho(v)$, and, by Corollary 1, we have:
$$\llbracket \mathsf{E}_1 \rrbracket^*\langle\rho',\mu'\rangle = \llbracket \mathsf{E}_1 \rrbracket^*\langle\rho',\mu\rangle = \llbracket \mathsf{E}_1 \rrbracket^*\langle\rho,\mu\rangle = \langle\rho(s_{j-2}),\mu\rangle. \tag{13}$$
Similarly, since $s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E}_2)$, for every $v \in \mathsf{variables}(\mathsf{E}_2)$, $\rho'(v) = \rho(v)$, and, by Corollary 1, we have:
$$\llbracket \mathsf{E}_2 \rrbracket^*\langle\rho',\mu'\rangle = \llbracket \mathsf{E}_2 \rrbracket^*\langle\rho',\mu\rangle = \llbracket \mathsf{E}_2 \rrbracket^*\langle\rho,\mu\rangle = \langle\rho(s_{j-1}),\mu\rangle. \tag{14}$$
It is worth noting that since both E_1 and E_2 do not invoke any method, their evaluation in any state $\langle\rho_1,\mu_1\rangle$ do not modify the memory μ_1, hence the resulting memory when E_1 and E_2 are evaluated in $\langle\rho,\mu\rangle$ is μ (Equations 13 and 14). Hence, Equations 13 and 14 entail
$$\llbracket \mathsf{E} \rrbracket^*\langle\rho',\mu'\rangle = \llbracket \mathsf{E}_1[\mathsf{E}_2] \rrbracket^*\langle\rho',\mu'\rangle = \langle(\mu\rho(s_{j-2}).\phi)(\rho(s_{j-1})),\mu\rangle = \langle\rho'(s_{j-2}),\mu\rangle,$$
i.e.,
$$\llbracket \mathsf{E} \rrbracket\langle\rho',\mu'\rangle = \rho'(s_{j-2}) = \rho'(v_{n-2}).$$

<u>ins = arraystore t[ ]</u>. We have $L' = L$, $S' = S \setminus \{s_{j-3}, s_{j-2}, s_{j-1}\}$, $\rho' = \rho$ and
$$\mu' = \mu[(\mu\rho(s_{j-3}).\phi)(\rho(s_{j-2})) \mapsto \rho(s_{j-1})].$$
According to Rule #12 of Definition 21, $\Pi(\langle \mathsf{A}_0,\dots,\mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0,\dots,\mathsf{A}'_{n-4}\rangle$, where for each $0 \le r < n-3$,
$$\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})\}.$$
According to Definition 16, for every expresion $\mathsf{E} \in \mathbb{E}_\tau$, $\neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})$ holds if $s_{j-3}, s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})$ and if there is no evaluation of E which might read an element of an array of type t′[ ] where t′ ∈ compatible(t). Consider an expression $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$, for an arbitrary $0 \le r < n-3$. By hypothesis,
$$\llbracket \mathsf{E} \rrbracket\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$
Moreover, $s_{j-3}, s_{j-2}, s_{j-1} \notin \mathsf{variables}(\mathsf{E})$, hence for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho(v) = \rho'(v)$ and, by Corollary 1:
$$\llbracket \mathsf{E} \rrbracket\langle\rho',\mu\rangle = \llbracket \mathsf{E} \rrbracket\langle\rho,\mu\rangle = \rho'(v_r). \tag{15}$$
No evaluation of E might read an element of an array of type t′[ ], where t′ ∈ compatible(t). Therefore, E's value in any state definitely does not depend on any array element whose type is compatible with t and, consequentially, arraystore t[ ] never affects its value. Hence,
$$\llbracket \mathsf{E} \rrbracket\langle\rho',\mu'\rangle = \llbracket \mathsf{E} \rrbracket\langle\rho',\mu\rangle. \tag{16}$$
Hence, Equations 15 and 16 entail:
$$\llbracket \mathsf{E} \rrbracket\langle\rho',\mu'\rangle = \llbracket \mathsf{E} \rrbracket\langle\rho',\mu\rangle = \rho'(v_r).$$

<u>ins = dup t</u>. We have $L' = L$, $S' = S \cup \{s_j\}$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau') \setminus \{s_j\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_j) = \rho(s_{j-1})$. According to Rule #13 of Definition 21, $\Pi(\langle \mathsf{A}_0,\dots,\mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0,\dots,\mathsf{A}'_n\rangle$, where $\mathsf{A}'_r = \mathsf{A}_r \cup \mathsf{A}_r[s_j/s_{j-1}]$ for $r < n-1$, $\mathsf{A}'_{n-1} = \mathsf{A}_{n-1} \cup \{s_j\}$ and $\mathsf{A}_n = \mathsf{A}_{n-1} \cup \{s_{j-1}\}$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

- if $0 \le r < n-1$, then $\mathsf{A}'_r = \mathsf{A}_r \cup \mathsf{A}_r[s_j/s_{j-1}]$. If $\mathsf{E} \in \mathsf{A}_r$, then, by hypothesis,
$$\llbracket \mathsf{E} \rrbracket\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$
In this case, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 1, we have:
$$\llbracket \mathsf{E} \rrbracket\langle\rho',\mu'\rangle = \llbracket \mathsf{E} \rrbracket\langle\rho',\mu\rangle = \llbracket \mathsf{E} \rrbracket\langle\rho,\mu\rangle = \rho'(v_r).$$
Otherwise, if $\mathsf{E} \in \mathsf{A}_r[s_j/s_{j-1}]$, then there exists $\mathsf{E}_1 \in \mathsf{A}_r$ such that $\mathsf{E} = \mathsf{E}_1[s_j/s_{j-1}]$. Note that, by hypothesis,
$$\llbracket \mathsf{E}_1 \rrbracket\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

Since $\rho'(s_j) = \rho'(s_{j-1})$, we have, by Lemma 4,

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}_1[s_j/s_{j-1}]]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}_1]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}_1]\!]\langle\rho',\mu\rangle.$$

On the other hand, $\mathsf{E}_1 \in \mathsf{A}_r$ entails $s_j \notin \mathsf{variables}(\mathsf{E}_1)$, and therefore for every $v \in \mathsf{variables}(\mathsf{E}_1)$, $\rho'(v) = \rho(v)$. By Corollary 1 we have

$$[\![\mathsf{E}_1]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}_1]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

Thus, $[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}_1]\!]\langle\rho',\mu\rangle = \rho'(v_r)$.

– if $r = n-1$, then $v_r = s_{j-1}$ and $\mathsf{A}'_r = \mathsf{A}_{n-1} \cup \{s_j\}$. If $\mathsf{E} \in \mathsf{A}_{n-1}$, then by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, $s_j \notin \mathsf{variables}(\mathsf{E})$, and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 1, we have:

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

Otherwise, $\mathsf{E} = s_j$, and we have $[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = \rho'(s_j) = \rho'(s_{j-1}) = \rho'(v_r)$.

– if $r = n$, then $v_r = s_j$ and $\mathsf{A}'_r = \mathsf{A}_{n-1} \cup \{s_{j-1}\}$. If $\mathsf{E} \in \mathsf{A}_{n-1}$, then by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_{n-1}) = \rho'(v_n).$$

Moreover,$s_j \notin$variables(E), and therefore, for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. By Corollary 1, we have

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_n).$$

Otherwise, $\mathsf{E} = s_{j-1}$, and $[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = \rho'(s_{j-1}) = \rho'(s_j) = \rho'(v_n)$.

<u>ins ∈ {ifne t, ifeq t, catch, exception_is K}</u>. We have $L' = L$, $S' = S$ when ins ∈ {catch, exception_is K}, and $S' = S \setminus \{s_{j-2}, s_{j-1}\}$ otherwise. Moreover, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau')$, $\rho'(v) = \rho(v)$. According to Rule #14 of Definition 21, $\Pi(\langle \mathsf{A}_0,\ldots,\mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0,\ldots,\mathsf{A}'_{n'-1}\rangle$, where $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \neg\mathsf{canBeAffected}(\mathsf{E},\mathsf{ins})\}$ for each $0 \le r < n'$. According to Definition 16, for any $\mathsf{E} \in \mathbb{E}_\tau$, ¬canBeAffected(E, ins) always holds when ins ∈ {catch, exception_is K}, while ¬canBeAffected(E, ins) holds if $s_{j-2}, s_{j-1} \notin$ variables(E) when ins ∈ {ifne t, ifeq t}. Consider an expression $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ for an arbitrary $0 \le r < n'$. By hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, ¬canBeAffected(E, ins) entails variables(E) ⊆ dom(τ'), and therefore for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. Then, by Corollary 1,

$$[\![\mathsf{E}]\!]\langle\rho',\mu'\rangle = [\![\mathsf{E}]\!]\langle\rho',\mu\rangle = [\![\mathsf{E}]\!]\langle\rho,\mu\rangle = \rho'(v_r).$$

□

B.3.2 Soundness of Final Arcs

We show that, in the case of the propagation rules of the final arcs, both exceptional and non-exceptional concrete states belonging to the concretization of a correct approximation of the property of interest before a bytecode instruction is executed, are correctly propagated by the corresponding rule. It means that the propagation rules of the final arcs must soundly approximate the concrete behavior of each final bytecode instruction (return t, return void, throw κ) of a method or a constructor belonging to the program under analysis. Lemma 10 shows that this property actually holds.

Lemma 10 *The propagation rules* Rule #15 - Rule #17 *introduced by Definition 22 are sound. More precisely, let us consider a final arc from a bytecode* ins *and its propagation rule Π. Assume that* ins *has static type information τ at its beginning and τ' immediately after its execution (its non-exceptional execution if* ins *is a* return*, its exceptional execution if* ins *is a* throw κ*). Then, for every $A \in \mathrm{Alias}_\tau$ we have:*

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

(we recall that ins is the semantics of ins*, see Fig. 5).*

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contains i local variables $L = \{l_0, \ldots, l_{i-1}\}$ and j stack elements $S = \{s_0, \ldots, s_{j-1}\}$. For ease of representation, we let $\mathsf{dom}(\tau) = \{v_0, \ldots, v_{n-1}\}$, where $n = |\tau|$, $v_r = l_r$ for $0 \le r < i$ and $v_r = s_{r-i}$ for $i \le r < n$, like we did in Definition 2. Moreover, let $\mathsf{dom}(\tau') = L' \cup S'$, where L' and S' are the local and stack variables of $\mathsf{dom}(\tau')$, and let $n' = |\tau'|$.

We choose an arbitrary abstract element $A = \langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle \in \textsc{Alias}_\tau$, an arbitrary state $\sigma' = \langle \rho', \mu' \rangle \in ins(\gamma_\tau(A))$, and we show that $\sigma' \in \gamma_{\tau'}(\Pi(A))$, i.e., (Definition 19) that

$$\text{for each } 0 \le r < n' \text{ and every } \mathsf{E} \in \mathsf{A}_r, [\![\mathsf{E}]\!]\sigma' = [\![v_r]\!]\sigma'.$$

Note that, by the choice of σ', there exists $\sigma = \langle \rho, \mu \rangle \in \gamma_\tau(A)$ such that $\sigma' = ins(\sigma)$ and such that, for each $0 \le r < n$ and every $\mathsf{E} \in \mathsf{A}_r$, $[\![\mathsf{E}]\!]\sigma = [\![v_r]\!]\sigma = \rho(v_r)$.

<u>ins = return void</u>. We have $L' = L$, $S' = \emptyset$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau')$, $\rho'(v) = \rho(v)$. According to Rule #15 of Definition 22, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_{i-1} \rangle$, where for each $0 \le r < i$, $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\}$. Consider an expression $\mathsf{E} \in \mathsf{A}'_r \subseteq \mathsf{A}_r$ for an arbitrary $0 \le r < i$. By hypothesis,

$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, $\mathsf{noStackElements}(\mathsf{E})$ entails $\mathsf{variables}(\mathsf{E}) \subseteq \mathsf{dom}(\tau')$, and therefore for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. Therefore, by Corollary 1, we have:

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho'(v_r).$$

<u>ins = return t</u>. We have $L' = L$, $S' = \{s_0\}$, $\mu' = \mu$ and for every $v \in \mathsf{dom}(\tau') \smallsetminus \{s_0\}$, $\rho'(v) = \rho(v)$, while $\rho'(s_0) = \rho(s_{j-1})$. According to Rule #16 of Definition 22, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_i \rangle$, where $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\}$ for $r \ne i$ and $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_{n-1} \mid \mathsf{noStackElements}(\mathsf{E})\}$ for $r = i$. Note that for any $\mathsf{E} \in \mathbb{E}_\tau$, $\mathsf{noStackElements}(\mathsf{E})$ entails $\mathsf{variables}(\mathsf{E}) \subseteq \mathsf{dom}(\tau')$, thus for every $v \in \mathsf{variables}(\mathsf{E})$, $\rho'(v) = \rho(v)$. Then, by Corollary 1,

$$[\![\mathsf{E}]\!]\langle \rho', \mu \rangle = [\![\mathsf{E}]\!]\langle \rho, \mu \rangle. \tag{17}$$

Consider an expression $\mathsf{E} \in \mathsf{A}'_r$. We distinguish the following cases:

– If $r \ne i$ then $\mathsf{E} \in \mathsf{A}_r$ and, by hypothesis,

$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_r) = \rho'(v_r).$$

By (17), we have

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = \rho'(v_r).$$

– If $r = i$ then $\mathsf{E} \in \mathsf{A}_{n-1}$ and, by hypothesis,

$$[\![\mathsf{E}]\!]\langle \rho, \mu \rangle = \rho(v_{n-1}) = \rho(s_{j-1}) = \rho'(s_0) = \rho'(v_i).$$

By (17), we have

$$[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle = \rho'(v_i).$$

<u>ins = throw κ</u>. We have $L' = L$, $S' = \{s_0\}$ and for every $v \in L'$, $\rho'(v) = \rho(v)$. If $\rho(s_{j-1}) \ne \texttt{null}$, $\rho'(s_0) = \rho(s_{j-1})$ and $\mu' = \mu$. Otherwise, $\rho'(s_0) = \ell$ where ℓ is fresh and $\mu' = \mu[\ell \mapsto npe]$, where npe is a new object of class `NullPointerException` containing only fresh locations. It is worth noting that, under these circumstances, for every $\mathsf{E} \in \mathbb{E}_{\tau'}$, $[\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = [\![\mathsf{E}]\!]\langle \rho', \mu \rangle$. According to Rule #17 of Definition 22, $\Pi(\langle \mathsf{A}_0, \ldots, \mathsf{A}_{n-1} \rangle) = \langle \mathsf{A}'_0, \ldots, \mathsf{A}'_i \rangle$, where $\mathsf{A}'_r = \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E})\}$ for $r \ne i$ and $\mathsf{A}'_r = \varnothing$ for $r = i$. Consider an expression $\mathsf{E} \in \mathsf{A}_r$. For $r \ne i$, the proof is analogous to the proof of the corresponding case for return t. If $r = i$, then $\mathsf{A}'_i = \varnothing$, and $\forall \mathsf{E} \in \mathsf{A}'_i . [\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = \rho'(v_i)$ trivially holds. □

B.3.3 Soundness of Exceptional Arcs

We show that, in the case of the propagation rules of the exceptional arcs, the exceptional concrete states belonging to the concretization of a correct approximation of the property of interest before a bytecode instruction is executed, are correctly propagated by the corresponding rule. It means that the propagation rules of the exceptional arcs simulating the exceptional executions of the bytecode instructions that can throw an exception are sound. Lemma 11 shows that this property actually holds.

Lemma 11 *The propagation rules* Rule #18 *and* Rule #20 *introduced by Definition 23 are sound. More precisely, let us consider an exceptional arc from a bytecode* ins *distinct from* call *and its propagation rule Π. Assume that* ins *has static type information τ at its beginning and τ' immediately after its exceptional execution. Then, for every $A \in \mathrm{Alias}_\tau$ we have:*

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

(we recall that ins is the semantics of ins*, see Fig. 5).*

Proof. The proof is analogous to the proof of Lemma 10 when ins = throw κ. □

The situation is a bit different when a method is invoked on a `null` receiver. In that case we require that the exceptional states launched by the method are included in the approximation of the property of interest after the call to that method. Lemma 12 shows that this property actually holds. Its proof can be found in Appendix B.

Lemma 12 *The propagation rule* Rule #19 *introduced by Definition 23 is sound. More precisely, consider an exceptional arc from a method invocation ins_C = call $m_1 \ldots m_n$ and its propagation rule Π, and let π be the number of its actual arguments (*this *included). Then, for each $1 \leq w \leq q$, and every $\sigma = \langle\langle \mathsf{l} \mid\mid v_{\pi-1} :: \ldots :: v_1 :: \texttt{null} : \mathsf{s}\rangle, \mu\rangle \in \gamma_\tau(A)$ (σ assigns* `null` *to the receiver of ins_C right before it is executed), where $A \in \mathrm{Alias}_\tau$ is an arbitrary abstract element, we have:*

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

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

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contain local variables L and $j \geq \pi$ stack elements $S = \{s_0, \ldots, s_{j-\pi}, \ldots, s_{j-1}\}$, where π is the number of parameters of method m_w (including this). Consider an arbitrary abstract element $A \in \mathrm{Alias}_\tau$ and a state $\sigma = \langle\rho, \mu\rangle = \langle\langle \mathsf{l} \mid\mid v_{\pi-1} :: \ldots :: v_1 :: \texttt{null} : \mathsf{s}\rangle, \mu\rangle \in \gamma_\tau(A)$. Then, by Rule 3 from Fig. 7, we have that $\mathsf{dom}(\tau') = L \cup \{s_0\}$, and the resulting state $\sigma' = \langle\rho', \mu'\rangle$ is such that for each $a \in \mathsf{dom}(\tau') \smallsetminus \{s_0\}$, $\rho'(a) = \rho(a)$, $\rho(s_0) = \ell$, where ℓ is a fresh location and $\mu' = \mu[\ell \mapsto npe]$, where npe is a new instance of `NullPointerException`. Hence, $\sigma' = \langle\langle \mathsf{l} \mid\mid \ell\rangle, \mu[\ell \mapsto npe]\rangle$. Moreover, according to Rule #19,

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements(E)}\} & \text{if } r \neq i \\ \varnothing & \text{if } r = i. \end{cases}$$

We must prove that $\sigma' \in \gamma_{\tau'}(\Pi(A))$, i.e., (Definition 19) that

$$\text{for each } 0 \leq r \leq i \text{ and every } \mathsf{E} \in \mathsf{A}_r, [\![\mathsf{E}]\!]\sigma' = [\![v_r]\!]\sigma'.$$

Let $\mathsf{E}_r \in \mathsf{A}'_r$, for an arbitrary $0 \leq r \leq i$. We distinguish the following cases:

– If $r \neq i$ then $\mathsf{E} \in \mathsf{A}_r$ and, by hypothesis,

$$[\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho(v_r) = \rho'(v_r).$$

Moreover, $\mathsf{noStackElements(E)}$ entails $\mathsf{variables(E)} \subseteq \mathsf{dom}(\tau')$, hence for every $v \in \mathsf{variables(E)}$, $\rho'(v) = \rho(v)$. Then, by Corollary 1, we have:

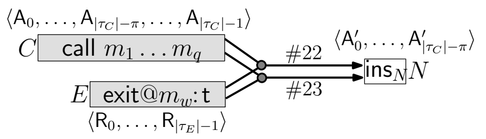
$$[\![\mathsf{E}]\!]\langle\rho', \mu\rangle = [\![\mathsf{E}]\!]\langle\rho, \mu\rangle = \rho'(v_r).$$

The only difference between μ and μ' is object o associated to a fresh location, which is not reachable from any other location. Therefore, μ and μ' behave like they were the same memory. Hence, for every $\mathsf{E} \in \mathbb{E}_{\tau'}$,

$$[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = [\![\mathsf{E}]\!]\langle\rho', \mu\rangle = \rho'(v_r).$$

– If $r = i$ then $\mathsf{A}'_n = \varnothing$, and therefore $\forall \mathsf{E} \in \mathsf{A}'_n.[\![\mathsf{E}]\!]\langle\rho', \mu'\rangle = \rho'(v_n)$ trivially holds.

Therefore, $\langle\langle \mathsf{l} \mid\mid \ell\rangle, \mu[\ell \mapsto npe]\rangle = \sigma' \in \gamma_{\tau'}(\Pi(A))$. □



$\langle \mathsf{A}_0,\ldots,\mathsf{A}_{|\tau_C|-\pi},\ldots,\mathsf{A}_{|\tau_C|-1}\rangle$
C
call $m_1 \ldots m_q$
E
exit@m_w: t
$\langle \mathsf{R}_0,\ldots,\mathsf{R}_{|\tau_E|-1}\rangle$
#22
#23
$\langle \mathsf{A}'_0,\ldots,\mathsf{A}'_{|\tau_C|-\pi}\rangle$
ins$_N$ N

Fig. 16 Arcs going into the node corresponding to ins_N.

B.3.4 Soundness of Parameter Passing Arcs

We show that the propagation rules of the parameter passing arcs are sound. Namely, this rule soundly approximates the behavior of the *makescope* function. Lemma 13 shows that this property actually holds.

Lemma 13 *The propagation rule* Rule #21 *introduced by Definition 24 is sound. More precisely, consider a parameter passing arc from a method invocation* $\mathsf{ins}_C = \mathsf{call}\ m_1 \ldots m_n$ *to the first bytecode of a callee* m_w*, for some* $w \in [1..k]$*, and its propagation rule* Π*. Assume that* ins_C *has static type information* τ *at its beginning and that* τ' *is the static type information at the beginning of* m_w*. Then, for every* $A \in \textsc{Alias}_\tau$ *we have:*

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

Proof. Let $\mathsf{dom}(\tau) = L \cup S$ contains local variables L and $j \geq \pi$ stack elements $S = \{s_0,\ldots,s_{j-\pi},\ldots,s_{j-1}\}$, where π is the number of parameters of method m_i (including `this`). Then, $\mathsf{dom}(\tau') = \{l_0,\ldots,l_{\pi-1}\}$. We choose an arbitrary abstract element $A = \langle \mathsf{A}_0,\ldots,\mathsf{A}_{n-1}\rangle \in \textsc{Alias}_\tau$, an arbitrary state $\sigma' = \langle \rho',\mu'\rangle \in (\textit{makescope}\ m_i)(\gamma_\tau(A))$, and we show that $\sigma' \in \gamma_{\tau'}(\Pi(A))$ i.e., (Definition 19) that

$$\text{for each } 0 \leq r < n' \text{ and every } \mathsf{E} \in \mathsf{A}_r,\ [\![\mathsf{E}]\!]\sigma' = [\![v_r]\!]\sigma'.$$

Note that, by the choice of σ', there exists $\sigma = \langle \rho,\mu\rangle \in \gamma_\tau(A)$ such that $\sigma' = \textit{ins}(\sigma)$ and such that, for each $0 \leq r < n$ and every $\mathsf{E} \in \mathsf{A}_r$, $[\![\mathsf{E}]\!]\sigma = [\![v_r]\!]\sigma = \rho(v_r)$.

According to Rule #21, $\Pi(\langle \mathsf{A}_0,\ldots,\mathsf{A}_{n-1}\rangle) = \langle \mathsf{A}'_0,\ldots,\mathsf{A}'_{\pi-1}\rangle$, where for each $0 \leq r < \pi$, $\mathsf{A}'_r = \varnothing$. Then, for each r, $\forall \mathsf{E} \in \mathsf{A}_r = \varnothing'.[\![\mathsf{E}]\!]\langle \rho',\mu'\rangle = \rho'(v_r)$ trivially holds. □

B.3.5 Soundness of Return and Side-Effects Arcs at Non-Exceptional Ends

We show that the propagation rules related to the return and side-effects arcs of an ACG are sound at non-exceptional end of a method or a constructor. Namely, in the case of a non-void method, the propagation rule of the return value arc enriches the resulting approximation of the definite aliasing information immediately after the call to that method by adding all those aliasing expressions that the returned value might correspond to. On the other hand, that method might modify the initial memory from which the method has been executed. These modifications must be captured by the propagation rules of the side-effects arcs. The approximation of the property of interest after the call to the method is, therefore, determined as the join ($\sqcup$) of the approximations obtained from the propagation rules of the return value and the side-effects arcs, and it is sound, as Lemma 14 shows. Lemma 15 handles the case of a void method, and therefore only the corresponding side-effects arc is considered there.

Lemma 14 *The propagation rules* Rule #22 *and* Rule #23 *introduced by Definitions 29 and 30 are sound at a* non-void *method return. Namely, let* $w \in [1..n]$ *and consider a return value and a side-effect arc from nodes* $\mathsf{C} = \boxed{\mathsf{call}\ m_1 \ldots m_n}$ *and* $\mathsf{E} = \boxed{\mathsf{exit@}m_w}$ *to a node* $\mathsf{Q} = \boxed{\mathsf{ins}_q}$ *and their propagation rules* $\Pi^{\#22}$ *and* $\Pi^{\#23}$*, respectively. We depict this situation in Fig. 16. Let* τ_c*,* τ_q *and* τ_e *be the static type information at* C*,* Q *and* E*, respectively, and let* d *be the* denotation *of* m_w*, i.e., a partial function from a state at its beginning to the corresponding state at its end. Then, for every* $A \in \textsc{Alias}_{\tau_c}$ *and* $R \in \textsc{Alias}_{\tau_e}$*, we have:*

$$d((\textit{makescope}\ m_w)(\gamma_{\tau_c}(A)) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}(\Pi^{\#22}(A,R) \sqcup \Pi^{\#23}(A,R)).$$

Proof. In the following we assume: $\mathsf{dom}(\tau_h) = L_h \cup S_h$, where $h \in \{C,E,N\}$; $\mathsf{dom}(\tau_h) = \{v_0,\ldots,v_{|\tau_h|-1}\}$, where $v_r = l_r$ when $0 \leq r < |L_h|$ and $v_r = s_{r-|L_h|}$ when $|L_h| \leq r < |\tau_a|$; π is the number of parameters of method m, $|\tau_C| - \pi \geq |L_C|$, $|\tau_N| = |\tau_C| - \pi + 1$, $L_N = L_C$ and $S_E = S_N = \{s_0\}$.

Consider two abstract elements: $A = \langle \mathsf{A}_0,\ldots,\mathsf{A}_{|\tau_C|-\pi},\ldots,\mathsf{A}_{|\tau_C|-1}\rangle \in \mathsf{A}_{\tau_C}$ and $R = \langle \mathsf{R}_0,\ldots,\mathsf{R}_{|\tau_E|-1}\rangle \in \mathsf{A}_{\tau_E}$, two concrete states corresponding to these abstract elements $\sigma_C = \langle \rho_C,\mu_C\rangle \in \gamma_{\tau_C}(A)$ and $\sigma_E = \langle \rho_E,\mu_E\rangle \in \gamma_{\tau_E}(R)$ and state $\sigma_N = \langle \rho_N,\mu_N\rangle = d((\textit{makescope}\ m_w)(\sigma_C)) \cap \Xi_{\tau_N}$. These states have to satisfy the following conditions imposed by Definition 11:

1. for every $0 \le r < |\tau_C| - \pi$, $\rho_N(v_r) = \rho_C(v_r)$;
2. $\rho_N(v_{|\tau_C|-\pi}) = \rho_E(v_{|\tau_E|-1})$;
3. $\mu_N = \mu_E$.

Moreover, since $\sigma_C \in \gamma_{\tau_C}(A)$, and $\sigma_E \in \gamma_{\tau_E}(A)$, by Definition 19, the following condition holds:

$$\forall 0 \le r < |\tau_h| . \forall \mathsf{E} \in \mathsf{A}_r . [\![\mathsf{E}]\!]\sigma_h = \rho_h(v_r), \tag{18}$$

where $h \in \{C, E\}$. Let us show that $\sigma_N \in \gamma_{\tau_N}(A')$, where $A' = \Pi^{\#22}(A, R) \sqcup \Pi^{\#23}(A, R)$, i.e., that Equation 18 also holds for $h = N$. By Definitions 18 and 20, we have $\Pi^{\#22}(A, R) \sqcup \Pi^{\#23}(A, R) = \langle \mathsf{A}'_0, \dots, \mathsf{A}'_{|\tau_C|-\pi} \rangle$, where A'_r are defined as follows:

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{safeExecution}(\mathsf{E}, \mathsf{ins}_c)\} & \text{if } r < |\tau_C| - \pi \\ \overbrace{\{\mathsf{E} = \mathsf{R}[\mathsf{E}_0, \dots, \mathsf{E}_{\pi-1}/l_0, \dots, l_{\pi-1}] | \mathsf{R} \in \mathsf{R}_{|\tau_E|-1} \wedge \mathsf{safeReturn}(\mathsf{R}, m_w) \wedge \mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_c)\}}^{X} & \\ \cup \underbrace{\{\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1, \dots, \mathsf{E}_{\pi-1}) \mid \mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_c)\}}_{Y} & \text{if } r = |\tau_C| - \pi \end{cases}$$

where maps noParameters, safeExecution, safeAlias and safeReturn are introduced in Definitions 25, 26, 27 and 28 respectively.

Let $\mathsf{E} \in \mathsf{A}_r$, for an arbitrary $0 \le r \le |\tau_C| - \pi$ and let us show that $[\![\mathsf{E}]\!]\sigma_N = \rho_N(v_r)$. We distinguish the following cases:

– if $r \ne |\tau_C| - \pi$, then $\mathsf{E} \in \mathsf{A}_r$ and $\mathsf{safeExecution}(\mathsf{E}, \mathsf{ins}_C)$ hold. It entails:
  1. $\mathsf{E} \in \mathsf{A}_r$, and therefore, by hypothesis (18),
  $$[\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle = \rho_C(v_r);$$
  2. $\mathsf{noParameters}(\mathsf{E})$ holds, and therefore $\mathsf{variables}(\mathsf{E}) \subseteq \{v_0, \dots, v_{|\tau_C|-\pi-1}\} \subseteq \mathsf{dom}(\tau_N)$, which entails
  $$\mathsf{E} \in \mathbb{E}_{\tau_N};$$
  3. $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}_C)$, i.e., execution of ins_C cannot affect evaluations of E, and therefore
  $$[\![\mathsf{E}]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle.$$

  Therefore,
  $$[\![\mathsf{E}]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle = \rho_C(v_r).$$

– if $r = |\tau_C| - \pi$, we distinguish two cases. In the first case, $\mathsf{E} \in X$, and therefore it has the following form: $\mathsf{E} = \mathsf{R}[\mathsf{E}_0, \dots, \mathsf{E}_{\pi-1}/l_0, \dots, l_{\pi-1}]$, where $\mathsf{R} \in \mathsf{R}_{|\tau_E|-1}$, $\mathsf{safeReturn}(\mathsf{R}, m_w)$ and $\mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_C)$ hold. According to Definition 27, the latter entails:
  1. for each $0 \le k < \pi$, $\mathsf{noParameters}(\mathsf{E}_k)$ holds, thus $\mathsf{variables}(\mathsf{E}_k) \subseteq \{v_0, \dots, v_{|\tau_C|-\pi-1}\} \subseteq \mathsf{dom}(\tau_N)$, which entails
  $$\mathsf{E}_k \in \mathbb{E}_{\tau_N};$$
  2. for each $0 \le k < \pi$, $\mathsf{E}_k \in \mathsf{A}_{|\tau_C|-\pi+k}$, and therefore, by hypothesis (18),
  $$[\![\mathsf{E}_k]\!]\langle \rho_C, \mu_C \rangle = \rho_C(v_{|\tau_C|-\pi+k});$$
  3. for each $0 \le k < \pi$, $\neg\mathsf{canBeAffected}(\mathsf{E}_k, \mathsf{ins}_C)$ holds, i.e., execution of ins_C cannot affect evaluations of E_k, and therefore
  $$[\![\mathsf{E}_k]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}_k]\!]\langle \rho_C, \mu_C \rangle;$$
  4. $$\begin{array}{c} \text{no evaluation of } \mathsf{E} \text{ (hence of } \mathsf{E}_0, \dots, \mathsf{E}_{\pi-1}, \mathsf{R}) \text{ might modify} \\ \text{any field that might be read by } \mathsf{E} \text{ (hence by } \mathsf{E}_0, \dots, \mathsf{E}_{\pi-1}, \mathsf{R}) \\ \text{or any element of an array of type } \mathsf{t}[\,] \text{ when } \mathsf{E} \text{ (hence } \mathsf{E}_0, \dots, \mathsf{E}_{\pi-1}, \mathsf{R}) \\ \text{might read an element of an array of type } \mathsf{t}'[\,] \text{ where } \mathsf{t}' \in \mathsf{compatible}(\mathsf{t}). \end{array} \tag{19}$$

  In other words, any evaluation of one of these expressions does not affect the value of any other of these expressions.

Since $\mathsf{R} \in \mathsf{R}_{|\tau_E|-1}$, by hypothesis (18) and by Definition 11,

$$[\![R]\!]\langle\rho_E,\mu_E\rangle = \rho_E(v_{|\tau_E|-1}) = \rho_N(v_{|\tau_C|-\pi}).$$

Moreover, according to Definition 28, $\mathsf{safeReturn}(\mathsf{R}, m_w)$ entails that $\mathsf{variables}(\mathsf{R}) \subseteq \{l_0,\dots,l_{\pi-1}\}$, i.e., every variable occurring in R corresponds to a formal parameter of m_w, and for each $l_k \in \mathsf{variables}(\mathsf{R})$, l_k is not modified by m_w, and therefore it has the same value at the beginning and at the end of execution of m_w. Since formal parameter l_k at E corresponds to the actual parameter $v_{|\tau_C|+k}$ at C, we obtain that for every $l_k \in \mathsf{variables}(\mathsf{R})$,

$$\rho_E(l_k) = \rho_C(v_{|\tau_C|+k}).$$

Evaluation of a variable l_k occurring in R in $\langle\rho_E,\mu_E\rangle$ gives

$$[\![l_k]\!]^*\langle\rho_E,\mu_E\rangle = \langle\rho_E(l_k),\mu_E\rangle = \langle\rho_E(l_k),\mu_N\rangle.$$

It is worth noting that the resulting memory does not change. On the other hand, evaluation of an alias expression $\mathsf{E}_k \in \mathsf{A}_{|\tau_C|-\pi+k}$ of $v_{|\tau_C|+k}$ at C in $\langle\rho_N,\mu_N\rangle$ might update the memory:

$$\begin{aligned}
[\![\mathsf{E}_k]\!]^*\langle\rho_N,\mu_N\rangle &= \langle[\![\mathsf{E}_k]\!]\langle\rho_C,\mu_C\rangle,\mu'_N\rangle && \text{[since } \neg\mathsf{canBeAffected}(\mathsf{E}_k,\mathsf{ins}_C)\text{ holds]}\\
&= \langle\rho_C(s_{p+k}),\mu'_N\rangle && \text{[by hypothesis (18)]}\\
&= \langle\rho_E(l_k),\mu'_N\rangle && \text{[since } \mathsf{safeReturn}(\mathsf{R}, m_w)\text{ holds]}
\end{aligned}$$

Therefore, E_k in $\langle\rho_N,\mu_N\rangle$ and l_k in $\langle\rho_E,\mu_E\rangle$ have the same value, but the resulting memory might be different. Nevertheless, (19) guarantees that of any $\mathsf{E}_0,\dots,\mathsf{E}_{\pi-1},\mathsf{R}$ in $\langle\rho_N,\mu'_N\rangle$ produces the same value which would be obtained if that expression were evaluated in $\langle\rho_N,\mu_N\rangle$, since μ'_N and μ_N might differ only on the fields and array elements that are not read by these expressions. Therefore, the results of these evaluations are $\rho_E(l_0),\dots,\rho_E(l_{\pi-1})$ respectively and evaluation of $\mathsf{E} = \mathsf{R}[\mathsf{E}_0,\dots,\mathsf{E}_{\pi-1}/l_0,\dots,l_{\pi-1}]$ in $\langle\rho_N,\mu_N\rangle$ gives the same value as the evaluation of R in $\langle\rho_E,\mu_E\rangle$. That value is

$$\rho_E(v_{|\tau_E|-1}) = \rho_N(v_{|\tau_C|-\pi}).$$

In the second case, $\mathsf{E} \in Y$, and $\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1,\dots,\mathsf{E}_{\pi-1})$, where $\mathsf{safeAlias}(\mathsf{E}, A, \mathsf{ins}_C)$ holds. The latter entails:

1. for each $0 \le k < \pi$, $\mathsf{noParameters}(\mathsf{E}_k)$ holds, thus $\mathsf{variables}(\mathsf{E}_k) \subseteq \{v_0,\dots,v_{|\tau_C|-\pi-1}\} \subseteq \mathsf{dom}(\tau_N)$, which entails $\mathsf{E}_k \in \mathbb{E}_{\tau_N}$. Hence, $\mathsf{E} \in \mathbb{E}_{\tau_N}$;
2. for each $0 \le k < \pi$, $\mathsf{E}_k \in \mathsf{A}_{|\tau_C|-\pi+k}$, and therefore, by hypothesis (18),
$$[\![\mathsf{E}_k]\!]\langle\rho_C,\mu_C\rangle = \rho_C(v_{|\tau_C|-\pi+k});$$
3. for each $0 \le k < \pi$, $\neg\mathsf{canBeAffected}(\mathsf{E}_k,\mathsf{ins}_C)$, i.e., execution of ins_C cannot affect evaluations of E_k, and therefore
$$[\![\mathsf{E}_k]\!]\langle\rho_N,\mu_N\rangle = [\![\mathsf{E}_k]\!]\langle\rho_C,\mu_C\rangle;$$
4. Analogously to (19), any evaluation of one of these expressions does not affect the value of any other of these expressions;

Statements 2, 3 and 4 above take us to the following conclusions:

$$\left.\begin{aligned}
[\![v_{|\tau_C|-\pi}]\!]^*\langle\rho_C,\mu_C\rangle &= \langle\rho_C(v_{|\tau_C|-\pi}),\mu_C\rangle\\
[\![\mathsf{E}_0]\!]^*\langle\rho_C,\mu_C\rangle &= \langle\rho_C(v_{|\tau_C|-\pi}),\mu_C^0\rangle\\
[\![\mathsf{E}_0]\!]^*\langle\rho_N,\mu_N\rangle &= \langle\rho_C(v_{|\tau_C|-\pi}),\mu_N^0\rangle
\end{aligned}\right\}\ \begin{array}{c}\mu_C,\mu_C^0 \text{ and } \mu_N^0 \text{ agree on}\\ \text{all fields and all array elements}\\ \text{that might be read by } \mathsf{E}_0,\dots,\mathsf{E}_{\pi-1},m\end{array}$$

$$\dots$$

$$\left.\begin{aligned}
[\![v_{|\tau_C|-1}]\!]^*\langle\rho_C,\mu_C\rangle &= \langle\rho_C(v_{|\tau_C|-1}),\mu_C\rangle\\
[\![\mathsf{E}_{\pi-1}]\!]^*\langle\rho_C,\mu_C^{\pi-2}\rangle &= \langle\rho_C(v_{|\tau_C|-1}),\mu_C^{\pi-1}\rangle\\
[\![\mathsf{E}_{\pi-1}]\!]^*\langle\rho_N,\mu_N^{\pi-2}\rangle &= \langle\rho_C(v_{|\tau_C|-1}),\mu_N^{\pi-1}\rangle
\end{aligned}\right\}\ \begin{array}{c}\mu_C,\mu_C^{\pi-1} \text{ and } \mu_N^{\pi-1} \text{ agree on}\\ \text{all fields and all array elements}\\ \text{that might be read by } \mathsf{E}_0,\dots,\mathsf{E}_{\pi-1},m\end{array}$$

These facts imply that the evaluations of $v_{|\tau_C|-\pi}.m(v_{|\tau_C|-\pi+1},\dots,v_{|\tau_C|-1})$ in $\langle\rho_C,\mu_C\rangle$ and of $\mathsf{E} = \mathsf{E}_0.m(\mathsf{E}_1,\dots,\mathsf{E}_{\pi-1})$ in $\langle\rho_N,\mu_N\rangle$ give the same value: namely, we execute method m on the object $\mu_C(\rho_C(v_{|\tau_C|-\pi})) = \mu_N^{\pi-1}(\rho_C(v_{|\tau_C|-\pi}))$ with parameters $\rho_C(v_{|\tau_C|-\pi+1}),\dots,\rho_C(v_{|\tau_C|-1})$ on memories μ_C and $\mu_N^{\pi-1}$ which agree on all the fields and all the array elements that might be read by m. Therefore, they will return the same value, and that value is, by Definition 11, memorized in $\rho_E(v_{|\tau_E|-1}) = \rho_N(v_{|\tau_C|-\pi})$. Hence,

$$[\![\mathsf{E}]\!]\langle\rho_N,\mu_N\rangle = [\![\mathsf{E}]\!]\langle\rho_C,\mu_C\rangle = \rho_N(v_{|\tau_C|-\pi}).$$

□

Lemma 15 *The propagation rule* Rule #23 *introduced by Definition 30 is sound at a* void *method return. Namely, let $w \in [1..n]$ and consider a side-effect arc from nodes* $\mathsf{C} = \boxed{\mathsf{call}\ m_1 \dots m_n}$ *and* $\mathsf{E} = \boxed{\mathsf{exit@}m_w}$ *to a node* $\mathsf{Q} = \boxed{\mathsf{ins}_q}$ *and its propagation rule $\Pi^{\#23}$. We depict this situation in Fig. 16, where the return value arc with the propagation rule #22 is omitted. Let τ_c, τ_q and τ_e be the static type information at* C, Q *and* E, *respectively, and let d be the* denotation *of m_w, i.e., a partial function from a state at its beginning to the corresponding state at its end. Then, for every $A \in \textsc{Alias}_{\tau_c}$ and $R \in \textsc{Alias}_{\tau_e}$, we have:*

$$d((makescope\ m_w)(\gamma_{\tau_c}(A)) \cap \Xi_{\tau_q} \subseteq \gamma_{\tau_q}(\Pi^{\#23}(A, R)).$$

Proof. The proof of this lemma is analogous to the case $r \neq |\tau_C| - \pi$ of the proof of Lemma 14. □

B.3.6 Soundness of Side-Effects and Exceptional Arcs at Exceptional Ends

This paragraph deals with the exceptional executions of the methods. Namely, the approximation of the definite aliasing information at the catch which captures the exceptional states of the method we are interested in, has to be affected by all possible modifications of the initial memory due to the side-effects of the method. This is the task of the propagation rules of the side-effects arcs. On the other hand, the final approximation of the definite expression aliasing property at the point of interest (catch) has to be affected by the exceptions raised by the method when it is invoked on `null`. As in the previous case, the approximation of the definite expression aliasing information is determined as the join ($\sqcup$) of the two approximations mentioned above, and Lemma 16 shows that it is correct.

Lemma 16 *The propagation rules* Rule #19 *and* Rule #23 *introduced by Definitions 23 and 30 s are sound when a method throws an exception. Namely, given nodes* $\mathsf{Q} = \boxed{\mathsf{catch}}$, $\mathsf{C} = \boxed{\mathsf{call}\ m_1 \dots m_n}$ *and* $\mathsf{E} = \boxed{\mathsf{exception@}m_w}$, *for a suitable $w \in [1..n]$, consider an exceptional arc from* C *to* Q *and a side-effect arc from* C *and* E *to* Q, *with their propagation rules $\Pi^{\#19}$ and $\Pi^{\#23}$, respectively. We depict this situation in Fig. 17. Let τ_c, τ_q and τ_e be the static type information at* C, Q *and* E, *respectively, and let d be the* denotation *of m_w, i.e., a partial function from a state at its beginning to the corresponding state at its end. Then, for every $A \in \textsc{Alias}_{\tau_c}$ and $R \in \textsc{Alias}_{\tau_e}$, we have:*

$$d((makescope\ m_w)(\gamma_{\tau_c}(A)) \cap \underline{\Xi}_{\tau_q} \subseteq \gamma_{\tau_q}(\Pi^{\#19}(A) \sqcup \Pi^{\#23}(A, R)).$$

Proof. In the following we assume: $\mathsf{dom}(\tau_h) = L_h \cup S_h$, where $h \in \{C, E, N\}$; $\mathsf{dom}(\tau_h) = \{v_0, \dots, v_{|\tau_h|-1}\}$, where $v_r = l_r$ when $0 \le r < |L_h|$ and $v_r = s_{r-|L_h|}$ when $|L_h| \le r < |\tau_a|$; π is the number of parameters of method m, $|\tau_C| - \pi \ge |L_C|$, $|\tau_N| = |\tau_C| - \pi + 1$, $L_N = L_C$ and $S_E = S_N = \{s_0\}$.

Consider two abstract elements: $A = \langle \mathsf{A}_0, \dots, \mathsf{A}_{|\tau_C|-\pi}, \dots, \mathsf{A}_{|\tau_C|-1} \rangle \in \mathsf{A}_{\tau_C}$ and $R = \langle \mathsf{R}_0, \dots, \mathsf{R}_{|\tau_E|-1} \rangle \in \mathsf{A}_{\tau_E}$, two concrete states corresponding to these abstract elements $\sigma_C = \langle \rho_C, \mu_C \rangle \in \gamma_{\tau_C}(A)$ and $\sigma_E = \langle \rho_E, \mu_E \rangle \in \gamma_{\tau_E}(R)$ and a state $\sigma_N = \langle \rho_N, \mu_N \rangle = d((makescope\ m_w)(\sigma_C)) \cap \underline{\Xi}_{\tau_N}$. These states have to satisfy the following conditions imposed by Definition 11:

1. for every $0 \le r < |\tau_C| - \pi$, $\rho_N(v_r) = \rho_C(v_r)$;
2. $\rho_N(v_{|\tau_C|-\pi}) = \rho_E(v_{|\tau_E|-1})$;
3. $\mu_N = \mu_E$.

Moreover, since $\sigma_C \in \gamma_{\tau_C}(A)$, and $\sigma_E \in \gamma_{\tau_E}(A)$, by Definition 19, the following condition holds:

$$\forall 0 \le r < |\tau_h|. \forall \mathsf{E} \in \mathsf{A}_r. [\![\mathsf{E}]\!] \sigma_h = \rho_h(v_r), \tag{20}$$

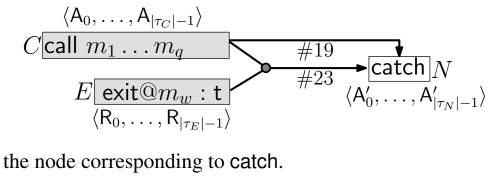

$\langle \mathsf{A}_0, \dots, \mathsf{A}_{|\tau_C|-1} \rangle$
C call $m_1 \dots m_q$
E exit@m_w : t
$\langle \mathsf{R}_0, \dots, \mathsf{R}_{|\tau_E|-1} \rangle$
#19
#23
catch N
$\langle \mathsf{A}'_0, \dots, \mathsf{A}'_{|\tau_N|-1} \rangle$

Fig. 17 Arcs going into the node corresponding to catch.

where $h \in \{C, E\}$. Let us show that $\sigma_N \in \gamma_{\tau_N}(A')$, where $A' = \Pi^{\#19}(A) \sqcup \Pi^{\#23}(A, R)$, i.e., that Equation 20 also holds for $h = N$. By Definitions 18 and 20, we have $\Pi^{\#19}(A) \sqcup \Pi^{\#23}(A, R) = \langle \mathsf{A}'_0, \dots, \mathsf{A}'_{|\tau_N|-\pi} \rangle$, where A'_r are defined as follows:

$$\mathsf{A}'_r = \begin{cases} \{\mathsf{E} \in \mathsf{A}_r \mid \mathsf{noStackElements}(\mathsf{E}) \wedge \neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}_C)\} & \text{if } r < |\tau_C| - \pi \\ \varnothing & \text{if } r = |\tau_C| - \pi \end{cases}$$

where $\mathsf{noStackElements}(\mathsf{E})$ is true if and only if $\mathsf{variables}(\mathsf{E}) \cap S_C = \varnothing$, i.e., if E contains no stack elements.

Let $\mathsf{E} \in \mathsf{A}_r$, for an arbitrary $0 \le r \le |\tau_C| - \pi$ and let us show that $[\![\mathsf{E}]\!]\sigma_N = \rho_N(v_r)$. We distinguish the following cases:

– if $r < |\tau_C| - \pi$, then E satisfies the following conditions:
  1. $\mathsf{noStackElements}(\mathsf{E})$ holds, and therefore $\mathsf{variables}(\mathsf{E}) \subseteq L_C = L_N \subseteq \mathsf{dom}(\tau_N)$, which entails $\mathsf{E} \in \mathbb{E}_{\tau_N}$;
  2. $\mathsf{E} \in \mathsf{A}_r$, and therefore, by hypothesis (20),

$$[\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle = \rho_C(v_r);$$

  3. $\neg\mathsf{canBeAffected}(\mathsf{E}, \mathsf{ins}_C)$, i.e., execution of ins_C cannot affect evaluations of E, and therefore

$$[\![\mathsf{E}]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle.$$

  Therefore,

$$[\![\mathsf{E}]\!]\langle \rho_N, \mu_N \rangle = [\![\mathsf{E}]\!]\langle \rho_C, \mu_C \rangle = \rho_C(v_r).$$

– if $r = |\tau_C| - \pi$, then $\mathsf{A}'_r = \varnothing$ and therefore $\forall \mathsf{E} \in \mathsf{A}'_N . [\![\mathsf{E}]\!]\langle \rho', \mu' \rangle = \rho'(v_r)$ trivially holds.

□

B.3.7 Theorem 2 from Section 4.4

Let $\langle b_{\mathit{first}(\mathtt{main})} \parallel \xi \rangle \Rightarrow^* \langle \boxed{\begin{array}{c}\mathsf{ins}\\ \mathit{rest}\end{array}} \begin{array}{l}\to b_1\\ \cdots\\ \to b_m\end{array} \parallel \sigma \rangle :: a$ be the execution of our operational semantics, from the block $b_{\mathit{first}(\mathtt{main})}$ starting with the first bytecode instruction of method $\mathtt{main}$, ins_0, and an initial state $\xi \in \Sigma_{\tau_0}$ (containing no aliasing information), to a bytecode instruction ins and assume that this execution leads to a state $\sigma \in \Sigma_\tau$, where τ_0 and τ are the static type information at ins_0 and ins, respectively. Moreover, let $A \in \mathrm{A{\scriptstyle LIAS}}_\tau$ be the reachability approximation at ins, as computed by our reachability analysis starting from A_0. Then, $\sigma \in \gamma_\tau(A)$ holds.

Proof. The blocks in the configurations of an activation stack, but the topmost, cannot be empty and with no successor. This is because the configurations are only stacked by rule (2) of Fig. 7 and if *rest* is empty there, then $m \ge 1$ or otherwise, the code ends with a call bytecode with no return, which is illegal in Java bytecode [22].

We proceed by induction on the length n of the execution

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

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

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

$$\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi \rangle \Rightarrow^{n_p} \langle \overbrace{\boxed{\begin{array}{c}\mathsf{ins}_p\\ \mathit{rest}_p\end{array}}}^{b_p} \begin{array}{l}\to b'_1\\ \cdots\\ \to b'_{m'}\end{array} \parallel \sigma_p \rangle :: a_p \Rightarrow^{n+1-n_p} \langle b_q \parallel \sigma_q \rangle :: a_q \qquad (21)$$

with $0 \le n_p \le n$, that is, it must have a strict prefix of length n_p whose final activation stack has the topmost configuration with a non-empty block b_p. Let such n_p be maximal. Given a bytecode ins_a, let τ_a and A_a be the static type information and the approximation of the property of interest at the ACG node $\boxed{\mathsf{ins}_a}$ respectively. By inductive hypothesis we know that $\sigma_p \in \gamma_{\tau_p}(A_p)$ and we show that also $\sigma_q \in \gamma_{\tau_q}(A_q)$ holds. We distinguish on the basis of the rule of the operational semantics that is applied at the beginning of the derivation $\Rightarrow^{n+1-n_p}$ in Equation 21.

Rule (1)**.** Then $ins_p(\sigma_p)$ is defined and ins_p is not a call.

<u>**case a:** ins_p is not a return nor a throw</u>

If $rest_p$ is non-empty then, by the maximality of n_p, (21) must be

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

Otherwise $m' \ge 1$ must hold (legal Java bytecode can only end with a return or a $\mathsf{throw}\ \kappa$) and, by the maximality of n_p, it must be the case that $b_q = b'_h$ for a suitable $1 \le h \le m'$, so that (21) must have the form

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

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

$$\begin{aligned}
\Xi_{\tau_q} \ni \sigma_q = \quad & ins_p(\sigma_p) & \\
\in \quad & ins_p(\gamma_{\tau_p}(A_p)) \cap \Xi_{\tau_p} & \text{[By hypothesis and pointwise extension of } ins_p\text{]}\\
\subseteq \quad & \gamma_{\tau_q}(\Pi(A_p)) = \gamma_{\tau_q}(A_q) & \text{[By Lemmas 9 and 11].}
\end{aligned}$$

<u>**case b:** ins_p is a $\mathsf{return}\ \mathsf{t}$</u>

We show the case when $\mathsf{t} \ne \mathsf{void}$, since the other case is simpler (there is no return value to consider). Then $rest_p$ is empty and $m' = 0$ (no code is executed after a return in legal Java bytecode, but the method terminates) and since $ins_p(\sigma_p) \in \Xi$ (definition of $\mathsf{return}\ \mathsf{t}$), (21) must be in one of these two forms, depending on the emptiness of block b in Rule (4):

$$\begin{aligned}
\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi\rangle \Rightarrow^{n_p} & \langle \underbrace{\boxed{\mathsf{return\ t}}}_{b_p} \parallel \underbrace{\langle\langle \mathsf{l}_p \parallel t :: \mathsf{s}_p\rangle, \mu_p\rangle\rangle}_{\sigma_p} :: \underbrace{\overbrace{\langle b_q \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c\rangle, \mu_c\rangle\rangle}^{\texttt{call-time}} :: a_q}_{a_p}\\
\overset{(1)}{\Rightarrow}\quad & \langle \boxed{} \parallel \langle\langle \mathsf{l}_p \parallel t\rangle, \mu_p\rangle\rangle :: \langle b_q \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c\rangle, \mu_c\rangle\rangle :: a_q\\
\overset{(4)}{\Rightarrow}\quad & \langle b_q \parallel \overbrace{\langle\langle \mathsf{l}_c \parallel t :: \mathsf{s}_c\rangle, \mu_p\rangle}^{\sigma_q}\rangle :: a_q
\end{aligned} \tag{22}$$

or

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

where, in the latter case, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \leq h \leq m'$. We only prove the case for (22), the other being similar. Consider the configuration `call-time`. Since only Rule (2) can stack configurations, `call-time` was the topmost one when a call was executed and, for a suitable $1 \leq w \leq n$, (22) must have the form

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

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

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

$$\begin{array}{rll}
\sigma_e & = \mathit{return}\ \mathsf{t}(\sigma_p) & \\
& \in \mathit{return}\ \mathsf{t}(\gamma_{\tau_p}(A_p)) & \text{[By hypothesis and monotonicity of } \mathit{return}\text{]} \\
& \subseteq \gamma_{\tau_e}(\Pi(A_p)) = \gamma_{\tau_e}(A_e) & \text{[By Lemma 10].}
\end{array}$$

In this case there are two $2-1$ arcs (a return value and a side-effect arc) going into $\boxed{\mathsf{ins}_q}$ (see Fig. 16), and A_c and A_e represent the correct approximations of the property of interest at the sources of these arcs. Let $A_q = \Pi^{RV}(A_c, A_e) \sqcup \Pi^{SE}(A_c, A_e)$, where Π^{RV} and Π^{SE} are the propagation rules of the return value and side-effect arcs respectively. We have

$$\begin{array}{rll}
\Xi_{\tau_q} \ni \sigma_q & = d((\mathit{makescope}\ m_w)(\sigma_c)) & \\
& \in d((\mathit{makescope}\ m_w)(\gamma_{\tau_c}(A_c))) \cap \Xi_{\tau_q} & \text{[By hypothesis and} \\
& & \text{ monotonicity of } d\text{]} \\
& \subseteq \gamma_{\tau_q}(\Pi^{RV}(A_c, A_e) \sqcup \Pi^{SE}(A_c, A_e)) & \text{[By Lemma 14]} \\
& = \gamma_{\tau_q}(A_q). &
\end{array}$$

<u>**case c:** ins_p is a throw</u>

If rest_p is empty and $m' > 0$, the execution (21) must have the form

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

where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \leq h \leq m'$. If rest_p is non-empty, the execution (21) must have the form

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

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

$$\begin{aligned}
\underline{\Xi}_{\tau_q} \ni \sigma_q &= \textit{throw}\ \kappa(\sigma_p)\\
&\in \textit{throw}\ \kappa(\gamma_{\tau_p}(A_p)) \cap \underline{\Xi}_{\tau_q} && \text{[By hypothesis and monotonicity of } \textit{throw}\text{]}\\
&\subseteq \gamma_{\tau_q}(\Pi(A_p)) = \gamma_{\tau_q}(A_q) && \text{[By Lemma 11].}
\end{aligned}$$

If $rest_p$ is empty and $m' = 0$, the execution (21) must have one of these two forms, depending on the emptiness of block b in Rule (5):

$$\begin{aligned}
\langle b_{\mathit{first}(\mathsf{main})} \parallel \xi\rangle \Rightarrow^{n_p} & \langle \underbrace{\boxed{\text{throw}\,\kappa}}_{b_p} \parallel \underbrace{\langle\langle \mathsf{l}_p \parallel e :: \mathsf{s}_p\rangle, \mu_p\rangle}_{\sigma_p}\rangle :: \overbrace{\underbrace{\langle b_q \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c\rangle, \mu_c\rangle\rangle :: a_q}_{a_p}}^{\texttt{call-time}}\\
\overset{(1)}{\Rightarrow}\ & \langle \boxed{} \parallel \underline{\langle\langle \mathsf{l}_p \parallel e\rangle, \mu_p\rangle}\rangle :: \langle b_q \parallel \langle\langle \mathsf{l}_c \parallel \mathsf{s}_c\rangle, \mu_c\rangle\rangle :: a_q\\
\overset{(5)}{\Rightarrow}\ & \langle b_q \parallel \overbrace{\underline{\langle\langle \mathsf{l}_c \parallel e\rangle, \mu_p\rangle}}^{\sigma_q}\rangle :: a_q,
\end{aligned} \tag{23}$$

or

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

where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \le h \le m'$. We only prove (23), the other being similar. Consider configuration `call-time`. Since only Rule (2) can stack configurations, it was the tompost one when the call was executed and (23) must have the form

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

where j is the number of stack elements before $\text{ins}_c = \text{call } m_1 \ldots m_q$ is executed, π is the number of parameters of method m, $b_{\mathit{first}(m_w)}$ is the block where the implementation of m_w starts and the rules in the portion $\Rightarrow^{n_p - n_c - 1}$ never make the stack lower than at the beginning of that portion. We recall that, only in this proof, we slightly abuse notation and use $v_0, \ldots, v_{j-1}$ to denote the values of variables $v_0, \ldots, v_{j-1}$ in σ_c. By the semantics of Java bytecode, since $\sigma_q \in \underline{\Xi}$, the only possibility for ins_q is to be a catch.

Consider $\sigma_c = \langle\langle \mathsf{l}_c \parallel v_{j-1} :: \ldots :: v_{j-\pi} :: \ldots :: v_0\rangle, \mu_c\rangle$ and $\sigma_p = \langle\langle \mathsf{l}_p \parallel e :: \mathsf{s}_p\rangle, \mu_p\rangle$. By inductive hypothesis for n_c and n_p we know that $\sigma_c \in \gamma_{\tau_c}(A_c)$ and $\sigma_p \in \gamma_{\tau_p}(A_p)$. Let $\sigma_e = \textit{throw}\ \kappa(\sigma_p) = \langle\langle \mathsf{l}_p \parallel e\rangle, \mu_p\rangle$. Then, the ACG contains a final arc from $\boxed{\text{throw}\,\kappa}$ to $\boxed{\text{exit@}m_w\text{:t}}$, for a suitable $1 \le w \le n$, $A_e = \Pi(A_p)$, where Π is the propagation rule #13 (Definition 20), and the following relations hold

$$\begin{aligned}
\sigma_e &= \textit{throw}\ \mathsf{t}(\sigma_p)\\
&\in \textit{throw}\ \mathsf{t}(\gamma_{\tau_p}(A_p)) && \text{[By hypothesis and monotonicity of } \textit{throw}\text{]}\\
&\subseteq \gamma_{\tau_e}(\Pi(A_p)) = \gamma_{\tau_e}(A_e) && \text{[By Lemma 10].}
\end{aligned}$$

In this case there are two arcs (a side-effect and an exceptional arc) going into $\boxed{\textsf{catch}}$ (see Fig. 17), and A_c and A_e represent the correct approximations of the property of interest at the sources of these arcs. Let $A_q = \varPi^E(A_c) \sqcup \varPi^{SE}(A_c, A_e)$, where $\varPi^E$ and $\varPi^{SE}$ are the propagation rules of the exceptional and the side-effects arcs respectively. We have

$$\begin{aligned}
\underline{\varXi}_{\tau_q} \ni \sigma_q &= d((\mathit{makescope}\ m_w)(\sigma_c)) \\
&\in d((\mathit{makescope}\ m_w)(\gamma_{\tau_c}(A_c))) \cap \underline{\varXi}_{\tau_q} && \text{[By hypothesis and} \\
& && \text{monotonicity of } d] \\
&\subseteq \gamma_{\tau_q}(\varPi^E(A_c) \sqcup \varPi^{SE}(A_c, A_e)) = \gamma_{\tau_q}(A_q) && \text{[By Lemma 16].}
\end{aligned}$$

Rule (2). By definition of *makescope*, (21) must have the form

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

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

$$\begin{aligned}
\sigma_q &= \mathit{makescope}(\sigma_p) \\
&\in \mathit{makescope}(\gamma_{\tau_p}(A_p)) && \text{[By hypothesis and monotonicity of } \mathit{makescope}] \\
&\subseteq \gamma_{\tau_q}(\varPi(A_p)) = \gamma_{\tau_q}(A_q) && \text{[By Lemma 13].}
\end{aligned}$$

Rule (3). Let i and j be the number of local variables and stack elements before $\textsf{ins}_p = \textsf{call}\ m_1 \dots m_q$ is executed and π be the number of parameters of methods m_w. In this case, (21) must have the form

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

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

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

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

$$\begin{aligned}
\underline{\varXi}_{\tau_q} \ni \sigma_q &= d((\mathit{makescope}\ m_w)(\sigma_p)) \cap \underline{\varXi}_{\tau_q} \\
&\subseteq \gamma_{\tau_q}(\varPi(A_p)) = \gamma_{\tau_q}(A_q) && \text{[By Lemma 12].}
\end{aligned}$$

$\square$