

# Definite Expression Aliasing Analysis for Java Bytecode

Đurica Nikolić<sup>1,2</sup> and Fausto Spoto<sup>1</sup>

<sup>1</sup> Dipartimento di Informatica, University of Verona

<sup>2</sup> The Microsoft Research - University of Trento Center for Computational and Systems Biology  
{durica.nikolic, fausto.spoto}@univr.it

**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 is a useful piece of information for a 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, defined and proved correct in the abstract interpretation framework. Moreover, we show the benefits of our definite expression aliasing analysis for nullness and termination analysis with Julia.

## 1 Introduction

Static analyses infer properties of computer programs and prove those programs secure for some classes of bugs. 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 [9], whose instructions operate on stack or 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;  
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.

```

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

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. Moreover, when a bytecode instruction affects a variable, this modification is propagated to all the expressions containing that variable. This way we can determine different properties about the aliased expressions. 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). We show, experimentally, that this analysis supports nullness and termination analyses of our tool Julia, but this paper is only concerned with the expression aliasing analysis itself. Our analysis has been proven sound, but due to space limitations, proofs can only be found in an extended version of this paper [10].

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 `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).

Our definite expression aliasing analysis is constraint-based: a large constraint is built from the program, whose solution is a sound approximation of the expressions aliased to each variable at each program point. The correctness of our analysis is proved in the abstract interpretation framework [5] and follows from a correct treatment of the potential side-effects of statements.

**Related Work.** Alias analysis belongs to the large group of pointer analyses [7], 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 [12]). On the other hand, definite alias analysis 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, the authors of [6] 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 a constraint-based analysis is not new: we have already used it to formalize possible analyses [14,11]. However, the construction of the constraint and the definition of the propagation rules are different there. A static analysis that over-approximates the set of fields that might be `null` at some point has been introduced in [13]. More complex expressions than fields are not considered there, though. 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. Fields can be sometimes transformed into local variables before a static analysis is performed [4], but this requires a preliminary modification of the code and we want to deal with more general expressions than just fields.

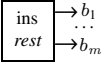
## 2 Operational Semantics

This section introduces a formal operational semantics of our target, Java bytecode-like language, used also in [14,11] and inspired by the standard informal semantics [8].

The target language contains the following instructions: `const`  $x$ , `dup`, `load`, `store`, `inc`, `ifeq`, `ifne`, `new`, `getfield`, `putfield`, `throw` and `call`. They abstract whole classes of Java bytecode instructions such as `iconst_x`, `ldc`, `bipush`, `dup`, `iload`, `aload`, `istore`, `astore`, `iinc`, `ifeq`, `ifne`, `if_null`, `if_nonnull`, `new`, `getfield`, `putfield`, `athrow`, `invokevirtual`, and `invokespecial`. In addition, we introduce an instruction `op` corresponding to the arithmetic bytecode instructions such as `iadd`, `isub`, `imul`, `idiv` and `irem`, and an instruction `catch` starting the exception handlers. An informal semantics of this language is provided at the end of this section. We analyze programs at bytecode level for several reasons: there is a small number of bytecode instructions, compared to varieties of source statements; bytecode lacks complexities such as inner classes; our implementation of definite expression aliasing is at bytecode level as well, which brings formalism, implementation and correctness proofs closer.

For simplicity, we assume that the only primitive type is `int` and that reference types are *classes* containing *instance fields* and *instant methods* only. Our implementation handles all Java types and bytecodes, as well as classes with static fields and methods.

**Definition 1 (Classes).** We let  $\mathbb{K}$  denote the set of classes and we define  $\mathbb{T} = \{\text{int}\} \cup \mathbb{K}$ , the set of all possible types. Every class  $\kappa \in \mathbb{K}$  might have instance fields  $\kappa.f: \mathfrak{t}$  (field  $f$  of type  $\mathfrak{t} \in \mathbb{T}$  defined in class  $\kappa$ ) and instance methods  $\kappa.m(\vec{\mathfrak{t}}): \mathfrak{t}$  (method  $m$ , defined in class  $\kappa$ , with arguments of type  $\vec{\mathfrak{t}}$  taken from  $\mathbb{T}$ , returning a value of type  $\mathfrak{t} \in \mathbb{T} \cup \{\text{void}\}$ ), where  $\kappa$ ,  $\vec{\mathfrak{t}}$ , and  $\mathfrak{t}$  are often omitted. We let  $\mathbb{F}(\kappa)$  denote the set of all fields contained in  $\kappa$ .

We analyze bytecode preprocessed into a control flow graph (CFG), i.e., a directed graph of *basic blocks*, with no jumps inside them.  denotes a block of code starting at instruction `ins`, possibly followed by a sequence of instructions `rest` and linked to  $m$  subsequent blocks  $b_1, \dots, b_m$ .

*Example 1.* Consider the Java method `delayMinBy` and its corresponding graph of basic blocks of bytecode instructions given in Fig. 2. The latter contains a branch since the `getfield min` 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. Every bytecode instruction except `return` and `throw` always has one or more immediate successors. The latter are placed at the end of a method or constructor and typically have no successors. ■

Bytecode instructions operate on *variables*, which encompass both stack elements and local variables. A standard algorithm [8] infers their static types.

**Definition 2 (Type environment).** Let  $V$  be the set of variables from  $L = \{l_0, \dots, l_{i-1}\}$  ( $i$  local variables) and  $S = \{s_0, \dots, s_{j-1}\}$  ( $j$  stack elements). A type environment is a function  $\tau: V \rightarrow \mathbb{T}$ , and its domain is written as  $\text{dom}(\tau)$ . The set of all type environments is  $\mathcal{T}$ . For simplicity, we write  $\text{dom}(\tau) = \{v_0, \dots, v_{i+j-1}\}$ , where  $v_r = l_r$  if  $0 \leq r < i$  and  $v_r = s_{r-i}$  if  $i \leq r < i + j$ . Moreover, we let  $|\tau|$  denote  $|\text{dom}(\tau)| = i + j$ .

**Definition 3 (State).** A value is an element of  $\mathbb{V} = \mathbb{Z} \cup \mathbb{L} \cup \{\text{null}\}$ , where  $\mathbb{L}$  is an infinite set of memory locations. A state over a type environment  $\tau$  is  $\langle \rho, \mu \rangle$ , where

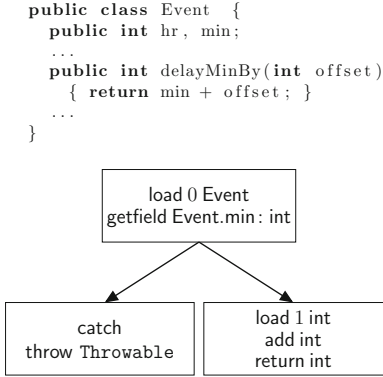
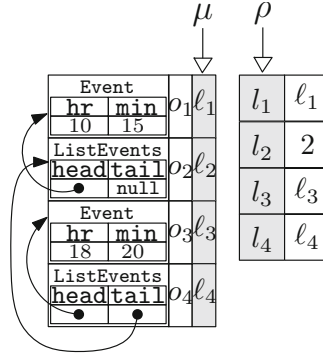


Fig. 2. Our running example

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

$\rho \in \text{dom}(\tau) \rightarrow \mathbb{V}$  is called environment and assigns a value to each variable from  $\text{dom}(\tau)$ , while  $\mu \in \mathbb{M}$  is called memory and binds locations to objects. Every object  $o$  has class  $o.k$  and an internal state  $o.\phi$  mapping each field  $f$  of  $o$  into its value  $(o.\phi)(f)$ . The set of all states over  $\tau$  is  $\Sigma_\tau$ . We assume that states are well-typed, i.e., variables hold values consistent with their static types.

The JVM supports exceptions and we distinguish between *normal* and *exceptional* states. These latter arise *immediately after* a bytecode throwing an exception and in that case there is only one element on the stack: a location bound to the thrown exception.

*Example 2.* Let  $\tau = [l_1 \mapsto \text{ListEvents}; l_2 \mapsto \text{int}; l_3 \mapsto \text{Event}; l_4 \mapsto \text{ListEvents}] \in \mathcal{T}$ , where class `ListEvents` defines two fields: `head` of type `Event` and `tail` of type `ListEvents`. Fig. 3 shows a state  $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ . Environment  $\rho$  maps variables  $l_1, l_2, l_3$  and  $l_4$  to values  $l_2, 2, l_3$  and  $l_4$ , respectively. Memory  $\mu$  maps locations  $l_2$  and  $l_4$  to objects  $o_2$  and  $o_4$  of class `ListEvents`, and  $l_3$  to  $o_3$  of class `Event`. Objects are shown as boxes in  $\mu$  with a class tag and a local environment mapping fields to integers, locations or `null`. For instance, fields `head` and `tail` of  $o_4$  contain locations  $l_3$  and  $l_2$ , respectively. ■

The semantics of an instruction `ins` of our target language is a partial map  $\text{ins} : \Sigma_\tau \rightarrow \Sigma_\tau$  from *initial* to *final* states. Number of local variables and stack elements at its start, as well as their static types, are specified by  $\tau \in \mathcal{T}$ . In the following we assume that  $\text{dom}(\tau)$  contains  $i$  local variables and  $j$  stack elements. Moreover, we suppose that the semantics is undefined for input states of wrong sizes or types, as is required in [8]. The formal semantics is given in [14] and we discuss it informally below.

**Basic Instructions.** `const x` pushes  $x \in \mathbb{Z}$  on top of the stack. Like any other instruction except `catch`, it is defined only when the JVM is in a normal state. `catch` starts instead the exceptional handlers from an exceptional state and is, therefore, undefined on a normal state. `dup t` duplicates the top of the stack, of type  $t$ . `load k t` pushes on the stack the value of local variable number  $k$ ,  $l_k$ , which must exist and have type  $t$ . Conversely, `store k t` pops the top of the stack of type  $t$  and writes it in local variable  $l_k$ ; it might potentially enlarge the set of local variables. In our formalization, conditional bytecodes are used in complementary pairs (such as `ifne t` and `ifeq t`), at a conditional branch. For

instance, `ifeq t` checks whether the top of the stack, of type `t`, is 0 when `t = int` or `null` when `t ∈ ℚ`. Otherwise, its semantics is undefined. Bytecode `inc k x` increments the integer held in local variable  $l_k$  by a constant  $x$ . Bytecode `op` pops two integers from the operand stack, performs a suitable binary algebraic operation on them, and pushes the integer result back onto the stack. `op` may be `add`, `sub`, `mul`, `div` and `rem`, and the corresponding algebraic operations are  $+$ ,  $-$ ,  $\times$ ,  $\div$  and  $\%$ .

**Object-Manipulating Instructions.** These create or access objects in memory. `new  $\kappa$`  pushes on the stack a reference to a new object  $o$  of class  $\kappa$ , whose fields are initialized to a default value: `null` for reference fields, and 0 for integer fields [8]. `getfield  $f$`  reads field  $f$  of a receiver object  $r$  popped from the stack. `putfield  $f$`  writes the top of the stack inside field  $f$  of the object pointed to by the underlying value  $r$ .

**Exception-Handling Instructions.** `throw  $\kappa$`  throws the top of the stack, whose type  $\kappa$  is a subclass of `Throwable`. `catch` starts an exception handler: it takes an exceptional state and transforms it into a normal state at the beginning of the handler. After `catch`, an appropriate handler dependent on the run-time class of the exception is selected.

**Method Call and Return.** We use an activation stack of states. Methods can be re-defined in object-oriented code, so a call instruction has the form `call  $m_1 \dots m_k$` , enumerating an over-approximation of the set of its possible run-time targets. See [14] for details.

### 3 Alias Expressions

In this section, we define our expressions of interest (Definition 4), their *non-standard evaluation* (Definition 6), which might modify the content of some memory locations and we introduce the notion of *alias expression* (Definition 7). Moreover, we specify in which cases *a bytecode instruction might affect the value of an expression* (Definition 8), and when *the evaluation of an expression might modify a field* (Definition 9).

**Definition 4 (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  $\Sigma_\tau$ . We define the set of expressions over  $\text{dom}(\tau)$ :  $\mathbb{E}_\tau \ni E ::= n \mid v \mid E \oplus E \mid E.f \mid E.m(E, \dots)$ , where  $n \in \mathbb{Z}$ ,  $v \in \text{dom}(\tau)$ ,  $\oplus \in \{+, -, \times, \div, \%\}$ ,  $f \in \mathcal{F}_\tau$  and  $m \in \mathcal{M}_\tau$ .

**Definition 5.** We define a map  $\text{vars} : \mathbb{E}_\tau \rightarrow \wp(\text{dom}(\tau))$  yielding the variables occurring in an expression and a map  $\text{flds} : \mathbb{E}_\tau \rightarrow \wp(\mathcal{F}_\tau)$  yielding the fields that might be read during the evaluation of an expression, for a given  $\tau \in \mathcal{T}$  as:

$E$	$\text{vars}(E)$	$\text{flds}(E)$
$n \in \mathbb{Z}$	$\emptyset$	$\emptyset$
$v \in \text{dom}(\tau)$	$\{v\}$	$\emptyset$
$E_1 \oplus E_2$	$\text{vars}(E_1) \cup \text{vars}(E_2)$	$\text{flds}(E_1) \cup \text{flds}(E_2)$
$E.f$	$\text{vars}(E)$	$\text{flds}(E) \cup \{f\}$
$E_0.m(E_1, \dots, E_\pi)$	$\bigcup_{i=0}^{\pi} \text{vars}(E_i)$	$\bigcup_{i=0}^{\pi} \text{flds}(E_i) \cup \{f \mid m \text{ might read } f\}$

Note that the definition of flds requires a preliminary computation of the fields possibly read by a method  $m$ , which might just be a transitive closure of the field  $f$  for which a `getfield` occurs in  $m$  or in at least one method invoked by  $m$ . For instance, if the static type of the local variable  $l_2$  is `Event`, then expression  $E = l_2.\text{delayMinBy}(15)$  satisfies the following equalities:  $\text{vars}(E_2) = \{l_2\}$ , and  $\text{flds}(E_2) = \{\text{min}\}$ . The latter follows from the fact that `delayMinBy` contains only one `getfield` and no call instruction (Ex. 1). There exist some more precise approximations of this useful piece of information, e.g., the one determined by our Julia tool. Anyway, in the absence of this approximation, we can always assume the least precise sound hypothesis: every method can read every field.

Some of the expressions defined above represent the result of a method invocation. Their evaluation, in general, might modify the memory, 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  $\mu'$  is the updated memory obtained from  $\mu$  after the evaluation of  $e$ .

**Definition 6 (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  $\llbracket \cdot \rrbracket^* : \mathbb{E}_\tau \rightarrow \mathbb{V} \times \mathbb{M}$  defined as:

- for every  $n \in \mathbb{Z}$ ,  $\llbracket n \rrbracket^* \sigma = \langle n, \mu \rangle$ , while for every  $v \in \text{dom}(\tau)$ ,  $\llbracket v \rrbracket^* \sigma = \langle \rho(v), \mu \rangle$ ;
- $\llbracket E_1 \oplus E_2 \rrbracket^* \sigma$  is defined only if  $\llbracket E_1 \rrbracket^* \sigma = \langle w_1, \mu_1 \rangle$ ,  $\llbracket E_2 \rrbracket^* \langle \rho, \mu_1 \rangle = \langle w_2, \mu_2 \rangle$  and  $w_1, w_2 \in \mathbb{Z}$ . In that case  $\llbracket E_1 \oplus E_2 \rrbracket^* \sigma = \langle w_1 \oplus w_2, \mu_2 \rangle$ , otherwise it is undefined;
- $\llbracket E.f \rrbracket^* \sigma$  is defined only if  $\llbracket E \rrbracket^* \sigma = \langle \ell, \mu_1 \rangle$ ,  $\ell \in \mathbb{L}$  and  $f \in \mathbb{F}(\mu_1(\ell).\kappa)$ . In that case  $\llbracket E.f \rrbracket^* \sigma = \langle (\mu_1(\ell).\phi)(f), \mu_1 \rangle$ ;
- in order to compute  $\llbracket E_0.m(E_1, \dots, E_\pi) \rrbracket^* \sigma$ , we determine  $\llbracket E_0 \rrbracket^* \langle \rho, \mu \rangle = \langle w_0, \mu_0 \rangle$ , and for each  $1 \leq i < \pi$ , we evaluate  $E_{i+1}$  in the state  $\langle \rho, \mu_i \rangle$ :  $\llbracket E_{i+1} \rrbracket^* \langle \rho, \mu_i \rangle = \langle w_{i+1}, \mu_{i+1} \rangle$ . If  $w_0 \in \mathbb{L}$ , we run  $m$  on the object  $\mu_\pi(w_0)$  with parameters  $w_1, \dots, w_\pi$  and if it terminates with no exception, the result of the evaluation is the pair of  $m$ 's return value  $w$  and the memory  $\mu'$  obtained from  $\mu_\pi$  as a side-effect of  $m$ .

We write  $\llbracket E \rrbracket \sigma$  for the value of  $E$ , without the updated memory.

**Definition 7 (Alias Expression).** We say that an expression  $E \in \mathbb{E}_\tau$  is an alias expression of a variable  $v \in \text{dom}(\tau)$  in a state  $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$  if and only if  $\llbracket E \rrbracket \sigma = \rho(v)$ .

We specify when the value of an expression might be affected by an instruction's execution. An information about the fields that might be modified during the execution of the methods is required. Without that information, the analysis would be less precise.

**Definition 8 (canBeAffected).** Let  $\tau$  and  $\tau'$  be the static type information at and immediately after an instruction  $\text{ins}$ . We define a map  $\text{canBeAffected}(\cdot, \text{ins}) : \mathbb{E}_\tau \rightarrow \{\text{true}, \text{false}\}$  which, for every expression  $E \in \mathbb{E}_\tau$  determines whether  $E$  might be affected by  $\text{ins}$ :

$E$	$\text{canBeAffected}(E, \text{ins})$
$n \in \mathbb{V}$	<i>false</i>
$v \in \text{dom}(\tau)$	$(v \notin \text{dom}(\tau')) \vee (\text{ins} \in \{\text{inc } k \ x, \text{store } k \ t\} \wedge v = l_k)$ $\vee (\text{ins} = \text{getfield } f \wedge v = s_{j-1}) \vee (\text{ins} = \text{op} \wedge v = s_{j-2})$
$E_1 \oplus E_2$	$\text{canBeAffected}(E_1, \text{ins}) \vee \text{canBeAffected}(E_2, \text{ins})$
$E.g$	$\text{canBeAffected}(E, \text{ins}) \vee (\text{ins} = \text{putfield } f \wedge f = g)$ $\vee (\text{ins} = \text{call } m \wedge \text{execution of } m \text{ might modify } g)$
$E_0.p(E_1, \dots, E_\pi)$	$\bigvee_{i=0}^{\pi} \text{canBeAffected}(E_i, \text{ins}) \vee (\text{ins} = \text{putfield } f \wedge f \in \text{flds}(E))$ $\vee (\text{ins} = \text{call } m \wedge \text{the execution of } m \text{ might modify a field in } \text{flds}(E))$

That is, instructions that remove some variables from the stack (`putfield`, `op`, `ifne`, `ifeq` and `store`) affect the evaluation of all the expressions in which these variables appear; instructions that write into one particular variable (`inc`, `store`, `getfield` and `op`) might affect the evaluation of the expressions containing that variable; `putfield`  $f$  might modify the evaluation of all the expressions that might read  $f$ ; `call`  $m_1 \dots m_k$  might modify the evaluation of all expressions that might read a field  $f$  possibly modified by an  $m_i$ .

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.

**Definition 9 (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*  $E$ :

- $\text{mightMdf}(n, F) = \text{mightMdf}(v, F) = \text{false}$ , for every  $n \in \mathbb{Z}$  and every  $v \in \text{dom}(\tau)$ ;
- $\text{mightMdf}(E_1 \oplus E_2, F) = \text{mightMdf}(E_1, F) \vee \text{mightMdf}(E_2, F)$ ;
- $\text{mightMdf}(E.g, F) = \text{mightMdf}(E, F)$ ;
- $\text{mightMdf}(E_0.p(E_1, \dots, E_\pi), F) = \text{true}$  if there exists  $0 \leq i \leq \pi$ , s.t.  $\text{mightMdf}(E_i, F) = \text{true}$  or if the execution of  $p$  might write a field from  $F$ .

## 4 Definite Expression Aliasing Analysis

The concrete semantics works over concrete states, that our abstract interpretation abstracts into tuples of sets of expressions.

**Definition 10 (Concrete and Abstract Domain).** *The concrete domain over*  $\tau \in \mathcal{T}$  *is*  $\mathbf{C}_\tau = \langle \wp(\Sigma_\tau), \subseteq, \cup, \cap \rangle$  *and the abstract domain over*  $\tau$  *is*  $\mathbf{A}_\tau = \langle (\wp(\mathbb{E}_\tau))^{|T|}, \sqsubseteq, \sqcup, \sqcap \rangle$ , *where for every*  $A^1 = \langle \mathbf{A}_0^1, \dots, \mathbf{A}_{|T|-1}^1 \rangle$  *and*  $A^2 = \langle \mathbf{A}_0^2, \dots, \mathbf{A}_{|T|-1}^2 \rangle$ ,  $A^1 \sqsubseteq A^2$  *if and only if for each*  $0 \leq i < |T|$ ,  $\mathbf{A}_i^1 \supseteq \mathbf{A}_i^2$ . *Moreover, the join operator*  $\sqcup$  *is defined as*  $A^1 \sqcup A^2 = \langle \mathbf{A}_0^1 \cap \mathbf{A}_0^2, \dots, \mathbf{A}_{|T|-1}^1 \cap \mathbf{A}_{|T|-1}^2 \rangle$ . *The meet operator*  $\sqcap$  *is dually defined.*

Concrete states  $\sigma$  corresponding to an abstract element  $\langle \mathbf{A}_0, \dots, \mathbf{A}_{|T|-1} \rangle$  must satisfy the aliasing information represented by the latter, i.e., for each  $0 \leq r < |T|$ , the value of all the expressions from  $\mathbf{A}_r$  in  $\sigma$  must coincide with the value of  $v_r$  in  $\sigma$  (*definite aliasing*).

**Definition 11 (Concretization map).** *Let*  $\tau \in \mathcal{T}$  *and*  $A = \langle \mathbf{A}_0, \dots, \mathbf{A}_{|T|-1} \rangle \in \mathbf{A}_\tau$ . *We define*  $\gamma_\tau : \mathbf{A}_\tau \rightarrow \mathbf{C}_\tau$  *as follows:*  $\gamma(A) = \{ \sigma = \langle \rho, \mu \rangle \in \Sigma_\tau \mid \forall 0 \leq r < |T|. \forall E \in \mathbf{A}_r. \llbracket E \rrbracket \sigma = \rho(v_r) \}$ .

Both  $\mathbf{C}_\tau$  and  $\mathbf{A}_\tau$  are complete lattices. Moreover, we proved  $\gamma_\tau$  co-additive, and therefore it is the concretization map of a Galois connection [5] and  $\mathbf{A}_\tau$  is actually an abstract domain, in the sense of abstract interpretation.

### 4.1 The Abstract Constraint Graph

Our analysis is constraint-based: we construct an *abstract constraint graph* from the program under analysis and then we solve these constraints. For each bytecode of the program there is a node containing an approximation of the actual aliasing information at that point. Arcs of the graph propagate these approximations, reflecting, in abstract terms, the effects of the concrete semantics on the aliasing information. In other words, an arc between the nodes corresponding to two bytecodes  $b_1$  and  $b_2$  propagates the aliasing information at  $b_1$  into that at  $b_2$ . The exact meaning of *propagates* depends here on  $b_1$ , since each bytecode has different effects on the abstract information.



	ins	$A'_r$
#1	dup t	$A'_r = \begin{cases} A_r \cup A_r[s_j/s_{j-1}] & \text{if } r <  \tau -1 \\ A_{ \tau -1} \cup \{s_j\} & \text{if } r =  \tau -1 \\ A_{ \tau -1} \cup \{s_{j-1}\} & \text{if } r =  \tau  \end{cases}$
#2	new $\kappa$	$A'_r = \begin{cases} A_r & \text{if } r \neq  \tau  \\ \emptyset & \text{if } r =  \tau  \end{cases}$
#3	load $k$ t	$A'_r = \begin{cases} A_r \cup A_r[s_j/l_k] & \text{if } r \notin \{k,  \tau \} \\ A_k \cup \{s_j\} & \text{if } r = k \\ A_k \cup \{l_k\} & \text{if } r =  \tau  \end{cases}$
#4	store $k$ t	$A'_r = \begin{cases} \{E \in A_r \mid \neg \text{canBeAffected}(E, \text{ins})\} & \text{if } r \neq k \\ \{E \in A_{ \tau -1} \mid \neg \text{canBeAffected}(E, \text{ins})\} & \text{if } r = k \end{cases}$
#5	getfield $f$	$A'_r = \begin{cases} \{E \in A_r \mid \neg \text{canBeAffected}(E, \text{ins})\} & \text{if } r \neq  \tau -1 \\ \{E.f \mid E \in A_{ \tau -1} \wedge \neg \text{canBeAffected}(E, \text{ins}) \\ \wedge \neg \text{mightMdf}(E, \{f\})\} & \text{if } r =  \tau -1 \end{cases}$
#6	putfield $f$	$A'_r = \{E \in A_r \mid \neg \text{canBeAffected}(E, \text{ins})\}$
#7	catch, ifne t, ifeq t	
#8	const $v$	$A'_r = \begin{cases} A_r & \text{if } r \neq  \tau  \\ \{x\} & \text{if } r =  \tau  \end{cases}$
#9	inc $k$ $x$	$A'_r = \begin{cases} \{E[l_k - x/l_k] \mid E \in A_r\} & \text{if } r \neq k \\ \emptyset & \text{if } r = k \end{cases}$
#10	op	$A'_r = \begin{cases} \{E \in A_r \mid \neg \text{canBeAffected}(E, \text{ins})\} & \text{if } r \neq  \tau -2 \\ \{E_1 \oplus E_2 \mid E_1 \in A_{ \tau -2} \wedge E_2 \in A_{ \tau -1} \\ \wedge \neg \text{canBeAffected}(E_1 \oplus E_2, \text{ins})\} & \text{if } r =  \tau -2 \end{cases}$
#11	return void	$A'_r = \{E \in A_r \mid \text{noStackElements}(E)\}$
#12	return t	$A'_r = \begin{cases} \{E \in A_r \mid \text{noStackElements}(E)\} & \text{if } r \neq i \\ \{E \in A_{ \tau -1} \mid \text{noStackElements}(E)\} & \text{if } r = i \end{cases}$
#13	throw $\kappa$	$A'_r = \begin{cases} \{E \in A_r \mid \text{noStackElements}(E)\} & \text{if } r \neq i \\ \emptyset & \text{if } r = i \end{cases}$
#14	call $m_1 \dots m_k$	
#15	new $\kappa$ , throw $\kappa$ getfield $f$ , putfield $f$	

Fig. 4. Propagation rules of 1–1 arcs

**Definition 12 (ACG).** Let  $P$  be the program under analysis, already in the form of a CFG of basic blocks for each method or constructor (Section 2). The abstract constraint graph (ACG) for  $P$  is a directed graph  $\langle V, E \rangle$  (nodes, arcs) where:

- $V$  contains a node  $\boxed{\text{ins}}$  for each bytecode ins in  $P$ ;
- for each method or constructor  $m$  in  $P$ ,  $V$  contains nodes  $\boxed{\text{exit}@m}$  and  $\boxed{\text{exception}@m}$ , representing the normal and the exceptional final states of  $m$ ;
- each node contains an abstract element  $A \in \mathbf{A}$  representing an approximation of the actual aliasing information at that point;
- $E$  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  $\mathbf{A}$ , from the aliasing information at its source(s) to the aliasing information at its sink.

The arcs in  $E$  are built from  $P$  as follows. We assume for all 1–1 arcs that  $\tau$  and  $\tau'$  are the static type information at and immediately after the execution of a bytecode

	$A'_r$	
#17	$A'_r =$	$\begin{cases} A_r & \text{if } r \neq  \tau_C  - \pi \\ \{E = R[E_0, \dots, E_{\pi-1}/l_0, \dots, l_{\pi-1}] \mid R \in R_{ \tau_E -1} \wedge \text{safeReturn}(R, m_w) \wedge \text{safeAlias}(E, \text{ins}_C)\} \\ \cup \{E = E_0.m(E_1, \dots, E_{\pi-1}) \mid \text{safeAlias}(E, \text{ins}_C)\} & \text{if } r =  \tau_C  - \pi \end{cases}$
#18	$A'_r =$	$\begin{cases} \{E \mid \text{safeExecution}(E, A_r, \text{ins}_C)\} & \text{if } r \neq  \tau_C  - \pi \\ \mathbb{E}_{\tau_N} & \text{if } r =  \tau_C  - \pi \end{cases}$
		$\text{safeExecution}(E, A, \text{ins}_C) = E \in A \wedge \text{noParameters}(E) \wedge \neg \text{canBeAffected}(E, \text{ins}_C)$ $\text{safeAlias}(E, \text{ins}_C) = \bigwedge_{k=0}^{\pi-1} \text{safeExecution}(E_k, A_{ \tau_C -\pi+k}, \text{ins}_C) \wedge \neg \text{mightMdf}(E, \text{flds}(E))$ $\text{safeReturn}(R, m_w) = \forall l_k \in \text{vars}(R) \subseteq \{l_0, \dots, l_{\pi-1}\}. l_k \text{ is not modified by } m_w$ $\text{noParameters}(E) = \text{vars}(E) \cap \{v_{ \tau_C -\pi}, \dots, v_{ \tau_C -1}\} = \emptyset$

Fig. 5. Propagation rules of 2–1 arcs

ins, respectively. Moreover, we assume that  $\tau$  contains  $j$  stack elements and  $i$  local variables and, for every expression  $E$ , we write  $\text{noStackElements}(E)$  to denote that no stack element appears in  $E$ , i.e.,  $\text{vars}(E) \cap \{s_0, \dots, s_{j-1}\} = \emptyset$ .

**Sequential arcs.** If ins is a bytecode in  $P$ , distinct from call, immediately followed by a bytecode ins', distinct from catch, then an 1–1 arc is built from  $\boxed{\text{ins}}$  to  $\boxed{\text{ins}'}$ , with a propagation rule  $\lambda \langle A_0, \dots, A_{|\tau|-1} \rangle. \langle A'_0, \dots, A'_{|\tau|-1} \rangle$  where, for each  $0 \leq r < |\tau|$ ,  $A'_r$  is defined by one of the rules #1 – #10 in Fig. 4.

**Final arcs.** For each return  $t$  and throw  $\kappa$  occurring in a method or constructor  $m$  of  $P$ , there are 1–1 arcs from  $\boxed{\text{return } t}$  to  $\boxed{\text{exit}@m}$  and from  $\boxed{\text{throw } \kappa}$  to  $\boxed{\text{exception}@m}$ , respectively, with a propagation rule  $\lambda \langle A_0, \dots, A_{|\tau|-1} \rangle. \langle A'_0, \dots, A'_{|\tau|-1} \rangle$  where, for each  $0 \leq r < |\tau|$ ,  $A'_r$  is defined by one of the rules #11 – #13 in Fig. 4.

**Exceptional arcs.** For each ins throwing an exception, immediately followed by a catch, an arc is built from  $\boxed{\text{ins}}$  to  $\boxed{\text{catch}}$  with a prop. rule  $\lambda \langle A_0, \dots, A_{|\tau|-1} \rangle. \langle A'_0, \dots, A'_{|\tau|-1} \rangle$  where, for each  $0 \leq r < |\tau|$ ,  $A'_r$  is defined by rules #14 or #15 in Fig. 4.

**Parameter passing arcs.** For each call  $m_1 \dots m_q$  occurring in  $P$  with  $\pi$  parameters (including the implicit parameter this), for each  $1 \leq i \leq q$  we build an 1–1 arc from  $\boxed{\text{call } m_1 \dots m_q}$  to the node corresponding to the first bytecode of  $m_i$ , with a propagation rule #16:  $\lambda \langle A_0, \dots, A_{|\tau|-1} \rangle. \langle A'_0, \dots, A'_{\pi-1} \rangle$  where, for each  $0 \leq r < \pi$ ,  $A'_r = \emptyset$ .

**Return value arcs.** For each  $\text{ins}_C = \text{call } m_1 \dots m_q$  to a method with  $\pi$  parameters (including the implicit parameter this) returning a value of type  $t \neq \text{void}$ , and each subsequent bytecode  $\text{ins}_N$  distinct from catch, we build, for each  $1 \leq w \leq q$ , a 2–1 arc from  $\boxed{\text{ins}_C}$  and  $\boxed{\text{exit}@m_w}$  (2 sources, in that order) to  $\boxed{\text{ins}_N}$ . Suppose that the static type information at  $\boxed{\text{ins}_C}$ ,  $\boxed{\text{exit}@m_w}$  and  $\boxed{\text{ins}_N}$  are  $\tau_C$ ,  $\tau_E$  and  $\tau_N$ , respectively. We define a propagation rule  $\lambda \langle A_0, \dots, A_p, \dots, A_{|\tau_C|-1} \rangle. \langle R_0, \dots, R_{|\tau_E|-1} \rangle. \langle A'_0, \dots, A'_{|\tau_C|-\pi} \rangle$ , where for each  $0 \leq r \leq |\tau_C| - \pi$ ,  $A'_r$  is defined by the rule #17 in Fig. 5.

**Side-effects arcs.** For each  $\text{ins}_C = \text{call } m_1 \dots m_q$  to a method with  $\pi$  parameters (including the implicit parameter this), and each subsequent bytecode  $\text{ins}_N$ , we build, for each  $1 \leq w \leq q$ , a 2–1 arc from  $\boxed{\text{ins}_C}$  and  $\boxed{\text{exit}@m_w}$  (2 sources, in that order) to  $\boxed{\text{ins}_N}$ , if  $\text{ins}_N$  is not a catch and a 2–1 arc from  $\boxed{\text{ins}_C}$  and  $\boxed{\text{exception}@m_w}$  (2 sources, in that order) to  $\boxed{\text{catch}}$ . Suppose that the static type information at  $\boxed{\text{ins}_C}$ ,  $\boxed{\text{exit}@m_w}$  (or  $\boxed{\text{exception}@m_w}$ ) and  $\boxed{\text{ins}_N}$  are  $\tau_C$ ,  $\tau_E$  and  $\tau_N$  respectively. We define a propagation rule

$\lambda\langle \mathbf{A}_0, \dots, \mathbf{A}_{|\tau_C|-\pi}, \dots, \mathbf{A}_{|\tau_C|-1} \rangle, \langle \mathbf{R}_0, \dots, \mathbf{R}_{|\tau_E|-1} \rangle, \langle \mathbf{A}'_0, \dots, \mathbf{A}'_{|\tau_N|-1} \rangle$ , where for each  $0 \leq r < |\tau_N|$ ,  $\mathbf{A}'_r$  is defined by the rule #18 in Fig. 5.

*Example 3.* In Fig. 6 we give the ACG of the method `delayMinBy` from 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. Arcs are decorated with the number of their associated propagation rules. In the following examples, for each node  $x$ , we let  $A^x = \langle \mathbf{A}_0^x, \dots, \mathbf{A}_{n_x}^x \rangle$  be the aliasing information at  $x$ , where  $n_x$  is the number of variables at  $x$  and, for each  $r$ , we let  $\mathbf{A}_r^x$  be an approximation of the definite aliasing expressions of variable  $v_r$ . ■

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  $\neg \text{canBeAffected}(E, \text{ins})$  holds. However, some new alias expressions might be added to the initial approximation as well. For instance, in the case of `ins = dup t` (rule #1), the new added variable  $v_n = s_j$  is a copy of  $v_{n-1} = s_{j-1}$ , hence they are trivially aliased to each other, and all definite alias expressions of  $v_{n-1}$  at `ins` become definite alias expressions of  $v_n$  at `ins'` (i.e.,  $\mathbf{A}'_{n-1} = \mathbf{A}_{n-1} \cup \{s_j\}$ ,  $\mathbf{A}'_n = \mathbf{A}_{n-1} \cup \{s_{j-1}\}$ ). Approximations related to the rest of the variables are enriched with the same expressions where occurrences of  $s_{j-1}$  are replaced by  $s_j$  ( $\mathbf{A}'_r = \mathbf{A}_r \cup \mathbf{A}_r[s_j/s_{j-1}]$ ). Rule #5 is more interesting: `ins = getField f` inserts an expression  $E.f$  among alias expressions of  $v_{n-1}$  at `ins'` if  $E$  is aliased to  $v_{n-1}$  (holding the receiver) at `ins`, it cannot be modified by `ins` ( $\neg \text{canBeAffected}(E, \text{ins})$ ) and the evaluation of  $E$  cannot modify the field  $f$  ( $\neg \text{mightMdf}(E, \{f\})$ ). For instance, suppose that in Ex. 3 we have  $n_2 = 3$  and  $\mathbf{A}_2^2 = \{v_0\}$ , i.e.,  $v_2$ , the top of the stack and the receiver of the `getField` at **2**, is aliased to  $v_0$ . There is an arc with rule #5 connecting nodes **2** and **3**. According to that rule, since the `getField` cannot affect  $v_0$ , but only  $v_2$ , and since no evaluation of  $v_0$  can modify any field (in particular `min`), we conclude that  $\mathbf{A}_2^3 = \{v_0.\text{min}\}$ , i.e., the new top of the stack is aliased to the field `min` of the only alias of the old top of the stack.

The **final arcs** feed nodes `exit@m` and `exception@m` for each method  $m$ . They propagate, for each local variable  $l_k$  at `ins'`, all those expressions aliased to  $l_k$  at `ins` where no stack variable occurs. In the case of `ins = return t`, with  $t \neq \text{void}$ , the alias expressions of  $v_i = s_0$  at `ins'` are alias expressions of  $v_{n-1} = s_{j-1}$  at `ins` with no stack elements.

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

Let us explain the auxiliary functions introduced in Fig. 5. An *execution of* `ins = call m1 ... mt` is *safe for an expression*  $E \in \mathbf{A}$  ( $\text{safeExecution}(E, \mathbf{A}, \text{ins})$  holds), if the fields possibly read during the evaluation of  $E$  must not be modified by the invoked method ( $\neg \text{canBeAffected}(E, \text{ins})$ ) and if no actual parameter of that method appears in  $E$  ( $\text{noParameters}(E)$ ), since they disappear from the stack after the call. An *alias expression*  $E$  in which  $E_0, \dots, E_{\pi-1}$  appear is *safe* ( $\text{safeAlias}(E, \text{ins})$  holds) if `ins` is safe for each  $E_r$  and if no field might be both read and modified during all possible evaluations of  $E$  ( $\neg \text{mightMdf}(E, \text{flds}(E))$ ). An *alias expression*  $\mathbf{R}$  of a return value at the exit from a method  $m_w$  is *safe*, i.e.,  $\text{safeReturn}(\mathbf{R}, m_w)$  holds, if only local variables corresponding to the formal parameters of  $m_w$  ( $l_0, \dots, l_{\pi-1}$ ) appear in  $\mathbf{R}$  and none of them is modified by  $m_w$  (for every  $l_k \in \text{vars}(\mathbf{R})$ , no `store k t` nor `inc k x` occurs in  $m_w$ ).

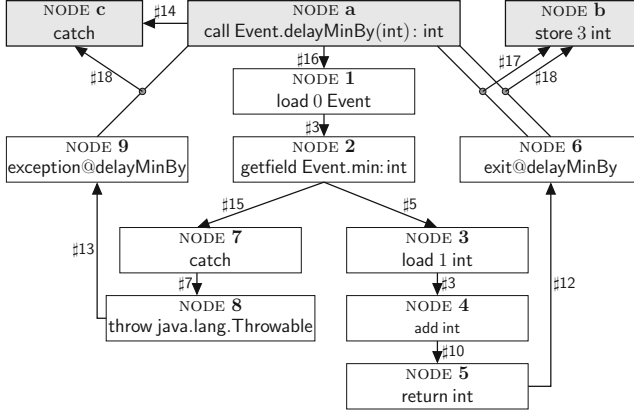


Fig. 6. The ACG for the method `delayMinBy` in Fig. 2

There exists a **return value 2–1 arc** for each target  $m_w$  of a call returning a value. Rule #17 considers  $\langle A_0, \dots, A_{|\tau_C|-1} \rangle$  and  $\langle R_0, \dots, R_{|\tau_E|-1} \rangle$ , approximations at  $\boxed{\text{ins}_C}$  and  $\boxed{\text{exit}@m_w}$ , and builds the alias expressions related to the returned value  $s_{|\tau_C|-\pi}$  at  $\boxed{\text{ins}_N}$ . An alias expression  $R \in R_{|\tau_E|-1}$  of the computed value  $s_0$  at  $\boxed{\text{exit}@m_w}$  can be turned into an alias expression of  $s_{|\tau_C|-\pi}$  at  $\boxed{\text{ins}_N}$  if (i)  $R$  is safe; (ii) every occurrence of a formal parameter  $l_k$  in  $R$  is replaced by an alias expression  $E_k \in A_{|\tau_C|-\pi+k}$  of the corresponding actual parameter  $s_{|\tau_C|-\pi+k}$  at  $\boxed{\text{ins}_C}$ , which is safe w.r.t.  $\text{ins}_C$ . Moreover,  $E = E_0.m_w(E_1, \dots, E_{\pi-1})$  can be an alias of  $s_{|\tau_C|-\pi}$  at  $\boxed{\text{ins}_N}$  if it is safe w.r.t.  $\text{ins}_C$ . For instance, suppose that in Ex. 3 the actual parameters of the call at node **a** (which become the local variables  $v_0$  and  $v_1$  inside `delayMinBy`) are aliased to  $v_1$  and 15, and that at the exit node **6** the return value is aliased to  $v_0.\text{min} + v_1$ . Since this expression is composed of local variables corresponding to the formal parameters of `delayMinBy` and the latter does not modify any variable (it contains no `store` nor `inc`), we conclude that  $v_0.\text{min} + v_1$  is safe.  $v_0$  and  $v_1$  at **6** correspond to the actual parameters at **a**, thus the aliases  $v_1$  and 15 of these latter can substitute these former obtaining the alias expression  $E = v_1.\text{min} + 15$  at **b**. Indeed, no evaluation of  $E$  can modify any field (Definition 9), so  $\neg\text{mightMdf}(E, \text{flds}(E))$  trivially holds. Moreover,  $v_1$  and 15 contain no actual parameter at **a**, and no execution of the method can modify a local variable or a constant of the caller, hence  $E$  is safe and can be an alias expression of the returned value at node **b**.

The **side-effects 2–1 arcs** consider the alias expressions  $E$  of the variables  $v_r$  different from the actual parameters ( $s_{|\tau_C|-\pi}, \dots, s_{|\tau_C|-1}$ ) of the method at  $\boxed{\text{ins}_C}$  and insert them among the alias expressions of  $v_r$  also at  $\boxed{\text{ins}_N}$  if they are safe w.r.t.  $\text{ins}_C$ .

**Definition 13 (Alias Expression Analysis).** A solution of an ACG is an assignment of an abstract element  $A_n$  to each node  $n$  of the ACG such that the propagation rules of the arcs are satisfied, i.e., for every arc from nodes  $n_1 \dots n_k$  to  $n$  with propagation rule  $\lambda A_1, \dots, \lambda A_k. \Pi(A_1, \dots, A_k)$ , the condition  $A_n \sqsubseteq \Pi(A_{n_1}, \dots, A_{n_k})$  holds. The alias expression analysis of the program is the minimal solution of its ACG w.r.t.  $\sqsubseteq$ .

Definition 13 entails 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

NODE $n$	$A_1^n$	$A_2^n$	$A_3^n$	$i_n$	$j_n$
<b>a</b>	$\emptyset$	$\{v_0.\text{getHead}(), s_0\}$	$\{v_0.\text{getHead}(), v_1\}$	$\{15\}$	2 2
<b>1</b>	$\emptyset$	$\emptyset$	–	–	2 0
<b>2</b>	$\{s_0\}$	$\emptyset$	$\{v_0\}$	–	2 1
<b>3</b>	$\emptyset$	$\emptyset$	$\{v_0.\text{min}\}$	–	
<b>4</b>	$\emptyset$	$\{s_1\}$	$\{v_0.\text{min}\}$	$\{v_1\}$	2 2
<b>5, 6</b>	$\emptyset$	$\emptyset$	$\{v_0.\text{min} + v_1\}$	–	2 1
<b>7, 8, 9</b>	$\emptyset$	$\emptyset$	$\emptyset$	–	
<b>b</b>	$\emptyset$	$\{v_0.\text{getHead}()\}$	$\{v_1.\text{delayMinBy}(15), v_0.\text{getHead}().\text{min}+15, v_1.\text{min}+15, v_0.\text{getHead}().\text{delayMinBy}(15)\}$	–	
<b>c</b>	$\emptyset$	$\{v_1.\text{getHead}()\}$	$\emptyset$	–	

Fig. 7. The solution of the ACG from Fig. 6

the actual aliasing information for each variable  $v$  at  $n$  is the intersection of the  $k$  sets related to  $v$ . The minimal solution w.r.t.  $\sqsubseteq$  corresponds to the greatest sets of expressions for each variable (Definition 10). In order to guarantee its existence, we fix an upper bounds on the height of the alias expressions (e.g., a maximal number of field accesses and method invocations) which makes the abstract domain  $A_\tau$  finite. The solution of the constraint can, hence, be computed by starting with the bottom approximation for every node: the set of all possible alias expressions; and then applying the propagation of the arcs and computing the intersection at each node entry, until stabilization.

*Example 4.* Fig. 7 shows the solution of the ACG from Fig. 6. For each node  $n$ , the values shown in columns  $i_n$ ,  $j_n$  and  $A_\tau^n$  are respectively the number of local variables, stack elements and the final approximation of the aliasing information related to the variable  $v_r$  at that point. When the latter is –, it means that  $v_r$  is not available there. It is worth noting that the variable  $v_0$  at nodes **a**, **b** and **c** is of type `ListEvents` (Ex. 2) and that `getHead` only returns the field `head` of that class. ■

The following theorem states that our analysis is sound.

**Theorem 1 (Soundness).** *Suppose that an execution of a program leads to a state  $\sigma \in \Sigma_\tau$ . Let  $A^{\text{ins}} \in A_\tau$  be the approximation of the definite expression aliasing information at the node **ins** corresponding to `ins`, computed by our static analysis. Then,  $\sigma \in \gamma_\tau(A^{\text{ins}})$ .*

## 5 Experiments

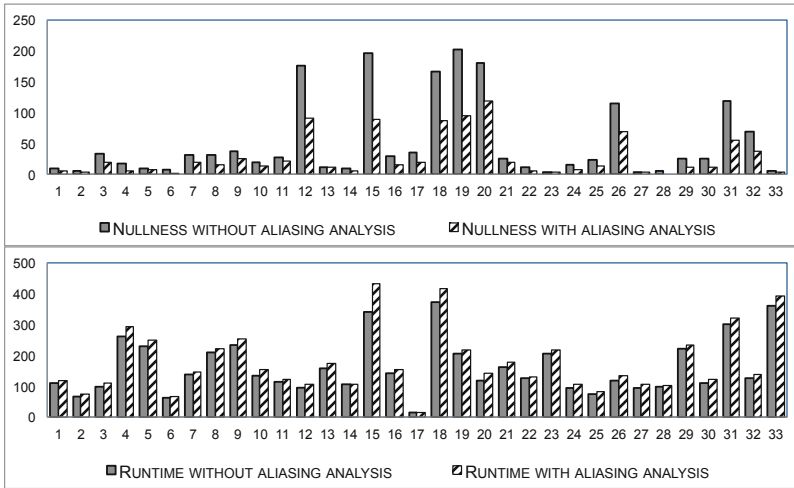
We have implemented our definite expression aliasing analysis inside the Julia analyzer for Java bytecode (<http://www.juliasoft.com>) and we have analyzed some real-life benchmark programs. We provide the names of these latter together with their identification numbers used in Fig. 8 and 9. The majority of our benchmarks are Android applications: Mileage (15), OpenSudoku (19), Solitaire (26) and TiltMazes<sup>1</sup> (29); Chime-Timer (4), Dazzle (7), OnWatch (18) and Tricorder<sup>2</sup> (31); TxWthr<sup>3</sup> (32). There are also some Java programs: JFlex (12) is a lexical analyzers generator<sup>4</sup>; Plume is a library by

<sup>1</sup> <http://f-droid.org/repository/browse/>

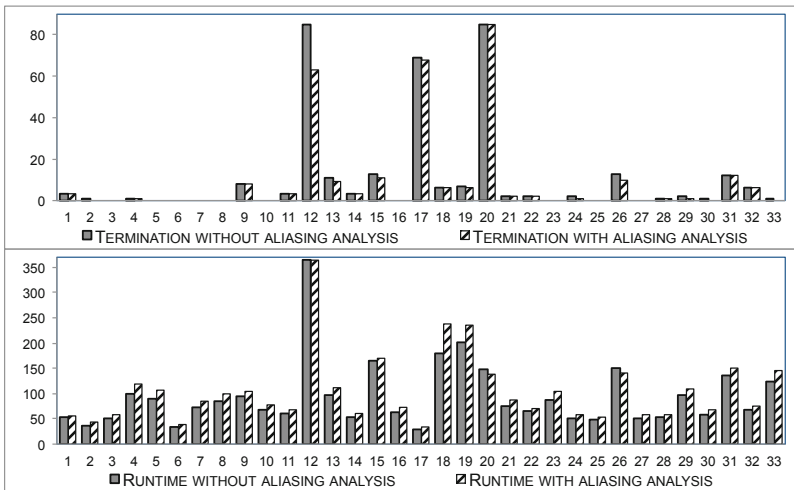
<sup>2</sup> <http://moonblink.googlecode.com/svn/trunk/>

<sup>3</sup> <http://typoweather.googlecode.com/svn/trunk/>

<sup>4</sup> <http://jflex.de>



**Fig. 8.** Comparison of the number of warnings (possible dereference of null, possibly passing null to a library method) produced by the nullness tool of Julia (top) and of the run-times (in seconds) of that tool (bottom) when our definite aliasing analysis is present and absent



**Fig. 9.** Comparison between number of warnings (possible divergence of constructors or methods) produced by the termination tool of Julia (top) and between run-times (in seconds) of that tool (bottom) when our definite aliasing analysis is present and absent

Michael D. Ernst<sup>5</sup>; Nti (17) is a non-termination analyzer by Étienne Payet<sup>6</sup>; Lisimplex (13) is a numerical simplex implementation by Ricardo Gobbo<sup>7</sup>. The others are sample programs from the Android 3.1 distribution by Google.

<sup>5</sup> <http://code.google.com/p/plume-lib>

<sup>6</sup> <http://personnel.univ-reunion.fr/epayet/Research/NTI/NTI.html>

<sup>7</sup> <http://sourceforge.net/projects/lisimplex>

Definite expression aliasing analysis is used 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` (resp. `x`) are upper (resp. lower) bounds for `x` (resp. `y`). This is important for termination analysis.

Figures 8 and 9 report the precision and the run-time of our nullness and termination analyses on a Linux quad-core Intel Xeon machine running at 2.66GHz, with 8 gigabytes of RAM. We performed these analyses first without and then with the help of our definite expression aliasing analysis. This way, we notice how the tools’ precision changes. A clear difference between the two runs is that the run-time of the nullness and termination analyses increased by 9.88% and 12.57% respectively, when the definite expression aliasing analysis is activated. On the other hand, the precision of both analyses is improved in the presence of the definite expression aliasing analysis: 45.98% and 11.44% less warnings are produced by the nullness and termination analyses, respectively. These improvements are well worth the extra time required for the analyses.

## 6 Conclusion

Our expression aliasing analysis is a *constraint-based definite* analysis for Java bytecode. To the best of our knowledge, it is the first definite aliasing analysis dealing with Java bytecode programs and with aliases to expressions. Our experimental evaluation shows the benefits of our new analysis for the nullness and termination analyses of Julia.

## 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. Albert, E., Arenas, P., Genaim, S., Puebla, G., Ramírez Deantes, D.V.: From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 100–116. Springer, Heidelberg (2010)
5. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of the 4th Symp. on Principles of Programming Languages (POPL), pp. 238–252. ACM (1977)
6. Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective Typestate Verification in the Presence of Aliasing. In: Proc. of the International Symposium on Software Testing and Analysis (ISSTA), pp. 133–144. ACM (2006)
7. Hind, M.: Pointer Analysis: Haven’t We Solved This Problem Yet? In: Proc. of the Workshop on Prog. Analysis for Software Tools and Engineering (PASTE), pp. 54–61. ACM (2001)
8. Lindholm, T., Yellin, F.: The Java<sup>TM</sup> Virtual Machine Specification, 2nd edn. Addison-Wesley (1999)

9. Logozzo, F., Fähndrich, M.: On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 197–212. Springer, Heidelberg (2008)
10. Nikolić, Đ., Spoto, F.: Definite Expression Aliasing Analysis for Java Bytecode, <http://profs.sci.univr.it/~nikolic/download/ICTAC2012/ICTAC2012Ext.pdf>
11. Nikolić, Đ., Spoto, F.: Reachability Analysis of Program Variables. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 423–438. Springer, Heidelberg (2012)
12. Ohata, F., Inoue, K.: JAAT: Java Alias Analysis Tool for Program Maintenance Activities. In: Proc. of the 9th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), pp. 232–244. IEEE (2006)
13. Spoto, F.: Precise Null-pointer Analysis. *Software and Syst. Modeling* 10(2), 219–252 (2011)
14. Spoto, F., Ernst, M.D.: Inference of Field Initialization. In: Proc. of the 33rd International Conference on Software Engineering (ICSE), pp. 231–240. ACM (2011)