Theoretical Computer Science 484 (2013) 16–40

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Inferring complete initialization of arrays

Đurica Nikolić [a,b,*], Fausto Spoto [a]

[a] *Dipartimento di Informatica, University of Verona, Strada Le Grazie 15, 37134 Verona, Italy*

[b] *Microsoft Research–University of Trento Centre for Computational and Systems Biology, Piazza Manifattura 1, 38068 Rovereto, Italy*

ARTICLE INFO

Article history:
Received 22 February 2012
Received in revised form 28 October 2012
Accepted 4 January 2013
Communicated by D. Sannella

Keywords:
Static analysis
Array initialization
Abstract interpretation
Finite-state automata

ABSTRACT

We define an automaton-based abstract interpretation of a trace semantics which identifies loops that definitely initialize *all* elements of an array to values satisfying a given property, a useful piece of information for the static analysis of Java-like languages. This results in a completely automatic and efficient analysis, that does not use manual code annotations. We give a formal proof of correctness that considers aspects such as side-effects of method calls. We show how the identification of those loops can be lifted to global invariants about the contents of elements of fields of array type, that hold everywhere in the code where those elements are accessed. This makes our work more significant and useful for the static analysis of real programs. The implementation of our analysis inside the Julia analyzer is both efficient and precise.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

Static analysis of Java-like languages is very often concerned with the identification of properties that hold for program variables at some specific program points. By *program variables* we mean both local variables (to a method) or *fields*, that is, variables local to an object. Identifying such properties, in the second case, is extremely more complex, since fields might well not be initialized before being accessed, in which case they hold a default or unspecified value, depending on the semantics of the language. Hence, for fields, it is necessary to be able to prove that fields are never read before being written at least once; only in that case it is possible to guarantee that, wherever they are read, they contain a value that was written before into them. Moreover, fields can be accessed through distinct expressions, aliased to the same object, which further complicates the static analysis process.

Things become yet more complex when variables hold arrays rather than a single value. For those variables, most sound static analyses just assume the worst: the elements of the arrays that they hold are approximated by a worst-case assumption. This results in many spurious warnings when it comes to verification techniques based on those static analyses. For instance, this work was born from a concrete problem faced during the static analysis of Java and Android programs. We want to analyze the `CubeWallpaper` Android program distributed by Google with the Android Software Development Kit and prove that it can never throw a null pointer exception at run-time. The interesting snippet of code is shown in Fig. 1. Fields `mOriginalPoints` and `mRotatedPoints` hold arrays, initialized by `readModel()` and later read and dereferenced in other methods. There, the null pointer analysis of Julia [24] issued spurious warnings since it used a worst-case assumption for the contents of the elements of `mOriginalPoints` and `mRotatedPoints`, i.e., such elements were assumed to be potentially `null`. We hence realized that we had to improve the analyzer with a technique able to reason about arrays and cope with those false alarms.

* Corresponding author at: Dipartimento di Informatica, University of Verona, Strada Le Grazie 15, 37134 Verona, Italy. Tel.: +39 3929142461.
E-mail addresses: nikolic.durica@gmail.com, durica.nikolic@univr.it (Đ. Nikolić), fausto.spoto@univr.it (F. Spoto).

0304-3975/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2013.01.007

```
private void readModel(String prefix) {
  ....
  String[] p = ....
  int numpoints = p.length;
  this.mOriginalPoints = new ThreeDPoint[numpoints];
  this.mRotatedPoints = new ThreeDPoint[numpoints];

  for (int i = 0; i < numpoints; i++) {
    this.mOriginalPoints[i] = new ThreeDPoint();
    this.mRotatedPoints[i] = new ThreeDPoint();
    String[] coord = p[i].split(" ");
    this.mOriginalPoints[i].x = Float.valueOf(coord[0]);
    this.mOriginalPoints[i].y = Float.valueOf(coord[1]);
    this.mOriginalPoints[i].z = Float.valueOf(coord[2]);
  }
  [point *]
  ....
}
```

Fig. 1. A snippet of code from the `CubeWallpaper` Android program by Google.

Namely, we wanted to prove automatically the following property that we call `global` in the rest of the paper:

> "*all elements of fields mOriginalPoints and mRotatedPoints are non-null wherever they are read*".

When we say *wherever*, we mean that this property should also hold in other methods, outside of `readModel()`. The first sensible goal in that direction is to prove the following property that we call `local` in the rest of the paper:

> "*all elements of fields mOriginalPoints and mRotatedPoints are completely initialized to non-null values at point **".

In our example, information *at point ** is of little interest, since arrays `mOriginalPoints` and `mRotatedPoints` are largely accessed *outside* of the method `readModel()`. What we actually need is the ability to use the local property `local` to prove the global property `global`, that also holds outside of `readModel()`. This lifting is far from obvious: those arrays might be modified in other program points, or their elements might be read and dereferenced before `readModel()` is executed. In the latter case, the default value `null` (or an unspecified value) would be found inside of them. Moreover, arrays might be modified by side-effects rather than by direct access. To the best of our knowledge, this lifting from local to global properties of arrays has never been studied before.

Related work

Automatic reasoning about properties of arrays has been addressed recently. Some of the approaches show that an array satisfies a certain property (specified by the user), while others discover valid properties of that array. A *decision procedure for satisfiability* in an expressive fragment of a theory of arrays is studied in [5]: the authors state that their main theory of arrays is motivated by practical requirements in software verification. They use Presburger arithmetic for the theory of indices and the theory of integers, reals and equalities to describe the contents of arrays. Another *decidable logic for reasoning about infinite arrays of integers* was introduced in [12]. This logic permits to express both: constraints on consecutive elements of arrays (e.g., an array is ascending) and periodic facts about elements of array (e.g., elements corresponding to even indices are equal to 0). Although a complete initialization of arrays is in general undecidable, there exist some approaches able to prove complete initialization in some particular cases. Most of them [9,10,3] are based on predicate abstraction [11], a special form of abstract interpretation, or on theorem proving [15,14]. [9] introduces a method for *automatically inferring loop invariants* and it is based on *predicate abstraction*, where the abstract domain is constructed using a given set of predicates. Predicates may be generated in a heuristic manner from the program text or by the programmer. Given a suitable set of predicates for a given loop, their algorithm infers universally-quantified loop invariants expressed as Boolean combinations of the predicates, and these invariants are crucial for verifying programs that manipulate unbounded data such as arrays. In [4] properties about arrays are analyzed using two opposite approaches: *array expansion* and *array smashing*. The former introduces an abstract element for each index in the array, it is precise, but in practice can only be used for arrays of small size, and even with up-to-date hardware cannot analyze unbounded arrays. The latter represents all array elements with one summary variable, allows one to handle arbitrary arrays efficiently but suffers from precision losses. A combination of the benefits of both these approaches is introduced in [10], where an abstract interpretation-based framework capable of capturing numeric properties of array elements is presented. There, the analysis attempts to partition array elements

into groups; assertions can be established and maintained about groups; in the case of array initialization, the analysis keeps, in distinct groups, array elements that were already initialized from uninitialized array elements. A similar idea is used in [13], where the authors restrict themselves to one-dimensional arrays and programs that manipulate arrays only through `for` loops that increment their index at each iteration and access arrays by simple expressions of the index. Moreover, they consider only a restricted class of all possible properties, and they state that in these settings their method detects interesting properties of non-trivial programs. All previous approaches can be applied on small target languages representing a restriction of real-life programming languages. They are intra-procedural and do not deal with side-effects of different method invocation, as we do instead.

The most detailed actual approach for discovering array properties is presented in [8], where `FunArray`, a parametric segmentation abstract domain functor for the fully automatic and scalable analysis of array content properties, is introduced. `FunArray` lifts existing analyses for scalar values to uniform compound data structures as arrays or collections. The authors implemented it into `Clousot`, an abstract interpretation-based static contract checker for `.NET`, and empirically validated the precision and the performance of the analysis by running it on the main libraries of `.NET` and on their own code. They could infer thousands of non-trivial invariants and deal with complex upper bound expressions for the loop variables. They state that theirs is the first analysis of this kind applied to such a large code base, and proven to scale. The technique introduced in [8] is able to prove our `local` property, but it looks as an overkill for proving a relatively simple property such as `local` and it does not explain how `local` can be lifted to `global`, in an automatic way. Our goal is a simpler analysis, both in implementational and theoretical terms. We also provide a formal proof of correctness, currently not available for [8].

This article discusses the array initialization algorithm for a simple imperative language, in order to keep definitions and proofs as simple as possible and concentrate on the relevant aspects of the analysis. However, we actually analyze full Java bytecode with the Julia analyzer, where aspects such as late binding of method calls and object-orientation are relevant for the definition of the analysis. The interested reader can find more details about those aspects of Julia in [26,25].

Where does our analysis fit inside julia?

Our analysis is initially intraprocedural, in the sense that loops that completely initialize all elements of an array are identified only if they lay, syntactically, inside a given method or constructor. Nevertheless, it uses some interprocedural information, related to the heap memory, such as reachability between program variables of reference type, purity of methods, creation point analysis and available expressions. This information is computed *before* our array initialization analysis, through independent analyses, hence not in a reduced product. The exact description of these supporting analyses is not the topic of this paper; they have been defined and described in other articles, already published or submitted for publication [20,17,19,16]. We only assume that this information is available and can be used by our array initialization analysis. Moreover, our base analysis is lifted to interprocedural properties of fields of array properties, through an oracle technique that we describe later.

The results of our array initialization analysis are subsequently useful to other static analyses, that are performed *after* it. In particular, in this article, we will show how those results are useful for nullness analysis, through the experiments that we perform in Section 8. That analysis is run independently from the array initialization analysis and exploits its results. Also in this case, we do not perform a reduced product of analyses but rather a sequence of analyses, one exploiting the results of the previous ones.

Java programs typically start from one or more main methods. However, libraries are called through many entry points, in any order. In the middle, there are frameworks such as Android, that are event-driven and hence have no actual main method, but rather event handling methods triggered by external events. Julia deals with all such situations, by assuming that entry points can be called with any possible input: their parameters might hold `null` and be shared in any way. In the case of Android, entry points are identified by looking at methods overriding prototype methods in the Android library (see [22] for more information).

2. Contributions and plan of the paper

The contributions of this article are:

1. a new abstract interpretation of traces of execution, proving that all elements of an array are definitely initialized at a given program point;
2. a detailed proof of correctness for the previous analysis;
3. a technique for lifting the results of the previous analysis to global invariants about fields of array type;
4. experiments showing that the analysis is efficient (one/two seconds for large software) and useful in practice.

Our abstract interpretation uses an automaton, whose states abstract properties of traces of execution. For each array variable a and index integer variables $i_1 \dots, i_n$, the automaton can be *executed* over the program to check if the array in a is completely initialized by using those index variables. In particular, each program point is decorated with a set of states of the automaton and all the traces leading to that program point must be among those represented by those states. If that set is actually the singleton {ACCEPT}, then all the traces reaching the program point definitely initialize all the elements of a with a (possibly nested) loop using the given index variables. This algorithm is run for every possible choice of a and

```
public Random(int Seed) {
 Contract.Requires(Seed != Int32.MinValue);
 int num2 = 161803398 - Math.Abs(Seed);
 this.SeedArray = new int[56];
 this.SeedArray[55] = num2;
 int num3 = 1;
 // Loop 1
 for (int i = 1; i < 55; i++) {
  int index = (21 * i) % 55;
  this.SeedArray[index] = num3;  // (*)
  num3 = num2 - num3;
  if (num3 < 0)  num3 += 2147483647;
  num2 = this.SeedArray[index];
 }
 Contract.Assert(Contract.Forall( // (**)
   0,this.SeedArray.Length - 1, i => a[i] >= -1));
 // Loop 2
 for (int j = 1; j < 5; j++) {
  // Loop 3
  for (int k = 1; k < 56; k++) {
   this.SeedArray[k] -= this.SeedArray[1 + (k + 30) % 55];
   if (this.SeedArray[k] < 0)
    this.SeedArray[k] += 2147483647;
  } }
 Contract.Assert(Contract.Forall(0,  // (***)
   this.SeedArray.Length, i => a[i] >= -1));
}
```

Fig. 2. Example extracted from [8].

$i_1, \ldots, i_n$, which in practice are very few. If a is actually a field of an object, initialized in the constructor of the object, we are also able to lift this *local* complete initialization of a to its complete initialization at *every* program point where an element of a is accessed, a much more useful piece of information.

We observe that our abstract domain is very simple, which gives rise to a fully detailed proof of soundness of our analysis for our target language. This proof is of low complexity, which is not always the case for analyses of arrays.

Our goal was to devise a simple static analysis that copes with the most frequent array initialization loops. Hence, our analysis is in general less precise than that in [8]. An example is shown in Fig. 2 (extracted from [8, Figure 1]): we are not able to cope with out-of-order array assignments. However, we cover the most realistic cases, as shown with the experimental evaluation of Section 8: our analysis spots most array initialization loops; we have also checked when our analysis fails in those experiments: this is always due to weaknesses in the underlying supporting alias analyses or failures in the detection of the expression holding the size of the array, rather than to weaknesses in our array initialization analysis. Other techniques based on static analysis would suffer from the same imprecisions as well.

A preliminary version of this paper appeared in [18], where only mono-dimensional arrays were considered. Moreover, contributions 2 and 3 above are not considered in [18] and discussions are much more condensed there.

The rest of the paper is organized as follows. Section 3 presents the syntax and semantics of a simple but representative imperative programming language with arrays. Section 4 presents our abstract interpretation, limited to arrays held in local variables. Section 5 describes the fixpoint analysis, based on that abstract interpretation. Section 6 shows the way we deal with arrays held in fields rather than in local variables, and with calls having side-effects, like `split()` in Fig. 1. Section 7 shows how we lift the local results of Sections 5 and 6 to global invariants about the elements of fields of array type (the lifting of `local` to `global`, in the example above). Section 8 presents our experiments of analysis. Section 9 concludes.

3. A simple imperative language and its semantics

We present here a simplified imperative language, inspired by [7]. The actual implementation of our analysis includes all features of mono-threaded Java bytecode such as classes, method calls and exceptions.

In our language, commands are labeled actions. These *actions* are executed when the interpreter of the language is at a given, *initial* label and lead to another, *successor* label. More actions can share the same initial label and hence our language is, in general, non-deterministic. The exact nature of labels is irrelevant: we can assume, for instance, that they are integers. At run-time, variables hold values which, in a programming language such as Java, may be primitive or non-primitive. In

LV	::=	x	Variable
	|	E[E]	Array element
	|	E.f	Field access
$\mathbb{E} \ni$ E	::=	n	Integer
	|	x	Variable
	|	E $\oplus$ E	$\oplus \in \{+, -, *, \div, \%\}$
	|	E.$length$	Array length
	|	E[E]	Array element
	|	E.f	Field access
$\mathbb{B} \ni$ B	::=	true	Truth
	|	false	Falsity
	|	$\neg$B	Negation
	|	$E \otimes E$	$\otimes \in \{<, \leq, =\}$
	|	$B \oslash B$	$\oslash \in \{\wedge, \vee\}$
$\mathbb{A} \ni$ A	::=	B	Test
	|	LV := E	Assignment
	|	LV := `new` t[E] $\cdots$ [E]	Creation of an array and assignment
	|	LV := `new` $C()$	Creation of an object and assignment
$\mathbb{C} \ni$ C	::=	L_1 : A $\to$ L_2;	Command

Fig. 3. Abstract syntax of programs. LV are *leftvalues*, that is, expressions that may occur on the left-hand side of an assignment, $\mathbb{E}$ and $\mathbb{B}$ are *arithmetic* and *Boolean* expressions, $\mathbb{A}$ are *actions* and $\mathbb{C}$ are *commands*.

our formalization we simplify the picture by considering that the only primitive type is int, while non-primitive types are *arrays* and *classes* containing *instance fields* only.

Definition 1 (*Types*)**.** We let $\mathcal{K}$ denote the set of *classes*, where every class might have at most one direct *superclass* and an arbitrary number of direct *subclasses*. We define $\mathcal{T}$, the set of *types*, as the minimal set containing int, $\mathcal{K}$ and $t[\,]$ for every $t \in \mathcal{T}$. We use $\mathcal{A}$ to denote the set of all *array types*. Every variable $v \in$ Var has a static type $\mathsf{t}(v)$, where Var denotes the set of all program variables. Every class $\kappa \in \mathcal{K}$ might have *instance fields* $\kappa.f : t$ (field f of type $t \in \mathcal{T}$ defined in class κ) We let $\mathsf{F}(\kappa)$ denote the set of all κ's fields. Every array type $t = t_1[] \in \mathcal{A}$ has its own *arity*, $\mathsf{arity}(t)$, which is 1 if $t_1 \in \{\mathsf{int}\} \cup \mathcal{K}$ or $1 + \mathsf{arity}(t_1)$ otherwise.

Definition 2 (*Type Ordering*)**.** We define the *type ordering* $\leq : \mathcal{T} \times \mathcal{T}$ as follows: given two types $t_1, t_2 \in \mathcal{T}$, we say that t_1 is a *subtype* of t_2 and we denote it by $t_1 \leq t_2$ if and only if:

- $t_1 = t_2$, or
- $t_1, t_2 \in \mathcal{K}$ and t_1 is a subclass of t_2, or
- $t_1 = t'_1[], t_2 = t'_2[] \in \mathcal{A}$, $\mathsf{arity}(t_1) = \mathsf{arity}(t_2)$ and $t'_1 \leq t'_2$.

Example 1. Let ForeignStudent and Student be two classes such that the former is a subclass of the latter. Then, the following relations hold:

$$\begin{aligned} \mathsf{ForeignStudent} &\leq \mathsf{Student}, \\ \mathsf{ForeignStudent}[\,] &\leq \mathsf{Student}[\,], \\ \mathsf{ForeignStudent}[\,][\,] &\leq \mathsf{Student}[\,][\,], \text{ etc.} \end{aligned}$$

We define the arithmetic and Boolean expressions of our target language.

Definition 3 (*Expressions*)**.** The set of *arithmetic expressions* $\mathbb{E}$ and the set of *Boolean expressions* $\mathbb{B}$ are defined by the grammar in Fig. 3. Namely, arithmetic expressions can be *integer constants* ($n \in \mathbb{Z}$), *variables* ($x \in$ Var), *arithmetic operations* between two expressions (E $\oplus$ E, where $\oplus \in \{+, -, *, \div, \%\}$), *length of arrays* (E.*length*), *array elements* (E[E]) and *field accesses* (E.f). On the other hand, Boolean expressions can be *truth* (*true*), *falsity* (*false*), *negation* of a Boolean expression ($\neg$B), *comparison* of arithmetic expressions (E$\otimes$E, where $\otimes \in \{<, \leq, =\}$), *conjunction* and *disjunction* of Boolean expressions (B $\oslash$ B, where $\oslash \in \{\wedge, \vee\}$).

The following definition introduces the syntax of our target imperative intra-procedural language.

Definition 4 (*Syntax of Programs*)**.** A *program* is a finite set of *commands*, with a distinguished *initial command* C_{init}. $\mathbb{C}$ is the set of commands C of the form L_1 : A $\to$ L_2;, where L_1 and L_2 are called *initial* and *successor* labels of C, and A is the *action* executed by C. We define selectors ini(C) = L_1, suc(C) = L_2 and act(C) = A. Actions can be Boolean expressions or assignments (Fig. 3). The set of all actions is $\mathbb{A}$.

Example 2. Consider the two Java loops on the left of Fig. 4. We translate them into our simple language and show the result on the right of the same figure.

Java code	Transition system

...
1. a = new int[3];
2. i = 0;
3. **while**(i < a.length) {
4.   a[i] = i+1;
5.   i = i+1;
  }
6. ...
	C_0 1: $a :=$ new int[3] → 2;
	C_1 2: $i := 0 \to 3$;
	C_2 3: $i < a.length \to 4$;
	C_3 3: $\neg(i < a.length) \to 6$;
	C_4 4: $a[i] := i + 1 \to 5$;
	C_5 5: $i := i + 1 \to 3$;
	C_6 6: $\cdots$

...
1. i = 0;
2. **while**(i < b.length) {
3.   j = 0;
4.   **while**(j < b[i].length) {
5.   b[i][j] = i+1;
6.   j++;
    }
7.   c[i] = new int[2];
8.   i++;
  }
9. ...
	C_0 1: $i := 0 \to 2$;
	C_1 2: $i < b.length \to 3$;
	C_2 2: $\neg(i < b.length) \to 9$;
	C_3 3: $j := 0 \to 4$;
	C_4 4: $j < b[i].length \to 5$;
	C_5 4: $\neg(j < b[i].length) \to 7$;
	C_6 5: $b[i][j] = i + 1 \to 6$;
	C_7 6: $j := j + 1 \to 4$;
	C_8 7: $c[i] =$ new int[2] → 8;
	C_9 8: $i := i + 1 \to 2$;
	C_{10} 9: $\cdots$

Fig. 4. Java loops that initialize arrays a and b and their transition systems.

Let us now introduce a notion of *reachable types*. Intuitively, int can reach only itself, array types can reach themselves, their subtypes as well as the types reachable from the array types of the same form but with a smaller arity, while classes can reach themselves, their subclasses and the types reachable from the types of their fields.

Definition 5 (*Reachable Types*)**.** Given a type $t \in \mathcal{T}$, we define a function reachTypes : $\mathcal{T} \to \wp(\mathcal{T})$ as:

$$\mathsf{reachTypes}(t) = \begin{cases} \{\mathsf{int}\} & \text{if } t = \mathsf{int} \\ \{t' \mid t' \leq t\} \cup \mathsf{reachTypes}(t_1) & \text{if } t = t_1[\,] \in \mathcal{A} \\ \{t' \mid t' \leq t\} \cup \bigcup\limits_{\{\kappa.f:t_1 \in \mathsf{F}(t') \mid t' \leq t\}} \mathsf{reachTypes}(t_1) & \text{if } t \in \mathcal{K}. \end{cases}$$

Example 3. According to Definition 5, the types reachable from int[ ][ ] are int[ ][ ], int[ ] and int. Suppose that the class Student introduced in Example 1 contains a field id of type int. Then, among the types reachable from Student are, for sure, Student, ForeignStudent and int.

The static typing function t for variables is extended to non-Boolean expressions, in a standard way. Hence we can talk of the static type t(E) of an expression E. Expressions and actions of a program are assumed type-checked, in the usual sense of a typical imperative programming language. For instance, the indexes of an array must be expressions of type int and assignments can be performed only if the right-hand side and the left-hand side have compatible types. Then, in an action $x[y[3]][i] := z$ we require that $\mathsf{t}(y) = \mathsf{int}[\,]$, $\mathsf{t}(i) = \mathsf{int}$ and $\mathsf{t}(x) = \mathsf{t}(z)[\,]$. Hence, in the following, we silently assume that expressions and actions are type-checked.

We define a map modVar that characterizes the set of variables modified by an action.

Definition 6. We define a function modVar: $\mathbb{A} \to \wp(\mathsf{Var})$ as:

$$\mathsf{modVar}(\mathsf{A}) = \begin{cases} \{x\} & \text{if A is } x := \cdots \\ \varnothing & \text{otherwise.} \end{cases}$$

Another piece of information, used in our static analysis, is the static types of the arrays that might be updated by the actions of the program under analysis. The following definition introduces the map modArr.

Definition 7. We define a function modArray : $\mathbb{A} \to \wp(\mathcal{T})$ as:

$$\mathsf{modArray}(\mathsf{A}) = \begin{cases} \{\mathsf{t}(\mathsf{E}_0)\} & \begin{array}{l}\text{if A is } \mathsf{E}_0[\mathsf{E}_1] := \mathsf{E}_2 \text{ or} \\ \text{if A is } \mathsf{E}_0[\mathsf{E}_1] := \mathtt{new}\ t[\mathsf{E}_2]\cdots[\mathsf{E}_n] \text{ or} \\ \text{if A is } \mathsf{E}_0[\mathsf{E}_1] := \mathtt{new}\ C()\end{array} \\ \varnothing & \text{otherwise.} \end{cases}$$

Recall that in this paper the only primitive type is int and the reference types are arrays and classes. This simplification does not limit the results we obtain in the present paper, since the latter only concerns arrays, possibly held in fields of classes, and integer counters. Note that multi-dimensional arrays are represented as mono-dimensional arrays containing other arrays as their elements. This is the representation that Java uses at run-time.

Definition 8 (*Values*). We define Val, the set of *values* that variables can be assigned to, as $\mathsf{Val} = \mathbb{Z} \cup \mathbb{L} \cup \{\mathtt{null}\}$, where $\mathbb{Z}$ is the set of integers, $\mathbb{L}$ is a finite set of *memory locations* and `null` is a value that can be assigned to any reference type. Arr is the set of arrays $a = \langle n, [v_0, \ldots, v_{n-1}]\rangle$, where $n \in \mathbb{N}$ is the length of a and $v_i \in \mathsf{Val}$ are its elements, for $i \in [0..n) \subseteq \mathbb{N}$. We define $a.\mathsf{length} = n$ and $a[i] = v_i$, for $i \in [0..n)$. We also define the *update* of a at i as $a[i \mapsto v] = \langle n, [v_0, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_{n-1}]\rangle \in \mathsf{Arr}$, which is undefined when i is outside the range of a. Every object o has class $o.\kappa$ and an internal environment $o.\phi$ mapping each field $f \in \mathsf{F}(o.\kappa)$ into its value $(o.\phi)(f) \in \mathsf{Val}$. The set of all objects is Obj.

Definition 9 (*Default Values*). The *default value* of a variable x is 0 if $\mathsf{t}(x) = \mathsf{int}$, and `null` otherwise. For every type $t \in \mathcal{T}$, we write d_t to denote the default value of variables of static type t.

An environment represents the state of an interpreter of the language. It provides a value for each variable and specifies the memory of the system.

Definition 10 (*Environment*). An *environment* is a tuple $e = \langle \rho, \mu_a, \mu_o\rangle$ of a total map $\rho : \mathsf{Var} \to \mathsf{Val}$, array memory $\mu_a : \mathbb{L} \to \mathsf{Arr}$ and object memory $\mu_o : \mathbb{L} \to \mathsf{Obj}$. Like in Java, we ban dangling pointers and we require that static types are respected. We let $\mathcal{E}$ be the set of all environments.

Example 4. At the end of the Java fragment given at the top of Fig. 4 we have $\rho(i) = 3 \in \mathbb{Z}$ and $\rho(a) = \ell \in \mathbb{L}$, where ℓ is such that $\mu_a(\ell) = \langle 3, [1, 2, 3]\rangle \in \mathsf{Arr}$. Hence, $\mu_a(\ell).\mathsf{length} = 3$ and for each $i \in [0..\mu_a(\ell).\mathsf{length})$, we have $\mu_a(\ell)[i] = i + 1 \in \mathbb{Z}$.

Definitions 11–13 introduce three *partial* maps providing the *evaluation of arithmetic expressions*, the *evaluation of Boolean expressions* and the *semantics of actions*. Since they are partial maps, they are undefined on parameters not belonging to their domains.

Definition 11. We define, for every environment $e = \langle \rho, \mu_a, \mu_o\rangle \in \mathcal{E}$, the evaluation of non-Boolean expressions as a partial map $\mathcal{A}[\![\mathsf{E}]\!] : \mathcal{E} \to \mathsf{Val}$:

$$\mathcal{A}[\![n]\!]e = n$$

$$\mathcal{A}[\![x]\!]e = \rho(x)$$

$$\mathcal{A}[\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]e = \begin{cases} \mathcal{A}[\![\mathsf{E}_1]\!]e \oplus \mathcal{A}[\![\mathsf{E}_2]\!]e \\ \quad \text{if } \oplus \text{ is defined on those values} \\ \mathsf{undefined} \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{A}[\![\mathsf{E}.\mathit{length}]\!]e = \begin{cases} \mu_a(\ell).\mathsf{length} \\ \quad \text{if } \ell = \mathcal{A}[\![\mathsf{E}]\!]e \in \mathbb{L} \text{ and } \mu_a(\ell) \in \mathsf{Arr} \\ \mathsf{undefined} \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{A}[\![\mathsf{E}_1[\mathsf{E}_2]]\!]e = \begin{cases} \mu_a(\ell)[i] \\ \quad \text{if } \ell = \mathcal{A}[\![\mathsf{E}_1]\!]e \in \mathbb{L},\ \mu_a(\ell) \in \mathsf{Arr}, \\ \quad i = \mathcal{A}[\![\mathsf{E}_2]\!]e \text{ and } 0 \le i < \mu_a(\ell).\mathsf{length} \\ \mathsf{undefined} \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{A}[\![\mathsf{E}.f]\!]e = \begin{cases} (\mu_o(\ell).\phi)(f) \\ \quad \text{if } \ell = \mathcal{A}[\![\mathsf{E}]\!]e \in \mathbb{L} \text{ and } \mu_o(\ell) \in \mathsf{Obj} \\ \mathsf{undefined} \\ \quad \text{otherwise.} \end{cases}$$

In this paper, error situations are represented through undefined behaviors. We could have introduced also a notion of exceptional state, but this would have made the formalization heavier.

Definition 12. The evaluation of Boolean expressions is a partial function $\mathcal{B}[\![\mathsf{E}]\!] : \mathcal{E} \to \{\mathit{true}, \mathit{false}\}$ defined as:

$$\mathcal{B}[\![\mathsf{true}]\!]e = \mathit{true}$$

$$\mathcal{B}[\![\mathsf{false}]\!]e = \mathit{false}$$

$$\mathcal{B}[\![\neg \mathsf{B}]\!]e = \neg \mathcal{B}[\![\mathsf{B}]\!]e$$

$$\mathcal{B}[\![\mathsf{E}_1 \oslash \mathsf{E}_2]\!]e = \mathcal{A}[\![\mathsf{E}_1]\!]e \oslash \mathcal{A}[\![\mathsf{E}_2]\!]e$$

$$\mathcal{B}[\![\mathsf{B}_1 \otimes \mathsf{B}_2]\!]e = \mathcal{B}[\![\mathsf{B}_1]\!]e \otimes \mathcal{B}[\![\mathsf{B}_2]\!]e.$$

This map is undefined when any of its arguments is undefined.

The execution of an action maps an initial environment into the subsequent environment, if any. Note that Boolean expressions are actions that filter those environments that make them true. It is worth noting that, in our target language, conditionals are used in complementary pairs. For instance, the comparison $i < a.length$ at line 3 of the Java fragment given in the top left corner of Fig. 4 is translated into the pair of commands 3: $i < a.length \to$ 4; and 3: $\neg(i < a.length) \to$ 6;.

Definition 13 (*Semantics of Actions*)**.** Given an action A and an environment $e \in \mathcal{E}$, the *semantics* of A in e is given by a partial map $\mathcal{S}[\![A]\!] : \mathcal{E} \to \mathcal{E}$ defined as:

$$\mathcal{S}[\![B]\!]e = \begin{cases} e & \text{if } \mathcal{B}[\![B]\!]e = true \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![x := E]\!]e = \langle \rho[x \mapsto \mathcal{A}[\![E]\!]e], \mu_a, \mu_o \rangle$$

$$\mathcal{S}[\![E_1[E_2] := E_3]\!]e = \begin{cases} \langle \rho, \mu_a\left[\mu_a(\ell)[i \mapsto \mathcal{A}[\![E_3]\!]e]\right], \mu_o \rangle \\ \quad \text{if } \ell = \mathcal{A}[\![E_1]\!]e \in \mathbb{L}, \mu_a(\ell) \in \text{Arr}, \\ \quad i = \mathcal{A}[\![E_2]\!]e \text{ and } 0 \le i < \mu_a(\ell).\text{length} \\ \text{undefined} \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![E_1.f := E_2]\!]e = \begin{cases} \langle \rho, \mu_a, \mu_o\left[(\mu_o(\ell).\phi)(f) \mapsto \mathcal{A}[\![E_2]\!]e\right] \rangle \\ \quad \text{if } \ell = \mathcal{A}[\![E_1]\!]e \in \mathbb{L} \text{ and } \mu_o(\ell) \in \text{Obj} \\ \text{undefined} \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![x := \texttt{new}\; t[E_1] \cdots [E_n]]\!]e = \langle \rho[x \mapsto \ell_f], \mu_a(\ell_f, t, \mathcal{A}[\![E_1]\!]e, \ldots, \mathcal{A}[\![E_n]\!]e), \mu_o \rangle$$

$$\mathcal{S}[\![E[E'] := \texttt{new}\; t[E_1] \cdots [E_n]]\!]e = \begin{cases} \langle \rho, \mu_a(\ell_f, t, w_1, \ldots, w_n)[\mu_a(\ell)[i \mapsto \ell_f]], \mu_o \rangle \\ \quad \text{if } \ell = \mathcal{A}[\![E]\!]e \in \mathbb{L}, \mu_a(\ell) \in \text{Arr}, \\ \quad \forall 1 \le i \le n, w_i = \mathcal{A}[\![E_i]\!]e \in \mathbb{Z}, \\ \quad i = \mathcal{A}[\![E']\!]e \text{ and } 0 \le i < \mu(\ell).\text{length} \\ \text{undefined} \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![E.f := \texttt{new}\; t[E_1] \cdots [E_n]]\!]e = \begin{cases} \langle \rho, \mu_a(\ell_f, t, w_1, \ldots, w_n), \mu_o[(\mu_o(\ell).\phi)(f) \mapsto \ell_f] \rangle \\ \quad \text{if } \ell = \mathcal{A}[\![E]\!]e \in \mathbb{L}, \mu_o(\ell) \in \text{Obj and} \\ \quad \forall 1 \le i \le n, w_i = \mathcal{A}[\![E_i]\!]e \in \mathbb{Z} \\ \text{undefined} \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![x := \texttt{new}\; C()]\!]e = \langle \rho[x \mapsto \ell_f], \mu_a, \mu_o[\ell_f \mapsto o_f] \rangle$$

$$\mathcal{S}[\![E[E'] := \texttt{new}\; C()]\!]e = \begin{cases} \langle \rho, \mu_a[\mu_a(\ell)[i \mapsto \ell_f]], \mu_o[\ell_f \mapsto o_f] \rangle \\ \quad \text{if } \ell = \mathcal{A}[\![E]\!]e \in \mathbb{L}, \mu_a(\ell) \in \text{Arr}, \\ \quad i = \mathcal{A}[\![E']\!]e \text{ and } 0 \le i < \mu(\ell).\text{length} \\ \text{undefined} \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![E.f := \texttt{new}\; C()]\!]e = \begin{cases} \langle \rho, \mu_a, \mu_o[(\mu_o(\ell).\phi)(f) \mapsto \ell_f, \ell_f \mapsto o_f] \rangle \\ \quad \text{if } \ell = \mathcal{A}[\![E]\!]e \in \mathbb{L} \text{ and } \mu_o(\ell) \in \text{Obj} \\ \text{undefined} \\ \quad \text{otherwise,} \end{cases}$$

where ℓ_f is a fresh location, o_f is a new created object such that for each field $\kappa.f{:}t \in \mathsf{F}(o.\kappa)$, its value $(o.\phi)(\kappa.f{:}t)$ is d_t, and $\mu(\ell, t, l_n, \ldots, l_1)$ is the memory μ enriched with the creation of a new array of dimensions $l_1, \ldots, l_n$, bound to location ℓ, of basic type t. Namely,

$$\mu(\ell, t, l_1) = \mu[\ell \mapsto \langle l_1, [\mathsf{d}_t, \ldots, \mathsf{d}_t] \rangle]$$
$$\mu(\ell, t, l_{n+1}, l_n, \ldots, l_1) = \mu(\ell_1, t, l_n, \ldots, l_1) \ldots (\ell_{l_{n+1}}, t, l_n, \ldots, l_1)$$
$$[\ell \mapsto \langle l_{n+1}, [\ell_1, \ldots, \ell_{l_{n+1}}] \rangle]$$

where $\ell_1, \ldots, \ell_{l_{n+1}}$ are all fresh. This is just the formalization of the allocation algorithm of multi-dimensional arrays, as performed by the Java runtime.

In the following examples we will omit the object memory component from the environments. That is because our examples refer to arrays of type int[] or int[][], and their updates do not give rise to any modification of the object memory.

Example 5. The semantics of the first action of the fragment considered in Example 4 (see Fig. 4), $A_0 = a := \texttt{new int}[3]$, in an arbitrary environment e is $e_0 = \mathcal{S}[\![A_0]\!]e = \langle [a \mapsto \ell], [\ell \mapsto \langle 3, [0, 0, 0] \rangle] \rangle = \langle \rho_0, \mu_0 \rangle$, where $\ell \in \mathbb{L}$ is fresh and 0 is the default value for int. The semantics of $A_1 = i := 0$ in environment e_0 is $e_1 = \mathcal{S}[\![A_1]\!]e_0 = \langle [i \mapsto 0, a \mapsto \ell], \mu_0 \rangle = \langle \rho_1, \mu_1 \rangle$. The semantics of $A_2 = i < a.\text{length}$ in environment e_1 is $\mathcal{S}[\![A_2]\!]e_1 = e_1$ since $\mathcal{B}[\![A_2]\!]e_1 = (\mathcal{A}[\![i]\!]e_1 < \mathcal{A}[\![a.\text{length}]\!]e_1) = (\rho_1(i) < \mu_1(\rho_1(a)).\text{length}) = (0 < 3) = true$.

Command	Traces
$@\mathsf{C}_0$	$\langle e, \mathsf{C}_0\rangle$
$@\mathsf{C}_1$	$\langle\langle\langle e, \mathsf{C}_0\rangle, \langle e_0, \mathsf{C}_1\rangle\rangle\rangle$
$@\mathsf{C}_2$	$\tau_2^1 = \langle\langle e, \mathsf{C}_0\rangle, \langle e_0, \mathsf{C}_1\rangle, \langle e_1, \mathsf{C}_2\rangle\rangle$ $\tau_2^2 = \tau_5^1 \circ \langle e_3, \mathsf{C}_2\rangle$ $\tau_2^3 = \tau_5^2 \circ \langle e_5, \mathsf{C}_2\rangle$ $\tau_2^4 = \tau_5^3 \circ \langle e_7, \mathsf{C}_2\rangle$
$@\mathsf{C}_3$	$\tau_3^1 = \langle\langle e, \mathsf{C}_0\rangle, \langle e_0, \mathsf{C}_1\rangle, \langle e_1, \mathsf{C}_3\rangle\rangle$ $\tau_3^2 = \tau_5^1 \circ \langle e_3, \mathsf{C}_3\rangle$ $\tau_3^3 = \tau_5^2 \circ \langle e_5, \mathsf{C}_3\rangle$ $\tau_3^4 = \tau_5^3 \circ \langle e_7, \mathsf{C}_3\rangle$
$@\mathsf{C}_4$	$\tau_4^1 = \tau_2^1 \circ \langle e_1, \mathsf{C}_4\rangle$ $\tau_4^2 = \tau_2^2 \circ \langle e_3, \mathsf{C}_4\rangle$ $\tau_4^3 = \tau_2^3 \circ \langle e_5, \mathsf{C}_4\rangle$
$@\mathsf{C}_5$	$\tau_5^1 = \tau_4^1 \circ \langle e_2, \mathsf{C}_5\rangle$ $\tau_5^2 = \tau_4^2 \circ \langle e_4, \mathsf{C}_5\rangle$ $\tau_5^3 = \tau_4^3 \circ \langle e_6, \mathsf{C}_5\rangle$
$@\mathsf{C}_6$	$\tau_6 = \tau_3^4 \circ \langle e_7, \mathsf{C}_6\rangle$

Environments
$e \in \mathcal{E}$
$e_0 = \langle[a \mapsto \ell], [\ell \mapsto \langle 3, [0,0,0]\rangle]\rangle$
$e_1 = \langle[i \mapsto 0, a \mapsto \ell], [\ell \mapsto \langle 3, [0,0,0]\rangle]\rangle$
$e_2 = \langle[i \mapsto 0, a \mapsto \ell], [\ell \mapsto \langle 3, [1,0,0]\rangle]\rangle$
$e_3 = \langle[i \mapsto 1, a \mapsto \ell], [\ell \mapsto \langle 3, [1,0,0]\rangle]\rangle$
$e_4 = \langle[i \mapsto 1, a \mapsto \ell], [\ell \mapsto \langle 3, [1,2,0]\rangle]\rangle$
$e_5 = \langle[i \mapsto 2, a \mapsto \ell], [\ell \mapsto \langle 3, [1,2,0]\rangle]\rangle$
$e_6 = \langle[i \mapsto 2, a \mapsto \ell], [\ell \mapsto \langle 3, [1,2,3]\rangle]\rangle$
$e_7 = \langle[i \mapsto 3, a \mapsto \ell], [\ell \mapsto \langle 3, [1,2,3]\rangle]\rangle$

Operational Semantics at C_6
$\{\epsilon\} \Rightarrow^{\mathsf{C}_0} @\mathsf{C}_0 \Rightarrow^{\mathsf{C}_1} @\mathsf{C}_1 \Rightarrow^{\mathsf{C}_2} \{\tau_2^1, \tau_3^1\}$
$\Rightarrow^{\mathsf{C}_4} \{\tau_4^1\} \Rightarrow^{\mathsf{C}_5} \{\tau_5^1\} \Rightarrow^{\mathsf{C}_2} \{\tau_2^2\}$
$\Rightarrow^{\mathsf{C}_4} \{\tau_4^2\} \Rightarrow^{\mathsf{C}_5} \{\tau_5^2\} \Rightarrow^{\mathsf{C}_2} \{\tau_2^3\}$
$\Rightarrow^{\mathsf{C}_4} \{\tau_4^3\} \Rightarrow^{\mathsf{C}_5} \{\tau_5^3\} \Rightarrow^{\mathsf{C}_3} \{\tau_3^4\} \Rightarrow^{\mathsf{C}_6} \{\tau_6\}$
$\Rightarrow @\mathsf{C}_6 = \{\tau_6\}$

Fig. 5. Finite partial traces and environments of the program from Fig. 4.

Example 6. Consider an action $\mathsf{A} = a := \mathtt{new}\, t[2][3]$ and an environment $e = \langle \rho, \mu\rangle \in \mathcal{E}$, where $t = \mathsf{int}[]$. Then, $\mathcal{S}[\![\mathsf{A}]\!]e = \langle \rho[a \mapsto \ell], \mu(\ell, t, 2, 3)\rangle$, where $\ell \in \mathbb{L}$ is fresh. By Definition 5, $\mu(\ell, t, 2, 3) = \mu(\ell_1, t, 3)(\ell_2, t, 3)[\ell \mapsto \langle 2, [\ell_1, \ell_2]\rangle]$, where $\ell_1, \ell_2 \in \mathbb{L}$ are fresh locations. Since $\mathsf{d}_t = \mathtt{null}$, we obtain $\mu(\ell, t, 2, 3) = \mu[\ell_1 \mapsto \langle 3, [\mathtt{null}, \mathtt{null}, \mathtt{null}]\rangle, \ell_2 \mapsto \langle 3, [\mathtt{null}, \mathtt{null}, \mathtt{null}]\rangle, \ell \mapsto \langle 2, [\ell_1, \ell_2]\rangle]$.

Our operational semantics works over execution traces of states. A state is an environment enriched with a component recording the next command to be executed, similar to the program counter in an actual interpreter of the language.

Definition 14 (*State*)**.** A *state* is a pair $\sigma = \langle e, \mathsf{C}\rangle \in \mathcal{E} \times \mathbb{C}$. The set of states is denoted by Σ. We define the selectors $\mathsf{env}(\sigma) = e$ and $\mathsf{cmd}(\sigma) = \mathsf{C}$.

Example 7. The initial state of the program fragment considered in Example 5 is $\sigma_0 = \langle e, \mathsf{C}_0\rangle$, while the states obtained after the execution of the first three commands are $\sigma_1 = \langle e_0, \mathsf{C}_1\rangle$, $\sigma_2 = \langle e_1, \mathsf{C}_2\rangle$ and $\sigma_3 = \langle e_1, \mathsf{C}_4\rangle$, respectively, where e_0 and e_1 are the environments calculated in Example 5.

A trace is a sequence of states that reflects an actual execution of the program, i.e., environments are modified in accordance to the semantics of the actions in the states of that sequence, while the sequence of labels corresponds to the sequence of commands in the program.

Definition 15 (*Trace*)**.** A *finite partial trace* τ of states is a finite sequence of states $\langle \sigma_1, \ldots, \sigma_n\rangle$, where $|\tau| = n$ is the *length* of τ. For every $i \in [1, \ldots, |\tau|)$, if $\sigma_i = \langle e, \mathsf{C}\rangle$, we require that $\mathcal{S}[\![\mathsf{act}(\mathsf{C})]\!]e$ is defined and that $\sigma_{i+1} = \langle \mathcal{S}[\![\mathsf{act}(\mathsf{C})]\!]e, \mathsf{C}'\rangle$ with $\mathsf{suc}(\mathsf{C}) = \mathsf{ini}(\mathsf{C}')$. When $n = 0$, the trace is empty and denoted by ϵ. Otherwise, we define $\mathsf{first}(\tau) = \sigma_1$ and $\mathsf{last}(\tau) = \sigma_n$. The set of traces is denoted by $\mathcal{T}$. The *concatenation* $\circ$ of two traces is defined as $\tau_1 \circ \epsilon = \tau_1$, $\epsilon \circ \tau_2 = \tau_2$ and

$$\langle \sigma_1^1, \ldots, \sigma_{n_1}^1\rangle \circ \langle \sigma_1^2, \ldots, \sigma_{n_2}^2\rangle = \begin{cases} \langle \sigma_1^1, \ldots, \sigma_{n_1}^1, \sigma_1^2, \ldots, \sigma_{n_2}^2\rangle \\ \quad \text{if } \mathsf{env}(\sigma_1^2) = \mathcal{S}[\![\mathsf{act}(\mathsf{cmd}(\sigma_{n_1}^1))]\!]\mathsf{env}(\sigma_{n_1}^1) \\ \qquad\qquad \text{is defined} \\ \quad \text{and } \mathsf{suc}(\mathsf{cmd}(\sigma_{n_1}^1)) = \mathsf{ini}(\mathsf{cmd}(\sigma_1^2)) \\ \text{undefined} \\ \quad \text{otherwise.} \end{cases}$$

We define the operational semantics of our language as a transformer of sets of traces: it expands every trace τ with a state whose next command to be executed is a given command C that can be attached to τ according to Definition 15.

Definition 16 (*Operational Semantics*)**.** Let $\mathsf{C} \in \mathbb{C}$ and $T \subseteq \mathcal{T}$. We define a function $\mathcal{S} : \mathbb{C} \times \wp(\mathcal{T}) \to \wp(\mathcal{T})$ as

$$\mathcal{S}(\mathsf{C}, T) = \{\tau \circ \langle e, \mathsf{C}\rangle \mid \tau \in T \wedge \exists e \in \mathcal{E}. \tau \circ \langle e, \mathsf{C}\rangle \in \mathcal{T}\}.$$

Given two sets $T, T' \subseteq \mathcal{T}$, we use $T \Rightarrow^{\mathsf{C}} T'$ to denote $\mathcal{S}(\mathsf{C}, T) = T'$. The *operational semantics* at C is the set $@\mathsf{C}$ of all possible traces that lead to C and start with the execution of a distinguished command C_{init}, that is, $@\mathsf{C} = \{\tau \in T_n \mid \exists T_1, \ldots, T_n \subseteq \mathcal{T}. \exists \mathsf{C}_1, \ldots, \mathsf{C}_n \in \mathbb{C}. \{\epsilon\} \Rightarrow^{\mathsf{C}_1} T_1 \Rightarrow^{\mathsf{C}_2} T_2 \cdots \Rightarrow^{\mathsf{C}_n} T_n \wedge \mathsf{C}_1 = \mathsf{C}_{init} \wedge \mathsf{C}_n = \mathsf{C}\}$.

Example 8. The second column of the table given in Fig. 5 (Traces) contains all the finite partial traces obtained during the execution of the program from Fig. 4. These traces are divided in 7 partitions corresponding to the commands of the program of interest (Command). Traces belonging to row $@\mathsf{C}_i$ of column Traces represent the operational semantics at C_i. Definitions of the traces mentioned above use different environments $e, e_0, \ldots, e_7$, and their definition is given at the top of the third column (Environments). At the bottom of the third column (Operational Semantics at C_6) we compute the operational semantics at command C_6.

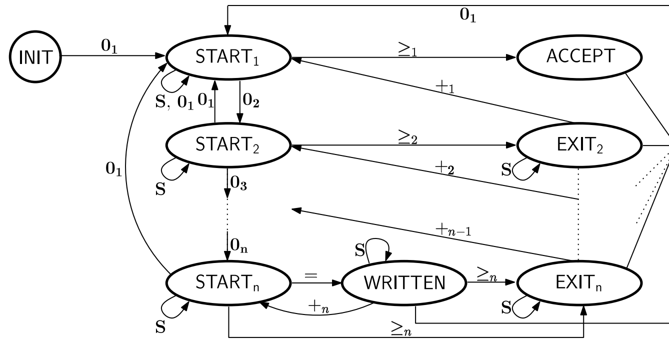


Fig. 6. Automaton detecting fully initialized arrays.

4. Regular trace approximation

We define here an approximation of the execution traces of a program through a finite deterministic automaton. Its states, that are just sets of traces, represent elements of an abstract domain. This domain is defined through regular expressions specifying sequences of commands that can be executed to construct the traces. We prove that the transition relation of the automaton is a correct approximation of the $\Rightarrow^{C}$ operational relation (Definition 16). Let a be an array of dimension n which is initialized through n nested loops with index variables $i_1, \ldots, i_n$ of integer type. We approximate the semantics of our programming language by the automaton given in Fig. 6, which is parameterized on a tuple of program variables $\langle a, i_1, \ldots, i_n\rangle$. The goal of this automaton is to detect nested loops that fully initialize the array a through index variables $i_1, \ldots, i_n$. The automaton's *alphabet* is $\Lambda = \{\mathbf{=}, \mathbf{S}, \mathbf{N}\} \cup \{\mathbf{0}_k, \mathbf{+}_k, \mathbf{\geq}_k \mid 1 \leq k \leq n\}$ and its *states* are $Q = \{\mathsf{INIT}, \mathsf{WRITTEN}, \mathsf{ACCEPT}\} \cup \{\mathsf{START}_k \mid 1 \leq k \leq n\} \cup \{\mathsf{EXIT}_k \mid 1 < k \leq n\}$.

As we formalize later (Definition 20), INIT means that nothing is known about the last executed commands. START_n means that the automaton is at the beginning of the loop for i_n. EXIT_n means that the automaton is at the end of the loop for i_n. WRITTEN means that an assignment to $a[i_1] \ldots [i_n]$ has just been executed and the automaton is waiting to match it with a corresponding unitary increment of i_n. Finally, ACCEPT means that the complete initialization of the array can be asserted.

As usual, the *transition relation* of the automaton is formalized by a function $\delta : Q \times \Lambda \to Q$: given states $\mathsf{p}, \mathsf{q} \in Q$ and $\lambda \in \Lambda$, if the automaton has a transition from p to q labeled by λ, then $\delta(\mathsf{p}, \lambda) = \mathsf{q}$. For simplicity, the automaton given in Fig. 6 is not complete, since there are missing transitions (for instance, there is no transition labeled by $\mathbf{0}_3$ that starts from START_1 or from WRITTEN). We silently assume that all these missing transitions lead to INIT.

The alphabet is an abstraction of the commands of the program.

Definition 17 (*Abstraction of Commands*). We define a function $s : \mathbb{C} \to \Lambda$ called *abstraction of commands* as

$$s(\langle \mathsf{L}_1 : \mathsf{A} \to \mathsf{L}_2; \rangle) = \begin{cases} \mathbf{0}_k & \text{if A is } i_k := 0 \\ \mathbf{+}_k & \text{if A is } i_k := i_k + 1 \\ \mathbf{\geq}_k & \text{if A is } \neg(i_k < a[i_1]...[i_{k-1}].length) \\ \mathbf{=} & \text{if A is } a[i_1] \ldots [i_n] := \mathsf{E} \\ \mathbf{S} & \text{otherwise, if } \mathsf{safe}(\mathsf{A}) = true \\ \mathbf{N} & \text{otherwise.} \end{cases}$$

We say that an action A is safe i.e., $\mathsf{safe}(\mathsf{A}) = true$, if it is a full array assignment, or if it does not modify the variable a nor any index variable $i_1, \ldots, i_n$, nor any array of a type reachable from $\mathsf{t}(a)$. Namely, $\mathsf{safe} : \mathbb{A} \to \{true, false\}$ is defined as $\mathsf{safe}(\mathsf{A}) = (\mathsf{modVar}(\mathsf{A}) \cap \{a, i_1, \ldots, i_n\} = \varnothing \wedge \mathsf{modArray}(\mathsf{A}) \cap \mathsf{reachTypes}(\mathsf{t}(a)) = \varnothing) \vee (\mathsf{A} \text{ is } a[j_1] \ldots [j_n] := \mathsf{E})$.

For any $k \in [1..n]$, commands with actions $i_k := 0$ and $i_k := i_k + 1$ are abstracted into $\mathbf{0}_k$ and $\mathbf{+}_k$ respectively. It is important to observe that this abstraction is syntactical. For instance, while a command with action $i_k := 0$ is abstracted into $\mathbf{0}_k$, another with action $i_k := 1 - 1$ is abstracted into $\mathbf{N}$. This does not affect the correctness of our analysis, but reduces its precision. We distinguish between full and partial array assignments: full array assignments are of the form $a[j_1] \ldots [j_n] := \mathsf{E}$, where for each $k \in [1..n]$, j_k satisfies bounds of kth array's dimension, while partial array assignments are of the form $a[j_1] \ldots [j_m]$, where $m < n$. Full array assignments of the form $a[i_1] \ldots [i_n] := \mathsf{E}$ are abstracted into $\mathbf{=}$, but any other full assignment is abstracted into $\mathbf{S}$ (*safe*), as imposed by the safe function. On the contrary, partial array assignments are abstracted into $\mathbf{N}$ (*non-safe*) because there are situations when some already initialized elements of the array of interest might be destroyed by a partial assignment to another array. Consider the Java fragment given at the bottom of Fig. 4. The inner loop (lines 4-6) fully initializes array $b[i]$, for each $i \in [0..b.length)$. But, at line 7, this initialization may be destroyed, since a new array of integers is created and assigned to array $c[i]$, and this array might be an alias of $b[i]$. Hence, we decided to consider not safe all the partial array assignments where the type of the array element modified by that command is reachable from the type of the array of interest. This choice, although sound, might be quite imprecise.

There are, though, some static analyses that improve the precision of our array initialization analysis. One of them is sharing analysis. We say that two variables a and b of array type share in an environment e when they reach a common

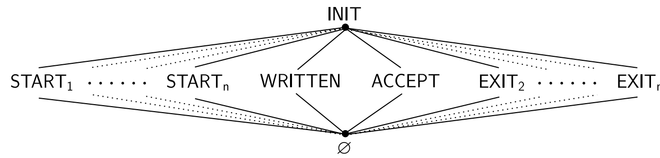


Fig. 7. The abstract domain $\mathcal{A}$.

location. A static analysis that determines, for each program point, an over-approximation of the pairs of variables that might share at that point can be integrated in our formalization by changing Definition 17 in the following way: we say that an action A is safe if it does not modify the array variable of interest a nor the index variables of interest $i_1, \ldots, i_n$ (i.e., $\mathsf{modVar}(\mathsf{A}) \cap \{a, i_1, \ldots, i_n\} = \varnothing$) and if the variables updated by A definitely do not share with a. Our static analyzer Julia implements this analysis [23].

The abstraction of a trace is just the element-wise application of s.

Definition 18 (*Abstraction of Traces*)**.** We definite a function $\beta : \mathcal{T} \to \Lambda^*$ called *abstraction of traces* as $\beta(\langle\sigma_1, \ldots, \sigma_{n-1}, \sigma_n\rangle) = \beta(\langle\sigma_1, \ldots, \sigma_{n-1}\rangle)\, s(\mathsf{cmd}(\sigma_n)) = s(\mathsf{cmd}(\sigma_1)) \ldots s(\mathsf{cmd}(\sigma_n))$, for non-empty traces, with $\beta(\epsilon) = \epsilon$.

The following definition introduces two functions that provide regular expressions describing the structure of the kth of n nested loops used for the initialization of array a: $\textsc{Tot}(k)$ and $\textsc{Part}(k)$ correspond to traces inside the loops that fully or partially initialize the first k dimensions of array a.

Definition 19 (*Completion*)**.** We define functions *total* and *partial completion*, $\textsc{Tot}$ and $\textsc{Part}$:

$$\textsc{Tot}(k) = \begin{cases} \mathbf{0}_n\mathbf{S}^*((\mathbf{=}\,\mathbf{S}^*)^+ + {}_n\mathbf{S}^*)^* \geq {}_n & \text{if } k = n \\ \mathbf{0}_k\mathbf{S}^*(\textsc{Tot}(k+1)\mathbf{S}^* + {}_k\mathbf{S}^*)^* \geq {}_k & \text{otherwise} \end{cases}$$

$$\textsc{Part}(k) = \begin{cases} \mathbf{0}_n\mathbf{S}^*((\mathbf{=}\,\mathbf{S}^*)^+ + {}_n\mathbf{S}^*)^* & \text{if } k = n \\ \mathbf{0}_k\mathbf{S}^*(\textsc{Tot}(k+1)\mathbf{S}^* + {}_k\mathbf{S}^*)^* & \text{otherwise,} \end{cases}$$

where * ($^+$) means zero or more (at least one) repetitions.

Note that, for every $k \in [1..n]$, we have $\textsc{Tot}(k) = \textsc{Part}(k) \geq {}_k$.

Since Λ contains abstractions of commands, the meaning of the states of the automaton, Q, becomes more clear. INIT means that nothing is known about the last executed commands. START_n means that assignments $i_1 := 0, \ldots, i_n := 0$ are executed and that they are potentially followed by an alternation of assignments to $a[i_1]\ldots[i_n]$ and unitary increments of i_n ($\textsc{Part}(n)$). START_{n-1} means that assignments $i_1 := 0, \ldots, i_{n-1} := 0$ are executed, they are potentially followed by an assignment $i_n := 0$ and in that case, there is an alternation of assignments to $a[i_1]\ldots[i_n]$ and unitary increments of i_n until a comparison $\geq {}_n$ is found ($\textsc{Tot}(n)$), then there is an unitary increment of i_{n-1}, which is potentially followed by another $\textsc{Tot}(n)$, and so on, but no unitary increments of i_{n-1} is followed by a comparison $\geq {}_{n-1}$ ($\textsc{Part}(n-1)$). States START_k for $k \in [1..n-2]$ are defined analogously. The only difference between states START_k and EXIT_k is the fact that the latter terminates with a comparison $\geq {}_k$, potentially followed by some safe actions ($\textsc{Tot}(k)$). WRITTEN means that an assignment to $a[i_1]\ldots[i_n]$ has just been executed and the automaton is waiting to match it with a corresponding unitary increment of i_n. Finally, ACCEPT means that the complete initialization of the array can be asserted. An arbitrary number of safe actions can always be executed between non-safe ones.

In formal terms, the states of the automaton are the sets of execution traces characterized by the following definition.

Definition 20 (*Abstract Domain*)**.** The states of the automaton in Fig. 6 correspond to the following sets of traces defined by regular expressions over Λ:

$$\begin{aligned}
\mathsf{INIT} &= \{\tau \in \mathcal{T} \mid \beta(\tau) \in \Lambda^*\} = \mathcal{T} \\
\mathsf{START}_k &= \{\tau \in \mathcal{T} \mid \beta(\tau) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)\}, \forall k \in [1..n] \\
\mathsf{EXIT}_k &= \{\tau \in \mathcal{T} \mid \beta(\tau) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k) \geq {}_k\mathbf{S}^*\}, \forall k \in [2..n] \\
\mathsf{WRITTEN} &= \{\tau \in \mathcal{T} \mid \beta(\tau) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)(\mathbf{=}\,\mathbf{S}^*)^+\} \\
\mathsf{ACCEPT} &= \{\tau \in \mathcal{T} \mid \beta(\tau) \in \Lambda^*\textsc{Part}(1) \geq {}_1\}.
\end{aligned}$$

We define the set $\mathcal{A} = \{\mathsf{INIT}, \mathsf{WRITTEN}, \mathsf{ACCEPT}, \varnothing\} \cup \{\mathsf{START}_k \mid 1 \leq k \leq n\} \cup \{\mathsf{EXIT}_k \mid 1 < k \leq n\}$.

Proposition 1. *$\mathcal{A}$ is a Moore family of $\wp(\mathcal{T})$ i.e., it is an abstract domain ordered by set inclusion (Fig. 7). As standard for Moore families, the induced abstraction map $\alpha : \wp(\mathcal{T}) \to \mathcal{A}$ is $\alpha(T) = \bigcap_{A \in \mathcal{A}, T \subseteq A} A$, for every $T \subseteq \mathcal{T}$.*

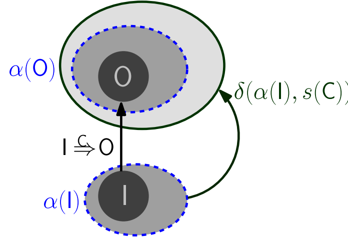
Proof. For any $k \in [1..n]$, the last safe instruction of any $\tau \in \mathsf{START}_k$ is $\mathbf{0}_k$ or $\mathbf{+}_k$; the last relevant instruction of any $\tau \in \mathsf{WRITTEN}$ is $\mathbf{=}$; the last relevant instruction of any $\tau \in \mathsf{ACCEPT}$ is $\geq {}_1$; if $k \neq 1$, the last relevant instruction of any $\tau \in \mathsf{EXIT}_k$ is $\geq {}_k \neq \geq {}_1$. Therefore, these $2n+1$ abstract elements are disjoint and their intersection is $\varnothing \in \mathcal{A}$. Since $\mathsf{INIT} = \mathcal{T}$, its intersection with any other abstract domain element p is p itself. Hence $\mathcal{A}$ is a Moore family. □

The following lemma shows that the transition function δ of the automaton given in Fig. 6 is monotonic.

Lemma 1. *Let* $\mathsf{p}, \mathsf{q} \in \mathcal{A}$ *and* $\lambda \in \Lambda$. *If* $\varnothing \subset \mathsf{p} \subseteq \mathsf{q}$ *then* $\delta(\mathsf{p}, \lambda) \subseteq \delta(\mathsf{q}, \lambda)$.

Proof. The result is immediate when $\mathsf{p} = \mathsf{q}$. Otherwise, from $\varnothing \subset \mathsf{p} \subset \mathsf{q}$ it follows (Fig. 7) that $\mathsf{q} = \mathsf{INIT}$. From Fig. 6, for all $\lambda \in \Lambda \setminus \{\mathbf{0}_1\}$ we have $\delta(\mathsf{q}, \lambda) = \mathsf{INIT}$ and hence $\delta(\mathsf{p}, \lambda) \subseteq \mathsf{INIT} = \delta(\mathsf{q}, \lambda)$. Moreover, again from Fig. 6, we have $\delta(\mathsf{p}, \mathbf{0}_1) = \mathsf{START}_1 = \delta(\mathsf{q}, \mathbf{0}_1)$. □

Lemma 2 states a consistency or correctness relation [6] between the operational semantics of Definition 16 and the transitions of the automaton in Fig. 6. Namely, suppose that two sets of traces I and O and a command C are connected by $\mathsf{I} \Rightarrow^{\mathsf{C}} \mathsf{O}$, i.e., each trace in O is obtained by attaching a state whose next command component is C to an opportune trace from I. Hence, the next command component of the last state of each trace belonging to O is C. We show that a similar property holds in abstract terms as well: if we perform a transition $s(\mathsf{C})$, representing the abstract behavior of C, from the automaton state $\alpha(\mathsf{I})$, corresponding to all the traces representable by the regular expressions that the traces of I give rise to, we reach an automaton state which contains at least the traces $\alpha(\mathsf{O})$.



The figure on the left illustrates this result: inner circles (with no borders) are I and O. Shapes with dashed borders are their abstractions through α. The shape with a solid border is the abstract state obtained by executing δ from $\alpha(\mathsf{I})$ and is, in general, an approximation of $\alpha(\mathsf{O})$.

Lemma 2. *Let* $\mathsf{C} \in \mathbb{C}$ *and* $\mathsf{I}, \mathsf{O} \subseteq \mathcal{T}$. *If* $\mathsf{I} \Rightarrow^{\mathsf{C}} \mathsf{O}$ *then* $\alpha(\mathsf{O}) \subseteq \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

Proof. Consider any $\tau \in \mathsf{O}$. By Definition 16, there exist $\tau' \in \mathsf{I}$ and $e \in \mathcal{E}$ such that $\tau = \tau' \circ \langle e, \mathsf{C}\rangle$. By Definition 18, we have $\beta(\tau) = \beta(\tau')s(\mathsf{C})$. We proceed by case analysis:

$\underline{\alpha(\mathsf{I}) = \mathsf{START}_k \wedge k \in [1..n] \wedge s(\mathsf{C}) = \mathbf{0}_1}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{START}_k$ and

$$\begin{aligned}\beta(\tau) &= \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)s(\mathsf{C})\\ &= \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)\mathbf{0}_1\\ &\subseteq \Lambda^*\mathbf{0}_1 \subseteq \Lambda^*\textsc{Part}(1).\end{aligned}$$

Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)$, we have $\tau \in \mathsf{START}_1$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{START}_1$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{START}_1) = \mathsf{START}_1 = \delta(\mathsf{START}_k, \mathbf{0}_1) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{START}_k \wedge k \in [1..n) \wedge s(\mathsf{C}) = \mathbf{0}_{k+1}}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{START}_k$ and

$$\begin{aligned}\beta(\tau) &= \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)s(\mathsf{C})\\ &= \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)\mathbf{0}_{k+1}\\ &\subseteq \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)\textsc{Part}(k+1).\end{aligned}$$

Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k+1)$, we have $\tau \in \mathsf{START}_{k+1}$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{START}_{k+1}$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{START}_{k+1}) = \mathsf{START}_{k+1} = \delta(\mathsf{START}_k, \mathbf{0}_{k+1}) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{START}_k \wedge k \in [1..n) \wedge s(\mathsf{C}) = \mathbf{0}_l \wedge l \notin \{1, k+1\}}$: It is clear that, in this case $\delta(\alpha(\mathsf{I}), s(\mathsf{C})) = \delta(\mathsf{START}_k, \mathbf{0}_l) = \mathsf{INIT}$, hence for any O, $\alpha(\mathsf{O}) \subseteq \mathsf{INIT} = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{START}_n \wedge s(\mathsf{C}) = \mathbf{0}_l \wedge l \notin \{1, n\}}$: In this case we have $\delta(\alpha(\mathsf{I}), s(\mathsf{C})) = \delta(\mathsf{START}_n, \mathbf{0}_l) = \mathsf{INIT}$, hence for any O, $\alpha(\mathsf{O}) \subseteq \mathsf{INIT} = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{START}_k \wedge s(\mathsf{C}) = \boldsymbol{+}_l \wedge k, l \in [1..n]}$: In this case we have $\delta(\alpha(\mathsf{I}), s(\mathsf{C})) = \delta(\mathsf{START}_k, \boldsymbol{+}_l) = \mathsf{INIT}$, hence for any O, $\alpha(\mathsf{O}) \subseteq \mathsf{INIT} = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{START}_k \wedge k \in [1..n) \wedge s(\mathsf{C}) = \boldsymbol{=}}$: In this case we have $\delta(\alpha(\mathsf{I}), s(\mathsf{C})) = \delta(\mathsf{START}_k, \boldsymbol{=}) = \mathsf{INIT}$, hence for any O, $\alpha(\mathsf{O}) \subseteq \mathsf{INIT} = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{START}_n \wedge s(\mathsf{C}) = \boldsymbol{=}}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{START}_n$ and therefore:

$$\begin{aligned}\beta(\tau) &= \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)s(\mathsf{C})\\ &= \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)\boldsymbol{=}\\ &\subseteq \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)(\boldsymbol{=}\,\mathbf{S}^*)^+.\end{aligned}$$

Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)(\boldsymbol{=}\,\mathbf{S}^*)^+$, we have $\tau \in \mathsf{WRITTEN}$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{WRITTEN}$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{WRITTEN}) = \mathsf{WRITTEN} = \delta(\mathsf{START}_n, \boldsymbol{=}) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{START}_k \wedge s(\mathsf{C}) = \geq_k \wedge k \in [2..n]}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{START}_k$ and

$$\begin{aligned}\beta(\tau) &= \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)s(\mathsf{C})\\ &= \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k) \geq_k\\ &\subseteq \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k) \geq_k \mathbf{S}^*.\end{aligned}$$

Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)(= \mathbf{S}^*)^* \geq_k \mathbf{S}^*$, we have $\tau \in \mathsf{EXIT}_k$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{EXIT}_k$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{EXIT}_k) = \mathsf{EXIT}_k = \delta(\mathsf{START}_k, \geq_k) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{START}_1 \wedge s(\mathsf{C}) = \geq_1}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{START}_1$ and therefore $\beta(\tau) = \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)s(\mathsf{C}) = \Lambda^*\textsc{Part}(1) \geq_k$. Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1) \geq_k$, we have $\tau \in \mathsf{ACCEPT}$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{ACCEPT}$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{ACCEPT}) = \mathsf{ACCEPT} = \delta(\mathsf{START}_1, \geq_1) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{START}_k \wedge k \in [1..n] \wedge s(\mathsf{C}) = \geq_l \wedge l \neq k}$: In this case $\delta(\alpha(\mathsf{I}), s(\mathsf{C})) = \delta(\mathsf{START}_k, \geq_l) = \mathsf{INIT}$, hence for any O, $\alpha(\mathsf{O}) \subseteq \mathsf{INIT} = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{START}_k \wedge k \in [1..n] \wedge s(\mathsf{C}) = \mathbf{N}}$: In this case we have $\delta(\alpha(\mathsf{I}), s(\mathsf{C})) = \delta(\mathsf{START}_k, \mathbf{N}) = \mathsf{INIT}$, hence for any O, $\alpha(\mathsf{O}) \subseteq \mathsf{INIT} = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{WRITTEN} \wedge s(\mathsf{C}) = \mathbf{0}_1}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{WRITTEN}$ and therefore

$$\begin{aligned}\beta(\tau) &= \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)(= \mathbf{S}^*)^+s(\mathsf{C})\\ &= \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k)(= \mathbf{S}^*)^+\mathbf{0}_1\\ &\subseteq \Lambda^*\mathbf{0}_1 \subseteq \Lambda^*\textsc{Part}(1).\end{aligned}$$

Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)$, we have $\tau \in \mathsf{START}_1$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{START}_1$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{START}_1) = \mathsf{START}_1 = \delta(\mathsf{WRITTEN}, \mathbf{0}_1) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{WRITTEN} \wedge s(\mathsf{C}) = +_n}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{WRITTEN}$ and therefore

$$\begin{aligned}\beta(\tau) &= \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n-1)\textsc{Part}(n)(= \mathbf{S}^*)^+s(\mathsf{C})\\ &= \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n-1)[\mathbf{0}_n\mathbf{S}^*((= \mathbf{S}^*)^+ +_n\mathbf{S}^*)^*](= \mathbf{S}^*)^+ +_n\\ &\subseteq \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n).\end{aligned}$$

Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)$, we have $\tau \in \mathsf{START}_n$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{START}_n$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{START}_n) = \mathsf{START}_n = \delta(\mathsf{WRITTEN}, +_n) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{WRITTEN} \wedge s(\mathsf{C}) = \geq_n}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{WRITTEN}$ and therefore

$$\begin{aligned}\beta(\tau) &= \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)(= \mathbf{S}^*)^+s(\mathsf{C})\\ &= \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)(= \mathbf{S}^*)^+ \geq_n\\ &\subseteq \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)(= \mathbf{S}^*)^* \geq_n\end{aligned}$$

Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)(= \mathbf{S}^*)^* \geq_n$, we have $\tau \in \mathsf{EXIT}_n$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{EXIT}_n$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{EXIT}_n) = \mathsf{EXIT}_n = \delta(\mathsf{WRITTEN}, \geq_n) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{WRITTEN} \wedge s(\mathsf{C}) = \mathbf{S}}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{WRITTEN}$ and therefore

$$\begin{aligned}\beta(\tau) &= \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)(= \mathbf{S}^*)^+s(\mathsf{C})\\ &= \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)(= \mathbf{S}^*)^+\mathbf{S}\\ &\subseteq \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)(= \mathbf{S}^*)^+\end{aligned}$$

Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(n)(= \mathbf{S}^*)^+$, we have $\tau \in \mathsf{WRITTEN}$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{WRITTEN}$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{WRITTEN}) = \mathsf{WRITTEN} = \delta(\mathsf{WRITTEN}, \mathbf{S}) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{WRITTEN} \wedge s(\mathsf{C}) \notin \{\mathbf{0}_1, +_n, \geq_n, \mathbf{S}\}}$: In this case we have $\delta(\alpha(\mathsf{I}), s(\mathsf{C})) = \delta(\mathsf{WRITTEN}, s(\mathsf{C}) = \mathsf{INIT}$ (Fig. 6), hence for any O, $\alpha(\mathsf{O}) \subseteq \mathsf{INIT} = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{EXIT}_k \wedge s(\mathsf{C}) = +_{k-1} \wedge k \in [2..n]}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{EXIT}_k$ and

$$\begin{aligned}\beta(\tau) &= \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k-1)\textsc{Part}(k) \geq_k \mathbf{S}^*s(\mathsf{C})\\ &= \Lambda^*\textsc{Part}(1)\ldots[\mathbf{0}_{k-1}\mathbf{S}^*(\textsc{Tot}(k)\mathbf{S}^* +_{k-1}\mathbf{S}^*)^*]\textsc{Part}(k) \geq_k \mathbf{S}^* +_{k-1}\\ &= \Lambda^*\textsc{Part}(1)\ldots[\mathbf{0}_{k-1}\mathbf{S}^*(\textsc{Tot}(k)\mathbf{S}^* +_{k-1}\mathbf{S}^*)^*]\textsc{Tot}(k)\mathbf{S}^* +_{k-1}\\ &\subseteq \Lambda^*\textsc{Part}(1)\ldots[\mathbf{0}_{k-1}\mathbf{S}^*(\textsc{Tot}(k)\mathbf{S}^* +_{k-1}\mathbf{S}^*)^*]\\ &= \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k-1)\end{aligned}$$

Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k-1)$, we have $\tau \in \mathsf{START}_{k-1}$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{START}_{k-1}$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{START}_{k-1}) = \mathsf{START}_{k-1} = \delta(\mathsf{EXIT}_k, +_{k-1}) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

```
1:  for all C ∈ ℂ do
2:      φ(C) := ∅;
3:  end for
4:  ws := [⟨C_init, INIT⟩];
5:  φ(C_init) := {INIT};
6:  while (!ws.isEmpty()) do
7:      ⟨C, σ♯⟩ := ws.pop();
8:      for all C_1 such that suc(C) = ini(C_1) do
9:          σ♯_1 := δ(σ♯, s(C));
10:         if (σ♯_1 ∉ φ(C_1)) then
11:             ws.push(⟨C_1, σ♯_1⟩);
12:             φ(C_1) := φ(C_1) ∪ {σ♯_1};
13:         end if
14:     end for
15: end while
```

Fig. 8. The ArrayInit algorithm.

$\underline{\alpha(\mathsf{I}) = \mathsf{EXIT}_k \wedge s(\mathsf{C}) = \mathbf{0}_1 \wedge k \in [2..n]}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{EXIT}_k$ and

$$\begin{aligned}\beta(\tau) &= \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k) \geq {}_k\mathbf{S}^*s(\mathsf{C})\\ &= \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k) \geq {}_k\mathbf{S}^*\mathbf{0}_1\\ &\subseteq \Lambda^*\mathbf{0}_1 \subseteq \Lambda^*\textsc{Part}(1)\end{aligned}$$

Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)$, we have $\tau \in \mathsf{START}_1$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{START}_1$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{START}_1) = \mathsf{START}_1 = \delta(\mathsf{EXIT}_k, \mathbf{0}_1) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{EXIT}_k \wedge s(\mathsf{C}) = \mathbf{S} \wedge k \in [2..n]}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{EXIT}_k$ and

$$\begin{aligned}\beta(\tau) &= \beta(\tau')s(\mathsf{C}) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k) \geq {}_k\mathbf{S}^*s(\mathsf{C})\\ &= \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k) \geq {}_k\mathbf{S}^*\mathbf{S}\\ &\subseteq \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k) \geq {}_k\mathbf{S}^*\end{aligned}$$

Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)\ldots\textsc{Part}(k) \geq {}_k\mathbf{S}^*$, we have $\tau \in \mathsf{EXIT}_k$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{EXIT}_k$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{EXIT}_k) = \mathsf{EXIT}_k = \delta(\mathsf{EXIT}_k, \mathbf{S}) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) = \mathsf{EXIT}_k \wedge s(\mathsf{C}) \notin \{\mathbf{0}_1, +_k, \mathbf{S}\}}$: In this case $\delta(\alpha(\mathsf{I}), s(\mathsf{C})) = \delta(\mathsf{EXIT}_k, s(\mathsf{C}) = \mathsf{INIT}$ (Fig. 6), hence for any O, $\alpha(\mathsf{O}) \subseteq \mathsf{INIT} = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) \in \{\mathsf{INIT}, \mathsf{ACCEPT}\} \wedge s(\mathsf{C}) = \mathbf{0}_1}$: In this case $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) \in \{\mathsf{INIT}, \mathsf{ACCEPT}\}$ and therefore $\beta(\tau) = \beta(\tau')s(\mathsf{C}) \in \beta(\tau')\mathbf{0}_1 \subseteq \Lambda^*\mathbf{0}_1 \subseteq \Lambda^*\textsc{Part}(1)$. Since $\beta(\tau) \in \Lambda^*\textsc{Part}(1)$, we have $\tau \in \mathsf{START}_1$ (Definition 20). Since $\tau \in \mathsf{O}$ is arbitrary, we have $\mathsf{O} \subseteq \mathsf{START}_1$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{START}_1) = \mathsf{START}_1 = \delta(\mathsf{INIT}, \mathbf{0}_1) = \delta(\mathsf{ACCEPT}, \mathbf{0}_1) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$.

$\underline{\alpha(\mathsf{I}) \in \{\mathsf{INIT}, \mathsf{ACCEPT}\} \wedge s(\mathsf{C}) \neq \mathbf{0}_1}$: In this case $\delta(\alpha(\mathsf{I}), s(\mathsf{C})) = \delta(\mathsf{INIT}, s(\mathsf{C}) = \delta(\mathsf{ACCEPT}, s(\mathsf{C})) = \mathsf{INIT}$ (Fig. 6), hence for any O, $\alpha(\mathsf{O}) \subseteq \mathsf{INIT} = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$. □

5. The static analysis algorithm

We describe here a static analysis that uses the automaton of the previous section to determine a subset of those commands of the program that are exactly at the end of a loop performing a complete initialization of an array. This subset is in general strict, since identification of completely initialized arrays is undecidable. The analysis, intra-procedural but aware of interprocedural side-effects, is designed for a specific tuple $\langle a, i_1, \ldots, i_n\rangle$ of program variables. Its result lets us compute an under-approximation of the points where an array a of dimension n has just been initialized through n nested loops with index variables $i_1, \ldots, i_n$. In principle, one has to repeat the analysis for every tuple $\langle a, i_1, \ldots, i_n\rangle$; in practice, a tuple $\langle a, i_1, \ldots, i_n\rangle$ is only significant when variables $a, i_1, \ldots, i_n$ occur in an array store action $a[i_1]\ldots[i_n] :=$ ..., which drastically reduces the number of tuples to consider up to an empty or singleton set.

Our analysis is formalized by ArrayInit, the working set-based fixpoint algorithm in Fig. 8. When the working set (ws) becomes empty (line 6), a fixpoint is reached. The algorithm starts by applying the automaton in Fig. 6 from C_{init} and the INIT state (line 4). Commands are read in any order allowed by the labels of the program and then *executed*. While the automaton reads those commands, its state evolves. At each command, the algorithm records the states of the automaton just before the execution of that command. We implement this by a map φ from commands to sets of states, initially empty (line 2) and updated when a new state is found at the beginning of a command (line 12).

Example 9. We show the application of the ArrayInit over the Java fragment and its corresponding transition system given at the top of Fig. 9. In this fragment, we initialize the loop counter i to 0 (line 1), check whether, for an arbitrary integer variable b, $b < 5$ holds (lines 2–3), and then execute $a.length$ iterations of the loop (lines 4–6). Each iteration initializes

```
...
1. i = 0;
2. if (b<5) {
3.    a[i] = 2;
   }
4. while (i< a.length) {
5.    a[i] = 3;
6.    i++;
   }
7. ...
```

C_0	1: $i := 0 \rightarrow 2$;
C_1	2: $b < 5 \rightarrow 3$;
C_2	2: $\neg(b < 5) \rightarrow 4$;
C_3	3: $a[i] := 2 \rightarrow 4$;
C_4	4: $i < a.length \rightarrow 5$;
C_5	4: $\neg(i < a.length) \rightarrow 7$;
C_6	5: $a[i] := 3 \rightarrow 6$;
C_7	6: $i := i + 1 \rightarrow 4$;
C_8	7: $\cdots$

it.	ws	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
0	$\langle C_0, I\rangle$	I	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
1	$\langle C_1, S_1\rangle, \langle C_2, S_1\rangle$	I	S_1	S_1	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
2	$\langle C_2, S_1\rangle, \langle C_3, S_1\rangle$	I	S_1	S_1	S_1	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
3	$\langle C_3, S_1\rangle, \langle C_4, S_1\rangle, \langle C_5, S_1\rangle$	I	S_1	S_1	S_1	S_1	S_1	$\varnothing$	$\varnothing$	$\varnothing$
4	$\langle C_4, S_1\rangle, \langle C_5, S_1\rangle, \langle C_4, W\rangle, \langle C_5, W\rangle$	I	S_1	S_1	S_1	S_1, W	S_1, W	$\varnothing$	$\varnothing$	$\varnothing$
5	$\langle C_5, S_1\rangle, \langle C_4, W\rangle, \langle C_5, W\rangle, \langle C_6, S_1\rangle$	I	S_1	S_1	S_1	S_1, W	S_1, W	S_1	$\varnothing$	$\varnothing$
6	$\langle C_4, W\rangle, \langle C_5, W\rangle, \langle C_6, S_1\rangle, \langle C_8, A\rangle$	I	S_1	S_1	S_1	S_1, W	S_1, W	S_1	$\varnothing$	A
7	$\langle C_5, W\rangle, \langle C_6, S_1\rangle, \langle C_8, A\rangle, \langle C_6, W\rangle$	I	S_1	S_1	S_1	S_1, W	S_1, W	S_1, W	$\varnothing$	A
8	$\langle C_6, S_1\rangle, \langle C_8, A\rangle, \langle C_6, W\rangle$	I	S_1	S_1	S_1	S_1, W	S_1, W	S_1, W	$\varnothing$	A
9	$\langle C_8, A\rangle, \langle C_6, W\rangle, \langle C_7, W\rangle$	I	S_1	S_1	S_1	S_1, W	S_1, W	S_1, W	W	A
10	$\langle C_6, W\rangle, \langle C_7, W\rangle$	I	S_1	S_1	S_1	S_1, W	S_1, W	S_1, W	W	A
11	$\langle C_7, W\rangle$	I	S_1	S_1	S_1	S_1, W	S_1, W	S_1, W	W	A

Fig. 9. A Java loop that fully initializes a and its analysis with ArrayInit.

the ith element of a and increments i by 1. Since lines 2–3 do not affect the initial value assigned to i, it is clear that, at the end of the loop (line 7), the array a will be completely initialized. Let us analyze this fragment by the ArrayInit algorithm. We assume the working set implemented as a queue. Due to space limitations, we write I, S_k, E_k, W and A for INIT, $START_k$, $EXIT_k$, WRITTEN and ACCEPT. Fig. 9 shows the evolution of ws and φ during the iterations. Column C_i stands for the content of $\varphi(C_i)$. Initially, $ws = [\langle C_0, I\rangle]$ with $C_0 = C_{init}$, $\varphi(C_0) = \{I\}$ and φ holds the empty set elsewhere. Then we pop $\langle C_0, I\rangle$ from ws and compute $\delta(I, s(C_0)) = \delta(I, \mathbf{0}_1) = S_1$. Since $suc(C_0) = ini(C_1) = ini(C_2)$, control passes to C_1 and C_2. Since $S_1 \notin \varnothing = \varphi(C_1) = \varphi(C_2)$, we push $\langle C_1, S_1\rangle$ and $\langle C_2, S_1\rangle$ into ws and update φ at C_1 and C_2. Since ws is not empty, the algorithm continues by popping $\langle C_1, S_1\rangle$ from ws and computes $\delta(S_1, s(C_1)) = \delta(S_1, \mathbf{S}) = S_1$. Since $suc(C_1) = ini(C_3)$ and $S_1 \notin \varnothing = \varphi(C_3)$, we push $\langle C_3, S_1\rangle$ into ws and update φ at C_3. The algorithm continues similarly until the working set is empty.

Example 10. Figs. 10 and 11 show other two Java fragments, their corresponding transition systems and the iterations of our algorithm. The former initializes the loop counter i to 0, and then iterates the loop while $i < a.length$ holds. At ith iteration, if i is a multiple of 3, a value is assigned to $a[i]$. Otherwise, we assign both $a[i]$ and $a[i+1]$. At the end of the execution of this fragment, a is completely initialized.[1] On the other hand, the fragment shown in Fig. 11 completely initializes the array a through two loops. They are examples of unusual and non-trivial array initializations, that our algorithm identifies, nevertheless, as complete initializations (continues in Example 12).

Example 11. Suppose that we removed line 7 from the nested loop given at the bottom of Fig. 4. Then, its transition system is similar to that given in the same figure: the only difference is command C_5, which now becomes $4: \neg(j < b[i].length) \rightarrow 8;$. This fragment represents a complete initialization of the matrix b through two nested loops. The outer one is iterated $b.length$ times, and at ith iteration it executes $b[i].length$ iterations of the inner loop. Each of these iterations initializes one of the elements of the matrix b. We analyzed the initialization of matrix b by ArrayInit, and we give the result in Fig. 12.

The following result states that the ArrayInit algorithm computes a correct approximation of the concrete operational semantics of Definition 16.

Proposition 2. *Let* $C \in \mathbb{C}$. ArrayInit *terminates with* $@C \subseteq \cup\varphi(C)$.

Proof. We actually prove a stronger property, which entails the thesis, i.e., we show that, at the end of the algorithm, for every sequence of application of $\Rightarrow$ of the form $\{\epsilon\} \Rightarrow^{C_1} T_1 \Rightarrow^{C_2} T_2 \cdots \Rightarrow^{C_n} T_n$, with $T_n \neq \varnothing$ and $C_1 = C_{init}$, we have $T_n \subseteq \cup\varphi(C_n)$ and, during the execution of the algorithm, there has been a step when a pair $\langle C_n, \sigma^\sharp\rangle$, with $\alpha(T_n) \subseteq \sigma^\sharp$, has been pushed in the working set ws. The thesis follows by Definition 16 for @C.

The above statement is proved by induction on the length $n \geq 1$ of the sequence of applications of $\Rightarrow$.

[1] In this example we assume that the length of a is a multiple of 3.

```
...
1. i = 0;
2. while(i < a.length) {
3.   if(i % 3 == 0) {
4.     a[i] = ...;
   } else {
5.     a[i] = ...;
6.     i++;
7.     a[i] = ...;
   }
8.   i++;
 }
9. ...
```

C_0 1: $i := 0 \to 2$;
C_1 2: $i < a.length \to 3$;
C_2 2: $\neg(i < a.length) \to 9$;
C_3 3: $i\%3 = 0 \to 4$;
C_4 3: $\neg(i\%3 = 0) \to 5$;
C_5 4: $a[i] := \ldots \to 8$;
C_6 5: $a[i] := \ldots \to 6$;
C_7 6: $i := i + 1 \to 7$;
C_8 7: $a[i] := \ldots \to 8$;
C_9 8: $i := i + 1 \to 2$;
C_{10} 9: $\cdots$

it.	ws	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
0	$\langle C_0, I\rangle$	I	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	$\langle C_1, S_1\rangle, \langle C_2, S_1\rangle$	I	S_1	S_1	∅	∅	∅	∅	∅	∅	∅	∅
2	$\langle C_2, S_1\rangle, \langle C_3, S_1\rangle, \langle C_4, S_1\rangle$	I	S_1	S_1	S_1	S_1	∅	∅	∅	∅	∅	∅
3	$\langle C_3, S_1\rangle, \langle C_4, S_1\rangle, \langle C_{10}, A\rangle$	I	S_1	S_1	S_1	S_1	∅	∅	∅	∅	∅	A
4	$\langle C_4, S_1\rangle, \langle C_{10}, A\rangle, \langle C_5, S_1\rangle$	I	S_1	S_1	S_1	S_1	S_1	∅	∅	∅	∅	A
5	$\langle C_{10}, A\rangle, \langle C_5, S_1\rangle, \langle C_6, S_1\rangle$	I	S_1	S_1	S_1	S_1	S_1	S_1	∅	∅	∅	A
6	$\langle C_5, S_1\rangle, \langle C_6, S_1\rangle$	I	S_1	S_1	S_1	S_1	S_1	S_1	∅	∅	∅	A
7	$\langle C_6, S_1\rangle, \langle C_9, W\rangle$	I	S_1	S_1	S_1	S_1	S_1	S_1	∅	∅	W	A
8	$\langle C_9, W\rangle, \langle C_7, W\rangle$	I	S_1	S_1	S_1	S_1	S_1	S_1	W	∅	W	A
9	$\langle C_7, W\rangle$	I	S_1	S_1	S_1	S_1	S_1	S_1	W	∅	W	A
10	$\langle C_8, S_1\rangle$	I	S_1	S_1	S_1	S_1	S_1	S_1	W	S_1	W	A

Fig. 10. A loop initializing the elements of a in different ways and its analysis.

```
...
1. i = 0;
2. while(i < a.length/(i+1)){
3.   a[i] = 7;
4.   if(i > a[i]){
5.     a[i] = i;
     }
6.   i++;
   }
7. while(i < a.length){
8.   a[i] = 10;
9.   i++;
   }
10. ...
```

C_0 1 : $i := 0 \to 2$;
C_1 2 : $i < a.length \div 2 \to 3$;
C_2 2 : $\neg(i < a.length \div 2) \to 7$;
C_3 3 : $a[i] := 7 \to 4$;
C_4 4 : $(i > a[i]) \to 5$;
C_5 4 : $\neg(i > a[i]) \to 6$;
C_6 5 : $a[i] := i \to 6$;
C_7 6 : $i := i + 1 \to 2$;
C_8 7 : $(i < a.length) \to 8$;
C_9 7 : $\neg(i < a.length) \to 10$;
C_{10} 8 : $a[i] := 10 \to 9$;
C_{11} 9 : $i := i + 1 \to 7$;
C_{12} 10: $\cdots$

it.	ws	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}
0	$\langle C_0, I\rangle$	I	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	$\langle C_1, S_1\rangle, \langle C_2, S_1\rangle$	I	S_1	S_1	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
2	$\langle C_2, S_1\rangle, \langle C_3, S_1\rangle$	I	S_1	S_1	S_1	∅	∅	∅	∅	∅	∅	∅	∅	∅
3	$\langle C_3, S_1\rangle, \langle C_8, S_1\rangle, \langle C_9, S_1\rangle$	I	S_1	S_1	S_1	∅	∅	∅	∅	S_1	S_1	∅	∅	∅
4	$\langle C_8, S_1\rangle, \langle C_9, S_1\rangle, \langle C_4, W\rangle, \langle C_5, W\rangle$	I	S_1	S_1	S_1	S_1	S_1	∅	∅	S_1	S_1	∅	∅	∅
5	$\langle C_9, S_1\rangle, \langle C_4, W\rangle, \langle C_5, W\rangle, \langle C_{10}, S_1\rangle$	I	S_1	S_1	S_1	S_1	S_1	∅	∅	S_1	S_1	S_1	∅	∅
6	$\langle C_4, W\rangle, \langle C_5, W\rangle, \langle C_{10}, S_1\rangle, \langle C_{12}, A\rangle$	I	S_1	S_1	S_1	S_1	S_1	∅	∅	S_1	S_1	S_1	∅	A
7	$\langle C_5, W\rangle, \langle C_{10}, S_1\rangle, \langle C_{12}, A\rangle, \langle C_6, W\rangle$	I	S_1	S_1	S_1	S_1	S_1	W	∅	S_1	S_1	S_1	∅	A
8	$\langle C_{10}, S_1\rangle, \langle C_{12}, A\rangle, \langle C_6, W\rangle, \langle C_7, W\rangle$	I	S_1	S_1	S_1	S_1	S_1	W	W	S_1	S_1	S_1	∅	A
9	$\langle C_{12}, A\rangle, \langle C_6, W\rangle, \langle C_7, W\rangle, \langle C_{11}, W\rangle$	I	S_1	S_1	S_1	S_1	S_1	W	W	S_1	S_1	S_1	W	A
10	$\langle C_6, W\rangle, \langle C_7, W\rangle, \langle C_{11}, W\rangle$	I	S_1	S_1	S_1	S_1	S_1	W	W	S_1	S_1	S_1	W	A
11	$\langle C_7, W\rangle, \langle C_{11}, W\rangle$	I	S_1	S_1	S_1	S_1	S_1	W	W	S_1	S_1	S_1	W	A
12	$\langle C_{11}, W\rangle$	I	S_1	S_1	S_1	S_1	S_1	W	W	S_1	S_1	S_1	W	A

Fig. 11. A pair of loops fully initializing an array and their analysis with ArrayInit.

Base case: we have $n = 1$ and the sequence of applications of $\Rightarrow$ becomes $\{\epsilon\} \Rightarrow^{C_1} T_1$ with $C_1 = C_{init}$. By line 5 of the algorithm and since that algorithm never removes states from the codomain of φ, we conclude that at the end of the algorithm we have $T_1 \subseteq \text{INIT} = \cup\varphi(C_{init})$. Moreover, line 4 of the algorithm pushed a pair $\langle C_{init}, \text{INIT}\rangle$ in ws with $\alpha(T_1) \subseteq \text{INIT}$.

Induction: Assume that the result holds for some $n \geq 1$. We prove it for $n + 1$. The sequence of applications of $\Rightarrow$ of length $n + 1$ has the form $\{\epsilon\} \Rightarrow^{C_1} T_1 \Rightarrow^{C_2} T_2 \cdots \Rightarrow^{C_n} T_n \Rightarrow^{C_{n+1}} T_{n+1}$, with $T_{n+1} \neq \varnothing$ and $C_1 = C_{init}$. Since $T_{n+1} \neq \varnothing$, we also have $T_n \neq \varnothing$, by Definition 16 of $\Rightarrow$. By inductive hypothesis applied to the subsequence $\{\epsilon\} \Rightarrow^{C_1} T_1 \Rightarrow^{C_2} T_2 \cdots \Rightarrow^{C_n} T_n$ we conclude that, at the end of the algorithm, we have $T_n \subseteq \cup\varphi(C_n)$ and,

it.	ws	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_9	C_{10}
0	$\langle C_0, I\rangle$	I	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	$\langle C_1, S_1\rangle, \langle C_2, S_1\rangle$	I	S_1	S_1	∅	∅	∅	∅	∅	∅	∅
2	$\langle C_2, S_1\rangle, \langle C_3, S_1\rangle$	I	S_1	S_1	S_1	∅	∅	∅	∅	∅	∅
3	$\langle C_3, S_1\rangle, \langle C_{10}, A\rangle$	I	S_1	S_1	S_1	∅	∅	∅	∅	∅	A
4	$\langle C_{10}, A\rangle, \langle C_4, S_2\rangle, \langle C_5, S_2\rangle$	I	S_1	S_1	S_1	S_2	S_2	∅	∅	∅	A
5	$\langle C_4, S_2\rangle, \langle C_5, S_2\rangle$	I	S_1	S_1	S_1	S_2	S_2	∅	∅	∅	A
6	$\langle C_5, S_2\rangle, \langle C_6, S_2\rangle$	I	S_1	S_1	S_1	S_2	S_2	S_2	∅	∅	A
7	$\langle C_6, S_2\rangle, \langle C_9, E_2\rangle$	I	S_1	S_1	S_1	S_2	S_2	S_2	∅	E_2	A
8	$\langle C_9, E_2\rangle, \langle C_7, W\rangle$	I	S_1	S_1	S_1	S_2	S_2	S_2	W	E_2	A
9	$\langle C_7, W\rangle$	I	S_1	S_1	S_1	S_2	S_2	S_2	W	E_2	A

Fig. 12. Analysis of initialization of the matrix b in Fig. 4.

during the algorithm, there has been a step when a pair $\langle C_n, \sigma^\sharp\rangle$ with $\alpha(T_n) \subseteq \sigma^\sharp$ has been pushed in the working set ws. Since the algorithm ends only when ws becomes empty (line 6), we conclude that pair must have been removed from the working set at some moment, at line 7. Since $T_{n+1} \neq \varnothing$, by Definition 15 we conclude that $\mathsf{suc}(C_n) = \mathsf{ini}(C_{n+1})$. Hence C_{n+1} has been considered in the loop at line 8, the state $\sigma_1^\sharp = \delta(\sigma^\sharp, s(C_n))$ has been computed at line 9 and compared against $\varphi(C_{n+1})$ at line 10. This might have had two outcomes:

- it was the case that $\sigma_1^\sharp \notin \varphi(C_{n+1})$: then line 12 added $\sigma_1^\sharp$ to $\varphi(C_{n+1})$ and it was still there at the end of the algorithm, since no state is ever removed from the range of φ. Then, at the end of the algorithm, by extensivity of α [6] we have

$$
\begin{aligned}
T_{n+1} &\subseteq \alpha(T_{n+1}) \\
&\subseteq \delta(\alpha(T_n), s(C_n)) && \text{[Lemma 2]} \\
&\subseteq \delta(\sigma^\sharp, s(C_n)) && \text{[ind. hypothesis + Lemma 1 } (T_n \neq \varnothing)] \\
&= \sigma_1^\sharp \subseteq \cup\varphi(C_{n+1}).
\end{aligned}
$$

  Moreover, line 11 pushed $\langle C_{n+1}, \sigma_1^\sharp\rangle$ into ws and from $T_{n+1} \subseteq \sigma_1^\sharp$ (just shown above) we have $\alpha(T_{n+1}) \subseteq \alpha(\sigma_1^\sharp) = \sigma_1^\sharp$ since $\sigma_1^\sharp \in \mathcal{A}$.
- it was the case that $\sigma_1^\sharp \in \varphi(C_{n+1})$: then it was still there at the end of the algorithm, since no state is ever removed from the range of φ. Exactly as in the previous case, we can prove that $T_{n+1} \subseteq \cup\varphi(C_{n+1})$ and $\alpha(T_{n+1}) \subseteq \sigma_1^\sharp$. □

We can now prove a result stating that our algorithm can be used to implement a correct array initialization analysis.

Theorem 3. *Consider a program P, variables a (array of dimension n), $i_1, \ldots, i_n$ (indexes) and the automaton in Fig. 6 for a and $i_1, \ldots, i_n$. At the end of the* ArrayInit *algorithm, for every* $C \in \mathbb{C}$ *such that* $\varphi(C) = \{\mathsf{ACCEPT}\}$, *we have that* $\mathsf{ini}(C)$ *is a point of P where all elements of a have been initialized by n nested loops with indexes* $i_1, \ldots, i_n$.

Proof. By Proposition 2, we know that $@C \subseteq \cup\varphi(C) = \{\mathsf{ACCEPT}\}$, i.e., every trace of execution τ leading to C belongs to the language of regular expression $\Lambda^*\textsc{Part}(1) \geq_1 = \Lambda^*\textsc{Tot}(1)$ (Definitions 19 and 20). By Definition 19 we have:

$$
\begin{aligned}
&\mathsf{ACCEPT} = \Lambda^*\textsc{Tot}(1) = \Lambda^*\mathbf{0}_1\mathbf{S}^*(\textsc{Tot}(2)\mathbf{S}^* +_1\mathbf{S}^*)^* \geq_1 = \cdots \\
&\Lambda^*\mathbf{0}_1\mathbf{S}^*\underbrace{(\cdots(}_{n-1}\mathbf{S}^*\mathbf{0}_{n-1}\mathbf{S}^*(\textsc{Tot}(n)\mathbf{S}^* +_{n-1}\mathbf{S}^*)^* \geq_{n-1})^*\cdots)\mathbf{S}^* +_1\mathbf{S}^*)^* \geq_1
\end{aligned}
$$

It means that all traces belonging to state ACCEPT correspond to executions that completely initialize a through n nested loops with indexes $i_1, \ldots, i_n$. For example, given arbitrary values for $i_1, \ldots, i_{n-1}$, regular expression $\textsc{Tot}(n) = \mathbf{0}_n\mathbf{S}^*((=\mathbf{S}^*)^+ +_n\mathbf{S}^*)^* \geq_n$ simulates those executions that completely initialize the array $a[i_1][i_2]\cdots[i_{n-1}]$ through a loop with index variable i_n: it starts with an assignment $i_n := 0$ ($\mathbf{0}_n$), which is followed by an alternation of assignments to $a[i_1]\cdots[i_n]$ ($=$) and unitary increments of i_n ($+_n$) until a comparison $i_n \geq a[i_1]\cdots[i_{n-1}].length$ is found. Between these actions, only safe actions ($\mathbf{S}$) are allowed. By definition of safe actions, they cannot modify the contents of the array a, since those actions do not affect arrays whose elements have a type reachable from that of a. Similarly, we can show that $\textsc{Tot}(n-1)$ simulates those executions which completely initialize the array $a[i_1]\cdots[i_{n-2}]$ through 2 nested loops with index variables i_{n-1} and i_n, and so on. □

Example 12. From Fig. 9 we know that $\varphi(C_8) = \{\mathsf{ACCEPT}\}$ at the end of the algorithm. By Theorem 3, we conclude that the array has been completely initialized when program point $7 = \mathsf{ini}(C_8)$ is reached. The same is proved for program points $9 = \mathsf{ini}(C_{10})$ in Fig. 10, $10 = \mathsf{ini}(C_{12})$ in Fig. 11 and $9 = \mathsf{ini}(C_{10})$ in Fig. 12.

```
x.f = this;                         ||   void foo(x) {
int i = 0;                          ||     y = x.f;
while (i<this.a.length){            ||     y.a = new T[...];
  this.a[i++] = 2;                  ||   }
  foo(x);                           ||
}                                   ||
```

Fig. 13. Side-effects hinder the full initialization of `this.a`.

6. Dealing with implicit upper bounds and side-effects

Sections 4 and 5 introduce a static analysis that determines where the elements of a *local variable* of array type are all initialized. The analysis is limited in many ways. Namely, it identifies the comparison between the loop index variable i and the size of the array a in a syntactical, explicit way (Definition 17): it must have the form $\neg(i < a.length)$. For instance, this is not the case in Fig. 1 and the analysis would fail there. Moreover, it works only for arrays held in local variables. Again, this is not the case in Fig. 1, where they are stored into *instance variables*, that is, fields. In that case, the analysis would require careful attention to side-effects. For instance, in the program in Fig. 13, method `foo()` recreates the array at each iteration. At the end of the loop, none of its elements is initialized. This problem might arise when one calls *non-pure* methods such as `foo()` (i.e., methods with side-effects). Then, a naif extension of our analysis to the case of arrays held in fields might easily turn out to be unsound. We discuss below how we overcome these two limitations.

6.1. Implicit upper bounds

We consider the most frequent implicit ways of expressing the upper bound of an array. Often, a variable is used, as `numpoints` in Fig. 1. In order to prove that this variable holds the length of the array, we use the *definite expression aliasing* [19] static analysis available in Julia. It is a version of the traditional available expression analysis [2] for bytecode: bindings of variables to expressions are *generated* by assignments, that also *kill* other bindings that use the old value of the variables. In Fig. 1, the binding `numpoints = this.mOriginalPoints.length` is generated by the first `new` statement and never killed later, since there is no subsequent statement modifying `this`, `numpoints` or `this.mOriginalPoints`. Similarly for the binding `numpoints = this.mRotatedPoints.length`. Definition 17 is consequently modified: when A is $\neg(i < var)$, its abstraction is $\geq$ whenever the definite expression aliasing information contains, there, the binding $var = a.length$ (a can be a local variable or a field).

Another frequent case consists in using a numerical constant as upper bound. This includes the case when a final static integer field is used (i.e., a symbolic constant) since compilers usually replace such fields with their numerical value and Julia analyzes the compiled bytecode. This case is more complex than it looks: we must be sure that the *same* constant has been used to specify the length of the array *wherever* it is created, hence also outside the method where the initialization loop occurs. Moreover, there might be more creation points for the objects stored inside the same variable and that condition must hold for all of them. Here, we exploit the *creation point analysis* available in Julia: for each variable at a given program point or field, it over-approximates the set of program points where its content might have been created. This is similar to class analysis [21], but more concrete information is kept: the creation points rather than just the types of the objects created. Definition 17 is modified again: when A is $\neg(i < con)$ and con is a numerical constant, its abstraction is $\geq$ whenever all creation points for a (the variable being initialized) have the form `new T[`*con*`]`.

6.2. Side-effects

When the array being initialized is stored in a field, as in $x.field$, we must strengthen the notion of safe action A (case **S** of Definition 17). Namely, we must also require that A does not modify *field*. The only actions that might modify *field* are explicit assignments to $y.field$, for any y, and calls to non-pure methods (the latter are not in Fig. 3, but naturally a real language includes them). Here, we use the *side-effects analysis* provided by Julia: for each method call, it over-approximates the set of fields modified during the execution of the callee(s) (and of the methods that the callees invoke, recursively). By checking if *field* belongs to that over-approximation, we determine if the call must be conservatively considered as unsafe. This means that our analysis works also when non-pure methods are called inside the initialization loop, as long as they do not affect the specific field being initialized.

7. Inferring global invariants

At this point, we are able to prove the *local* property `local` of Section 1. What we want now is an automatic technique to lift `local` to the *global invariant* `global`, which is our actual goal: *all elements of fields `mOriginalPoints` and `mRotatedPoints` contain non-`null` elements wherever they are read* (not just inside `readModel()`). Let us generalize `global`: *all elements of a field f, of array type, satisfy a condition X wherever they are read.* We split `global` in two simpler subproblems, that entail `global` and that we prove separately:

1. P_1: each element of f is never read before being written at least once;
2. P_2: only values satisfying X are assigned to the elements of f.

P_1 and P_2 predicate over the whole program, not just over the code that initializes the array (`readModel()` in our example).

In the following, we will show how P_1 and P_2 are proved. We observe for now that P_1 is proved by showing that any execution of any constructor of the class C defining f must initialize all elements of f, before it ever reads them. Since a static constraint of Java bytecode imposes that objects must be always initialized by a constructor before being used, this is enough to conclude that the elements of f are fully initialized *immediately after* the creation of its holder object and, because of P_2, they must be initialized to values satisfying X. Of course, this situation might later change if the elements of f are later reassigned by the code of C of by the code of other classes (when f is not `private` in C). But, again, condition P_2 ensures that the full initialization of the elements of f to values satisfying X is never lost along the lifespan of its holder object. In principle, one might identify a field from its name f, with no reference to its instantiation context. This might be imprecise if the field is `public`. Hence, the actual implementation distinguishes fields on the basis of their instantiation context (the creation point of their holder object) rather than just by their name. For simplicity, we do not describe this improvement here and we assume that fields are identified by their name only.

Note that the knowledge about the fact that, at a given program point, we write values satisfying the property X comes in general from an interprocedural static analysis. In our experiments in Section 8, this will be an interprocedural nullness analysis, which is independent from our (intraprocedural and local) array initialization analysis and is performed after it.

7.1. Proving P_1

This problem is similar to that faced in [24]: finding the so-called *candidate* fields, that are never read before being assigned. An algorithm is given there and proved correct. It performs a data-flow analysis from the constructors of each class and collects the fields that are *definitely* assigned before being *possibly* read. This analysis is inter-procedural and hence considers also fields initialized in *helper functions* that are often used to support the constructors. For instance, `readModel()` in Fig. 1 is a helper function used by the constructor of the class where it occurs. Since the analysis starts from the constructors, it does not consider methods unreachable from them. In [24], an assignment to f is an operation of the form $x.f = \ldots$ where x is a *definite* alias of `this` and a read of f is an operation of the form $x.f$ that does not modify f and where x is a *possible* alias of `this`. Note the different directions of approximation, that are essential to prove the algorithm correct. The same algorithm can be applied here if we use a different notion of read and write to f: in this paper, we assume that an assignment to the elements of f is the exit program point of a loop that *definitely* completely initializes `this.`f; this is checked through the technique of Sections 4 and 5. A read of the elements of f is any operation of the form $\ldots = x[\ldots]$ where x is a *possible* alias of `this.`f; this is checked through the same creation point analysis of Section 6.1. We rephrase that algorithm here for our purposes.

Definition 21 (*Candidate Field*)**.** A field f defined in class κ is *candidate* if it has array type and for every execution path x in every constructor of κ, that ends at a `return` instruction, x passes through a program point where all elements of `this.`f are definitely initialized. Moreover, if an execution path x in a constructor of κ passes through a program point that might read an element of `this.`f, then it passed before through a program point where all elements of `this.`f are definitely initialized.

The algorithm `candidates(c)` yields the set of candidate fields *w.r.t.* a given constructor `c`. A constructor is just the set of its commands $\mathsf{L}_1 : \mathsf{A} \to \mathsf{L}_2$; with a distinguished initial command. If class κ has constructors `c1`…`cn`, the intersection of `candidates(c1)`…`candidates(cn)` is a set of candidate fields from class κ. The algorithm computes, for every command a, sets $a.w$ and $a.r$. The former is the set of fields of array type of `this` whose elements are all definitely written in every execution path starting at a; the latter is the set of fields of κ of array type such that one of their elements might be read in some execution path starting at a before all the elements of that field have been definitely written.

```
Set candidates(Constructor c) {
  \\ all commands reachable from the first command of c are
  \\ added to a workset. Helper functions are also included
  Set ws = reachable(c.firstCommand(), new Set());
  for (Command com: ws)
    { com.w = new Set(); com.r = new Set(); }

  \\ we process the workset
  while (!ws.isEmpty()) {
    remove some com from ws;
    let com1..comn be the successors of com;

    \\ a field is read if it is read by some path
    com.r = com1.r union ... union comn.r;

    \\ a field is written if it is written by all paths
```

```
    if (n > 0) com.w = com1.w intersect ... intersect comn.w;

    if (com might read an element of a field f of this) {
      com.r.add(f); \\ this field is read
      com.w.remove(f);
    }
    else if (com definitely writes all the elements
                           of the field f of this) {
      com.r.remove(f);
      com.w.add(f); \\ this field is written
    }
    else if (com is a call M whose receiver is this) {
      let M1...Mn be the methods that might be called here;
      \\ we continue inside the helper function(s)
      Set r = union Mi.firstCommand().r over i
      Set w = intersection Mi.firstCommand().w over i
      com.w.removeAll(r);
      com.w.addAll(w);
      com.r.removeAll(w);
      com.r.addAll(r);
    } else if (com is a call M) {
      \\ this is not a helper function
      let M1...Mn be the methods that might be called here;
      Set r = all fields of array type read by some Mi
                           or any method that Mi calls;
      com.w.removeAll(r);
      com.r.addAll(r);
    }

    if (com.w or com.r changed during this iteration)
      add all predecessors of com to ws;
  }

  return c.firstCommand().w;
}

\\ yields the set of commands reachable from com.
\\ Helper functions are included
Set reachable(Command com, Set result) {
  if (!result.contains(com)) {
    result.add(com);

    for (Command f: com.successors()) reachable(f, result);

    if (com contains a call M whose receiver is an alias of local 0)
      for each M1...Mn, the methods that might be called here
        reachable(Mi.firstCommand(), result);
  }

  return result;
}
```

The algorithm uses a working set of commands, those reachable from the beginning of the constructor by following helper functions as well. Their initial approximation is empty. The working set is analyzed until it is empty. Every time a command `com` is extracted from the working set, we compute the union of the fields that might be read by its successors and the intersection of the fields that are definitely written by its successors. Those sets are added to `com.`r and to `com.`w, respectively. Then `com` is considered. It might be a command that might read an element of a field of array type or a command that definitely writes all elements of a field of array type. It might also be a call over `this`, that is, a *helper function* used to initialize the constructed object; in that case we compute the fields r whose elements are read by some called method and the fields w whose elements are all written by all called methods. Set r is removed from `com.`w and added to `com.`r and set w is added to `com.`w and removed from `com.`r. For the other calls, we conservatively add to `com.`r all fields whose elements might be read by the method(s) that it calls. Every time that the approximation of a command changes, its predecessors are added to the working set. If `com` is the beginning of a helper function, by *predecessors* we mean the commands that call the helper function.

In Fig. 1, the only assignment to the elements of `mRotatedPoints` is hence program point * and there are no reads of the elements of that field (reachable from the constructor): the data-flow algorithm concludes that the elements of `mRotatedPoints` are never read before being assigned. For field `mOriginalPoints`, instead, the only assignment to its elements is point * again but there are three operations reading its elements, inside the same initialization loop in `readModel()`: the data-flow algorithm does *not* conclude that the elements of `mOriginalPoints` are never read before being assigned, which is a conservative but imprecise result. Indeed, the three reads from `this.mOriginalPoints[i]` access exactly the element that has been initialized two lines before to a new `ThreeDPoint`. This situation is frequent and we must cope with it. Namely, we do not consider anymore a read operation of $x.f[i]$ as such when the analyzer is able to find an assignment to the same element that dominates that operation and such that no side-effect to $x.f[i]$ occurs in the middle, as is the case in Fig. 1. We implemented this check as a backward syntactical check that uses the same side-effects analysis exploited in Section 6.2. With this improvement, our technique concludes that also the elements of `mOriginalPoints` are never read before being assigned, that is, both `mRotatedPoints` and `mOriginalPoints` can be considered as candidate fields.

7.2. Proving P_2

We can apply here any static analysis for the property X and check its outcome wherever an element of f is assigned. For instance, when X is *being non-*`null`, we can use the nullness analysis in [24]. Two points deserve careful consideration, though. The first is that assignments to elements of f are not always syntactically apparent in code: one can modify the elements of the array held in field f by writing into a data structure that shares with the object holding f, rather than by writing into f itself. Hence, we compute an over-approximation W_f^u of the set W_f of program points where an element of f is assigned: W_f^u contains the program points where an array write operation $a[\ldots] = \ldots$ occurs and a and f share at least a creation point. Here, we use the same creation point analysis of Section 6.1. The second point is that we are inferring a global invariant to improve the precision of the static analysis for X and we are, at the same time, using the results of that analysis to infer the global invariant. This cycle is broken through an *oracle-based* technique: we start with an oracle $O = \cup\{f \mid f \text{ is a field of array type}\}$. We assume, optimistically, that only values satisfying X are written inside the elements of the fields in O. We perform the analysis for X using this (probably initially unsound) hypothesis. For every field f in O, we check the results of the analysis to see if only values satisfying X are written in f. If this is not the case, we drop f from O. We restart the analysis for X, recheck and shrink O again. This process is iterated until a fixpoint oracle. Then we conclude that inside the elements of the fields in the fixpoint oracle only values satisfying X are written. Note that the array initialization analysis is performed before this fixpoint iteration and only once: the iteration concerns the analysis for X, which is performed after our array initialization analysis. This technique is not new: it was defined and proved correct in [24] but has never been applied before to fields of array type. Note that it can be highly optimized through caches to keep small the cost of the iterated static analyses. Moreover, our nullness analysis already performs this oracle-based iteration for the fields of reference type [24], so the inference of global invariants about fields of array type does not introduce any extra cost.

Let us hence define an *oracle*.

Definition 22 (*Oracle*)**.** An *oracle* is a set of candidate fields. The set of oracles is $\mathbb{O}$. An oracle $O \in \mathbb{O}$ is *correct* if, for every $f \in O$, only values that satisfy X are written inside the elements of f.

Given an oracle O, we parameterize the static analysis for X by assuming that only values satisfying X are written inside the elements of the fields in O. This hypothesis can be exploited to improve the precision of the static analysis for X parameterized *w.r.t.* O but, of course, we have no guarantee that static analysis yields correct results. This depends on the choice of O. But an interesting result states that the analysis for X, parameterized *w.r.t.* O, is correct if O is correct.

Proposition 3. *If $O \in \mathbb{O}$ is correct, then the analysis for X parameterized w.r.t. O is correct.*

Proof. We assume that the non-parameterized analysis for X is sound. The only difference between that analysis and its parameterized version is that the latter can exploit the fact that a value satisfying X is written inside the elements of the fields in O. But since O is correct, that assumption is sound and the resulting analysis, parameterized *w.r.t.* O, is consequently sound. □

If O is not correct, it is actually the case that the analysis for X parameterized *w.r.t.* O is correct *w.r.t.* a non-standard semantics parameterized *w.r.t.* O, that forces the value of the fields in O to be arrays of elements satisfying X. This non-standard semantics is distinct from the standard one only for the definition of the evaluation of $\mathsf{E}.f$:

$$\mathcal{A}[\![\mathsf{E}.f]\!]e = \begin{cases} arr & \text{if } \ell = \mathcal{A}[\![\mathsf{E}]\!]e \in \mathbb{L}, o = \mu_o(\ell) \in \mathsf{Obj}, a = \mu_a((o.\phi)(f)) \in \mathsf{Arr}, \\ & f \in O \text{ and } a \text{ has elements not satisfying } X \\ a & \text{otherwise, if } \ell = \mathcal{A}[\![\mathsf{E}]\!]e \in \mathbb{L}, o = \mu_o(\ell) \in \mathsf{Obj} \text{ and} \\ & a = \mu_a((o.\phi)(f)) \in \mathsf{Arr} \\ \mathsf{undefined} & \text{otherwise} \end{cases} \quad (1)$$

where *arr* is an array of values satisfying X. This array can always be constructed since there must be at least a value satisfying X, or otherwise the static analysis for X is vacuously useless.

Lemma 4. *The static analysis for X parameterized w.r.t. O is sound w.r.t. the non-standard semantics parameterized w.r.t. O.*

Proof. Over expressions distinct from E.f and over all actions, the standard and non-standard semantics coincide and, on the other hand, the static analysis for X parameterized *w.r.t.* O and the non-parameterized static analysis for X coincide. Since the non-parameterized static analysis for X is assumed sound, the same must hold for the parameterized static analysis for X over those expressions and actions. If $f \notin O$, the standard and non-standard semantics of E.f coincide and, on the other hand, the static analysis for X parameterized *w.r.t.* O and the non-parameterized static analysis for X coincide for E.f. Since the non-parameterized static analysis for X is assumed sound, the same must hold for the parameterized static analysis for X over the expression E.f. Assume now instead that $f \in O$. The static analysis for X parameterized *w.r.t.* O assumes that the value of E.f has only elements that satisfy X. But this is exactly how the non-standard semantics for E.f is defined (Eq. (1)). We conclude that, also in this case, the static analysis for X parameterized *w.r.t.* O is correct *w.r.t.* the non-standard semantics for E.f parameterized *w.r.t.* O. □

The problem now is to find a correct $O \in \mathbb{O}$. The obvious choice $O = \emptyset$ is correct but leads to a static analysis for X that assumes that a value satisfying X is never written inside a field of array type. This is too conservative. The following result will help us.

Proposition 4. *Define* $F : \mathbb{O} \to \mathbb{O}$ *as*

$$F(O) = \left\{ f \in O \;\middle|\; \begin{array}{l} \textit{the analysis for } X \textit{ parameterized w.r.t. } O \textit{ proves that only} \\ \textit{values satisfying } X \textit{ are written inside the elements of } f \end{array} \right\}.$$

If O is a fixpoint of F then O is correct.

Proof. Let $O \in \mathbb{O}$ be a fixpoint of F and assume, by contradiction, that O is not correct. Hence $I = \{f \in O \mid$ a value that does not satisfy X is written inside an element of $f\}$ is not empty. Hence there is a finite execution x of the program that leads to an assignment to an element of the array held in a field f and that assignment writes a value that does not satisfy X. We can assume, without loss of generality, that this never happened before, in x, for any of the fields in I (i.e., we end x where, for the first time, a value that does not satisfy X is written inside an element of an array held in a field f). Consider an operation that reads an element of a field g, executed during x. By the choice of x, if $g \in O \setminus I$ then g must hold an array whose elements all satisfy X and the operation reads hence a value that satisfies X. If instead $g \in I$ then g is candidate and, by Definition 21, all elements of the array held in g must have been already initialized. By the hypothesis about x, we conclude that also in this case the operation reads a value that satisfies X. In conclusion, all operations that read an element from an array held in the fields in O always read a value that satisfies X i.e., they behave accordingly to the non-standard semantics of Eq. (1). This means that the execution x is also a non-standard execution that uses the semantics of Eq. (1). Since the static analysis parameterized *w.r.t.* O is correct *w.r.t.* the non-standard semantics parameterized *w.r.t.* O (Lemma 4), we conclude that it cannot prove that only values satisfying X are written inside f, since x writes a value not satisfying X inside f. Then $f \notin F(O)$ and $O \neq F(O)$, a contradiction. We conclude that O must be correct. □

By computing $F(O)$, one applies the static analysis for X parameterized *w.r.t.* O and checks in which fields of O the program writes only values satisfying X. By definition, $F(O) \subseteq O$. Take O_0 equal to the set of all candidate fields and compute $O_1 = F(O_0)$. If $O_1 = O_0$ then O_0 is correct (Proposition 4); otherwise $O_0 \supset O_1$ and compute $O_2 = F(O_1)$; again, if $O_2 = O_1$ then O_1 is correct; otherwise $O_1 \supset O_2$ and compute $O_3 = F(O_2)$ and so on. Since the number of candidate fields of P is finite, the decreasing chain $O_0 \supset O_1 \supset O_2 \supset O_3 \supset \ldots$ must be finite and converge to a correct oracle (a greatest fixpoint of F). In words, one starts with the optimistic hypothesis that all candidate fields hold arrays whose elements satisfy X and iteratively removes those fields for whose elements one has no proof of satisfying X. When no more fields are removed, one gets a correct oracle and the last iteration of the analysis is correct (Proposition 3). For instance, in Fig. 1, we start with an oracle O that contains both `mOriginalPoints` and `mRotatedPoints` (together with other fields that are not shown in that figure). If we consider the property of *being non-`null`* as X, the nullness analysis in [24] proves that only values satisfying X are written inside those fields: this is clear in Fig. 1 (`new ThreeDPoint()` is obviously non-`null`) and is true also in other program points that we do not show. Hence those two fields are never removed from O and belong to the final fixpoint oracle.

8. Experiments

We have implemented our analysis inside the Julia analyzer [1]. Experiments were performed on a quad-core Intel Xeon 64 bits machine running at 2.66GHz, with 8GB of RAM, Linux 2.6.27 and Sun jdk 1.6. We analyzed some real-life benchmarks. Some of these benchmarks are Java programs: JFlex is a lexical analyzers generator[2]; Plume is a library by Michael D. Ernst[3]; Nti is a non-termination analyzer by Étienne Payet.[4] We also analyzed some Android applications: ChimeTimer, Dazzle,

---

2 http://jflex.de.

3 http://code.google.com/p/plume-lib.

4 http://personnel.univ-reunion.fr/epayet/Research/NTI/NTI.html.

NAME	LOC	TOTAL LOC	ARRAY INITIALIZATION			TOTAL TIME
			TOTAL	DETECTED	TIME	
AbdTest	489	56334	1	1	2.36	121.73
AccelerometerPlay	306	46854	1	1	0.35	71.99
CubeWallpaper	370	25654	3	3	0.12	28.51
HoneycombGallery	948	71501	1	0	1.06	157.85
TicTacToe	607	59040	3	3	0.70	102.65
Snake	420	57075	1	0	0.36	117.49
Real3D	1228	74384	2	2	1.06	177.95
ChimeTimer	4095	95781	9	7	0.80	383.45
Dazzle	4376	100271	4	1	1.02	394.44
OnWatch	9746	113368	10	6	2.91	525.15
Tricorder	10410	106100	17	11	1.01	467.58
TestAppv2	377	58365	1	1	0.38	102.34
TxWthr	2024	74441	7	1	0.42	179.78
JFlex	7681	40872	7	6	1.35	72.46
nti	2372	13098	4	4	0.09	13.55
plume	8587	43302	24	21	1.19	113.07

Fig. 14. Experiments with our array initialization analysis.

OnWatch and Tricorder[5]; TxWthr[6]. The others are sample programs taken from the Android 3.1 distribution by Google. This means that we have analyzed non-trivial libraries as well as complete Java and Android applications. Most of the Android applications are games or libraries or demonstrations of the use of the standard Android API. Those programs contain simple as well as nested loops and we analyze both kinds of loops.

Fig. 14 shows that our array initialization analysis is fast and precise. We explain the meaning of different columns:

- LOC is the number of non-blank, non-comment source program lines reached and hence analyzed by Julia; this information is recovered from the debug information generated by the compiler;
- TOTAL LOC is the total number of analyzed lines, including `java.*` and `android.*` libraries;
- TOTAL is the number of reachable loops in those programs (libraries excluded) that fully initialize an array, computed by manual check;
- DETECTED is the number of these initializing loops that our analysis successfully spot as complete initializations of arrays. In principle, for the most precise static analysis we have DETECTED=TOTAL;
- TIME is the runtime in seconds of our array initialization analysis;
- TOTAL TIME is the runtime in seconds of the nullness analysis of Julia: it includes parsing of the class files, preprocessing, aliasing, sharing, creation points, expression aliasing and side-effects analyses.

Fig. 15 shows that our array initialization analysis is useful to a client analysis. In particular, it considers those programs from Fig. 14 that contain at least a loop initializing an array of reference type, since otherwise the array initialization analysis would be irrelevant for the nullness tool of Julia. The figure reports the number of null-pointer warnings with (ArrInit column) and without (~~ArrInit~~ column) the analysis introduced in this paper. In the former case, the precision of the nullness analysis is improved by 8.48% on average on these programs and its cost is only 0.47% higher (compare TIME and TOTAL TIME in Fig. 14). Actually, the total time is not affected by the presence of the extra array initialization analysis, since we perform it in a separate thread and its results are available well before they are needed by the client nullness analysis. Moreover, those results improve the precision of the nullness analysis, but never induce a reduction in the number of its fixpoint iterations, at least in our experiments.

When Julia fails to spot complete array initialization, the problem is related to the weaknesses of the supporting analyses of our array initialization analysis rather than to those of the latter. For instance, there is a complete array initialization in `HoneycombGallery` that Julia fails to spot (Fig. 14). Here it is:

5 http://moonblink.googlecode.com/svn/trunk/
6 http://typoweather.googlecode.com/svn/trunk/

NAME	Nullness Warnings	
	ArrInit	~~ArrInit~~
AccelerometerPlay	3	6
ChimeTimer	33	36
CubeWallpaper	0	3
TicTacToe	0	2
JFlex	57	65
OnWatch	82	85
Real3D	19	19
Tricorder	107	121
TxWthr	48	49
nti	15	15
plume	57	59

Fig. 15. Effects of our array initialization analysis on the number of warnings produced by Julia's nullness tool.

```
String[] items = new String[cat.getEntryCount()];
for (int i = 0; i < cat.getEntryCount(); i++)
  items[i] = cat.getEntry(i).getName();
```

The loop upper bound is `cat.getEntryCount()`, which does not fall in the cases considered in Section 6. The use of a method call as loop upper bound is problematic since the definite expression aliasing analysis must be able to prove that the value of `cat.getEntryCount()` is constant between the creation of the array and the check of the loop upper bound, also when the loop body has side-effects, as here.

9. Conclusion

We have described a new abstract interpretation that detects fully initialized arrays held in local variables or fields. In the case of fields, we have shown how local complete initialization can be lifted to a global invariant, much more useful to client analyses. Our implementation shows the efficiency of the analysis and its effectiveness to support a client nullness analysis. It is worth observing that our array initialization analysis is not tailored to nullness, but can instead support any other client analysis.

An interesting consideration is that provably sound and precise array analyses and related global invariants can be computed only over a complex static analysis infrastructure, since they are built on previous creation points and side-effects global static analyses. They are all available in Julia and are now highly optimized and debugged after years of use.

References

[1] Julia static analyzer: www.juliasoft.com.

[2] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, 1986.

[3] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, Path Invariants, in: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, PLDI 2007, ACM, 2007, pp. 300–309.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, L. Miné, D. Monniaux, X. Rival, Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter, in: The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, in: Lecture Notes in Computer Science, vol. 2566, Springer, 2002, pp. 85–108.

[5] Aaron R. Bradley, Zohar Mann, Henny B. Sipma, What's decidable about arrays? in: Proceedings of the 7th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2006, Lecture Notes in Computer Science, vol. 3855, Charleston, South Carolina, USA, 2006, pp. 427–442.

[6] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Proceedings of the 6th Symposium on Principles of Programming Languages, POPL 1979, ACM, 1979, pp. 269–282.

[7] P. Cousot, R. Cousot, Systematic design of program transformation frameworks by abstract interpretation, in: Proceedings of the 29th Symposium on Principles of Programming Languages, POPL 2002, ACM, 2002, pp. 178–190.

[8] Patrick Cousot, Radhia Cousot, Francesco Logozzo, A parametric segmentation functor for fully automatic and scalable array content analysis, in: Proceedings of the 38th Symposium on Principles of Programming Languages, POPL 2011, ACM, New York, NY, USA, 2011, pp. 105–118.

[9] C. Flanagan, S. Qadeer, Predicate abstraction for software verification, in: Proceedings of the 29th Symposium on Principles of Programming Languages, POPL 2002, ACM, 2002, pp. 191–202.

[10] Denis Gopan, Thomas Reps, Mooly Sagiv, A framework for numeric analysis of array operations, in: Proceedings of the 32nd annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2005, ACM, New York, NY, USA, 2005, pp. 338–350.

[11] S. Graf, H. Saïdi, Construction of Abstract State Graphs with PVS, in: Proceedings of the 9th International Conference on Computer Aided Verification, CAV 1997, in: Lecture Notes in Computer Science, vol. 1254, Springer, 1997, pp. 72–83.

[12] Peter Habermehl, Radu Iosif, Tomás Vojnar, What else is decidable about integer arrays? in: Proceedings of the 11th International Conference on Foundations of Software Science and Computational Structures, FOSSACS 2008, in: Lecture Notes in Computer Science, vol. 4962, Springer, Budapest, Hungary, 2008, pp. 474–489.

[13] Nicolas Halbwachs, Mathias Péron, Discovering properties about arrays in simple programs, in: Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI 2008, ACM, New York, NY, USA, 2008, pp. 339–348.

[14] Krystof Hoder, Laura Kovács, Andree Voronkov, Case studies on invariant generation using a saturation theorem prover, in: Proceedings of the 10th Mexican International Conference on Artificial Intelligence, MICAI 2011, in: Lecture Notes in Computer Science, vol. 7094, Springer, 2011, pp. 1–15.

[15] Laura Kovács, Andrei Voronkov, Finding loop invariants for programs over arrays using a theorem prover, in: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering, FASE 2009, in: Lecture Notes in Computer Science, vol. 5503, Springer, 2009, pp. 470–485.

[16] Đurica Nikolić, Fausto Spoto, Definite expression aliasing in java bytecode programs: a constraint-based static analysis (submitted for publication). http://profs.sci.univr.it/~nikolic/download/ICTAC2012/ICTAC2012Ext.pdf.
[17] Đurica Nikolić, Fausto Spoto, Reachability analysis of program variables (submitted for publication). http://profs.sci.univr.it/~nikolic/download/IJCAR2012/IJCAR2012Ext.pdf.
[18] Đurica Nikolić, Fausto Spoto, Automaton-based array initialization analysis, in: Proceedings of the 6th International Conference on Language and Automata Theory and Application, vol. 7183, LATA 2012, Springer, 2012, pp. 420–432.
[19] Đurica Nikolić, Fausto Spoto, Definite expression aliasing analysis for java bytecode, in: Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing, ICTAC 2012, in: Lecture Notes in Computer Science, vol. 7521, Springer, 2012, pp. 74–89.
[20] Đurica Nikolić, Fausto Spoto, Reachability analysis of program variables, in: Proceedings of the 6th International Joint Conference on Automated Reasoning, IJCAR 2012, in: Lecture Notes in Artificial Intelligence, vol. 7364, Springer, 2012, pp. 423–438.
[21] J. Palsberg, M.I. Schwartzbach, Object-oriented type inference, in: Proceedings of Object-Oriented Programming, Systems, Languages & Applications, OOPSLA 1991, in: ACM SIGPLAN Notices, vol. 26(11), ACM, 1991, pp. 146–161.
[22] É Payet, F. Spoto, Static analysis of android programs, Information & Software Technology 54 (11) (2012) 1192–1201.
[23] Stefano Secci, Fausto Spoto, Pair-sharing analysis of object-oriented programs, in: Proceedings of the 12th International Static Analysis Symposium, SAS 2005, in: Lecture Notes in Computer Science, vol. 3672, Springer, 2005, pp. 320–335.
[24] F. Spoto, Precise null-pointer analysis, Software and Systems Modeling 10 (2) (2011) 219–252.
[25] Fausto Spoto, The nullness analyser of Julia, in: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning — LPAR (Dakar), in: Lecture Notes in Computer Science, vol. 6355, Springer, 2010, pp. 405–424.
[26] Fausto Spoto, Thomas P. Jensen, Class analyses as abstract interpretations of trace semantics, ACM Transactions on Programming Languages and Systems 25 (5) (2003) 578–630.