Automaton-Based Array Initialization Analysis

Đurica Nikolić and Fausto Spoto

Dipartimento di Informatica, Università di Verona, Italy
`{durica.nikolic,fausto.spoto}@univr.it`

Abstract. We define an automaton-based abstract interpretation of a trace semantics which identifies loops that definitely initialize all the elements of an array, a useful piece of information for the static analysis of imperative languages. This results in a fully automatic and fast analysis, that does not use manual code annotations. Its implementation inside the Julia analyzer is efficient and precise.

1 Introduction

This work was born from a problem faced during the static analysis of Java and Android programs. Fig. 1 shows an example: fields `mOriginal` and `mRotated` hold arrays, initialized by `readModel()` and then read and dereferenced by other methods. In this case the `null`-pointer analysis of Julia [12] issued spurious warnings since it could not prove that the elements of these arrays were initialized.

We wanted to prove, automatically, that *all elements of fields `mOriginal` and `mRotated` are initialized at point **. Complete initialization of arrays to some value is undecidable [2,9], but static analysis can often prove it. Typically, theorem proving or predicate abstraction are used [7,8]. However, not all techniques are automatic; some require a previous manual annotation of the program with invariants; others must be instantiated with different abstract domains (or predicate abstractions), depending on the specific program at hand; others have an overwhelming cost. An impressive evaluation of those techniques has been done in [6]: its authors implemented and compared them, with the result that very few are completely automatic and none efficient for real large software. Moreover, they present a new technique based on abstract interpretation [3,4]: a fixpoint over an abstraction of arrays into *segments* with strict or non-strict bounds. The abstraction of the elements of a single segment is given over an abstract domain, left parametric.

Our contributions are:

1. a new automaton-based abstract interpretation of execution traces, proving that all elements of an array are definitely initialized at a program point;
2. a proof of correctness for the previous analysis;
3. experiments showing that the analysis is efficient (1 second on average for large software) precise and scales to the analysis of more than $100,000$ lines.

Our static analysis is intraprocedural, but aware of interprocedural side-effects. It is a whole-program analysis, hence it does not apply to classes in isolation nor to libraries. We give definitions and proofs for an automaton spotting the full

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

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

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

initialization of monodimensional arrays, with a standard pattern of initialization from element 0 upwards. This is, of course, a restriction, but covers the most frequent situations, as confirmed by our experiments in Section 6. We have also implemented some automata for full initialization of bidimensional arrays and for other orders of initialization, but they are not presented here due to space limitations, and we do not consider them a contribution of this paper.

Our abstract domain is rather simple: it contains only 5 elements. This permits us to provide a fully detailed soundness proof of low complexity. The key to this simplicity is abstract interpretation, that relates the semantics of the program with the pattern recognition ability of the automaton. The analysis in [6] is more general and precise than ours (for instance, it deals with out-of-order array initializations, while we are not able to do it), but this comes at the cost of a higher theoretical complexity. We could not perform an experimental comparison with them, since we analyze Java bytecode while the Clousot analyzer of [6] works for .NET. Also, the great precision of [6] largely depends on its specific instantiation and available supporting aliasing analysis.

One can apply our analysis also to position-based collection classes such as `java.util.ArrayList`. However, those Java classes have been devised in such a way that elements can be read only if they have already been set, or an exception is thrown. Hence, in Java, they are always fully initialized, potentially to `null`.

Proving full array initialization at a program point does not mean that the array is initialized *to some kind of values* (e.g. to non-`null` values) and that this remains true later, when the array is accessed, possibly in different methods from the one which initializes it. The local initialization at a program point must be lifted to a global property at all program points where the array is read. An extended version of this paper [10] presents our solutions to these problems.

2 A Simple Imperative Language and Its Semantics

We present here a simplified imperative language, inspired by [5]. It exposes only features relevant to our work. Namely, it has a minimal set of types, does

E	$::=$	$n \mid x$	Integer / Variable	B	$::=$	$\mathtt{true} \mid \mathtt{false}$	Truth/Falsity
	$\mid$	$x.length$	Array length		$\mid$	$\neg\mathsf{B}_1$	Negation
	$\mid$	$x[\mathsf{E}]$	Array element		$\mid$	$\mathsf{E}_1 \otimes \mathsf{E}_2$	$\otimes \in \{<, \leq, =\}$
	$\mid$	$\mathsf{E}_1 \oplus \mathsf{E}_2$	$\oplus \in \{+, -, *, \div, \%\}$		$\mid$	$\mathsf{B}_1 \oslash \mathsf{B}_2$	$\oslash \in \{\wedge, \vee\}$
A	$::=$	B	Test	C	$::=$	$\mathsf{L}_1 : \mathsf{A} \to \mathsf{L}_2;$	Command
	$\mid$	$x := \mathsf{E}$	Variable assignment				
	$\mid$	$x := \mathtt{new}\ t[\mathsf{E}]$	Creation of an array	$n \in \mathbb{Z},\ x \in \mathsf{Var},\ \mathsf{E}, \mathsf{E}_1, \mathsf{E}_2 \in \mathbb{E},$			
	$\mid$	$x[\mathsf{E}] := \mathsf{E}$	Array element assign	$\mathsf{B}, \mathsf{B}_1, \mathsf{B}_2 \in \mathbb{B},\ \mathsf{A} \in \mathbb{A},\ \mathsf{C} \in \mathbb{C},\ t \in \mathsf{Type}$			

Fig. 2. Abstract syntax of programs

not include classes, structures, procedures (our analysis is intraprocedural) nor exceptions. The actual implementation of our analysis includes all features of monothreaded 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.

Definition 1 (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* $\mathsf{L}_1 : \mathsf{A} \to \mathsf{L}_2;$, *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* $\mathsf{ini}(\mathsf{C}) = \mathsf{L}_1$, $\mathsf{suc}(\mathsf{C}) = \mathsf{L}_2$ *and* $\mathsf{act}(\mathsf{C}) = \mathsf{A}$. *Actions can be Boolean expressions in* $\mathbb{B}$ *(whose definition uses arithmetic expressions in* $\mathbb{E}$*), creation of arrays and assignments to local variables in* Var *or to array elements, and they are defined by the grammar in Fig. 2. The set of* types, Type, *is the minimal set containing* int *and* $\mathsf{array\ of}\ t$ *for every* $t \in \mathsf{Type}$. *We assume that every* $v \in \mathsf{Var}$ *has a static type* $\mathsf{t}(v)$.

We assume programs well-typed. For instance, given an action $x[y[3]] := z$, then $\mathsf{t}(y) = \mathsf{array\ of\ int}$ and $\mathsf{t}(x) = \mathsf{array\ of}\ \mathsf{t}(z)$. We let $\mathsf{vars}(D)$ stand for the variables occurring in an expression or action D, and $\mathsf{mod}(\mathsf{A}) \subseteq \mathsf{vars}(\mathsf{A})$ for the variables modified (i.e., assigned) by an action A. Namely, $\mathsf{mod}(\mathsf{B}) = \varnothing$, $\mathsf{mod}(x := \mathsf{E}) = \{x\}$, $\mathsf{mod}(x := \mathtt{new}\ t[\mathsf{E}]) = \{x\}$ and $\mathsf{mod}(x[\mathsf{E}_1] := \mathsf{E}_2) = \varnothing$.

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

Example 1. The figure on the left shows a Java loop (left) initializing an array a and its corresponding transition system (right). This code fragment is well-typed with $\mathsf{t}(i) = \mathsf{int}$ and $\mathsf{t}(a) = \mathsf{array\ of\ int}$.

At run-time, variables hold values which, in a programming language such as Java, may be primitive or non-primitive; the latter include objects and arrays. Def. 2 simplifies the picture by only considering integers as primitive values and arrays as non-primitive values. That simplification does not limit the results of this paper, that is only concerned about arrays and integer counters.

Definition 2 (Values). Values *are elements of* $\mathsf{Val} = \mathbb{Z} \cup \mathbb{L} \cup \{\boldsymbol{null}\}$*, where* $\mathbb{L}$ *is a finite set of* memory locations. 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*. A* memory *is a partial map* $\mu : \mathbb{L} \to \mathsf{Arr}$*.*

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 3 (Environment). *An* environment *is a pair* $e = \langle \rho, \mu \rangle$ *of a total map* $\rho : \mathsf{Var} \to \mathsf{Val}$ *and a memory* μ*. As in Java, we ban dangling pointers, i.e.,* $\forall v \in \mathsf{Var}$*, if* $\rho(v) \in \mathbb{L}$*, then* $\mu(\rho(v))$ *is defined, and for every* $a \in \mathsf{Arr}$ *and* $\forall i \in [0..a.\mathsf{length})$*, such that* $a[i] \in \mathbb{L}$*, then* $\mu(a[i])$ *is defined. We require static types respected i.e.,* $\forall v \in \mathsf{Var}$ *we have* $\rho(v) \approx_\mu \mathsf{t}(v)$*, where (i)* $x \approx_\mu \mathsf{int}$ *iff* $x \in \mathbb{Z}$*; and (ii)* $x \approx_\mu \mathsf{array\ of}\ t$ *iff* $x = \boldsymbol{null}$ *or (*$x \in \mathbb{L}$ *and* $\forall i \in [0..\mu(x).\mathsf{length})$*,* $\mu(x)[i] \approx_\mu t$*). We let* $\mathcal{E}$ *be the set of all environments.*

Definition 4 (Value of Expressions). *The evaluations* $\mathcal{A}[\![\mathsf{E}]\!] : \mathcal{E} \to \mathsf{Val}$ *and* $\mathcal{B}[\![\mathsf{B}]\!] : \mathcal{E} \to \{\mathsf{true}, \mathsf{false}\}$ *of expressions are partial maps defined as*

$$\begin{aligned}
\mathcal{A}[\![n]\!]\langle\rho,\mu\rangle &= n & \mathcal{B}[\![\mathsf{true}]\!]e &= \mathsf{true}\\
\mathcal{A}[\![x]\!]\langle\rho,\mu\rangle &= \rho(x) & \mathcal{B}[\![\mathsf{false}]\!]e &= \mathsf{false}\\
\mathcal{A}[\![x.\mathit{length}]\!]\langle\rho,\mu\rangle &= \mu(\rho(x)).\mathsf{length} & \mathcal{B}[\![\neg\mathsf{B}]\!]e &= \neg\mathcal{B}[\![\mathsf{B}]\!]e\\
\mathcal{A}[\![x[\mathsf{E}]]\!]\langle\rho,\mu\rangle &= \mu(\rho(x))[\mathcal{A}[\![\mathsf{E}]\!]\langle\rho,\mu\rangle] & \mathcal{B}[\![\mathsf{E}_1 \otimes \mathsf{E}_2]\!]e &= \mathcal{A}[\![\mathsf{E}_1]\!]e \otimes \mathcal{A}[\![\mathsf{E}_2]\!]e\\
\mathcal{A}[\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]\langle\rho,\mu\rangle &= \mathcal{A}[\![\mathsf{E}_1]\!]\langle\rho,\mu\rangle \oplus \mathcal{A}[\![\mathsf{E}_2]\!]\langle\rho,\mu\rangle & \mathcal{B}[\![\mathsf{B}_1 \oslash \mathsf{B}_2]\!]e &= \mathcal{B}[\![\mathsf{B}_1]\!]e \oslash \mathcal{B}[\![\mathsf{B}_2]\!]e.
\end{aligned}$$

Both maps are undefined when their defining expression is undefined or when any of of their arguments is undefined.

The execution of an action maps an initial environment into one of its successors.

Definition 5 (Semantics of Actions). *The semantics of an action* A *is a partial map* $\mathcal{S}[\![\mathsf{A}]\!] : \mathcal{E} \to \mathcal{E}$ *defined as*

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

$$\mathcal{S}[\![x := \mathsf{E}]\!]\langle\rho,\mu\rangle = \langle \rho[x \mapsto \mathcal{A}[\![\mathsf{E}]\!]\langle\rho,\mu\rangle], \mu\rangle$$

$$\mathcal{S}[\![x[\mathsf{E}_1] := \mathsf{E}_2]\!]\langle\rho,\mu\rangle = \begin{cases} \langle \rho, \mu[l \mapsto \mu(l)[i \mapsto \mathcal{A}[\![\mathsf{E}_2]\!]\langle\rho,\mu\rangle]]\rangle\\ \qquad \text{if } l = \rho(x),\ l \neq \boldsymbol{null}\\ \qquad i = \mathcal{A}[\![\mathsf{E}_1]\!]\langle\rho,\mu\rangle \text{ and } 0 \leqslant i < \mu(l).\mathsf{length}\\ \text{undefined otherwise}\end{cases}$$

$$\mathcal{S}[\![x := \boldsymbol{new}\ t[\mathsf{E}]]\!]\langle\rho,\mu\rangle = \langle \rho[x \mapsto l_f], \mu[l_f \mapsto \langle \mathcal{A}[\![\mathsf{E}]\!]\langle\rho,\mu\rangle, [\mathsf{def}, \ldots, \mathsf{def}]\rangle]\rangle,$$

where l_f *is a fresh location i.e.,* $l_f \notin \mathsf{dom}(\mu)$ *and* def *is the default value for* t*. This map is undefined when any of its arguments is undefined.*

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 6 (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* $\mathsf{cmd}(\sigma) = \mathsf{C}$*.*

A trace is a sequence of states that reflects an actual execution of the program.

Definition 7 (Trace). *A* finite partial trace τ *of states is a finite sequence of states* $\langle \sigma_1, \ldots, \sigma_n\rangle$*. For every* $1 \leqslant i < n$*, 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* $\langle \sigma^1_1, \ldots, \sigma^1_{n_1}\rangle \circ \langle \sigma^2_1, \ldots, \sigma^2_{n_2}\rangle = \langle \sigma^1_1, \ldots, \sigma^1_{n_1}, \sigma^2_1, \ldots, \sigma^2_{n_2}\rangle$ *if the latter is a trace; it is undefined otherwise.*

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 Def. 7.

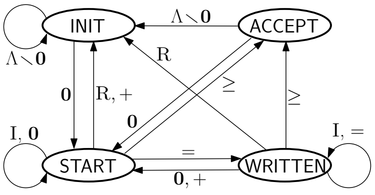
Definition 8 (Operational Semantics). *Let* $\mathsf{C} \in \mathbb{C}$ *and* $\Rightarrow^{\mathsf{C}} : \wp(\mathcal{T}) \to \wp(\mathcal{T})$ *be defined as* $T {\Rightarrow^{\mathsf{C}}} \{\tau \circ \langle e, \mathsf{C}\rangle \mid \tau \in T \wedge e \in \mathcal{E} \wedge \tau \circ \langle e, \mathsf{C}\rangle \text{ is defined}\}$*. The* operational semantics *at* C *is the set* $@\mathsf{C}$ *of all possible traces that lead to* C *and start with the execution of the distinguished command* C_{init}*, that is,* $@\mathsf{C} = \{\tau \in T_n \mid \exists T_1, \ldots, T_n \subseteq \mathcal{T}.\{\epsilon\} \Rightarrow^{\mathsf{C}_1} T_1 \Rightarrow^{\mathsf{C}_2} T_2 \cdots \Rightarrow^{\mathsf{C}_n} T_n \wedge \mathsf{C}_1 = \mathsf{C}_{init} \wedge \mathsf{C}_n = \mathsf{C}\}$*.*

3 Regular Trace Approximation

We define here an approximation of the execution traces of a program through a finite deterministic automaton. Its states are sets of traces and represent elements of an abstract domain. This 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 $\Rightarrow^{\mathsf{C}}$ operational relation. The automaton (Fig. 3) is designed for a pair of program variables a, of type array, and i, of integer type. It is defined over the *alphabet* $\Lambda = \{\mathbf{0}, +, =, \geqslant, \mathrm{I}, \mathrm{R}\}$ and has *states* $S = \{\mathsf{INIT}, \mathsf{START}, \mathsf{WRITTEN}, \mathsf{ACCEPT}\}$. Its *transition relation* is a function $\delta : S \times \Lambda \to S$: given states $\mathsf{p}, \mathsf{q} \in S$ and $\lambda \in \Lambda$, if the automaton has a transition from p to q labelled by λ, then $\delta(\mathsf{p}, \lambda) = \mathsf{q}$.

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

Definition 9 (Abstraction of Commands). *Consider an array a and an index integer variable i. The* abstraction of commands*,* $s : \mathbb{C} \to \Lambda$*, is defined as:* $s(\langle \mathsf{L}_1 : \mathsf{A} \to \mathsf{L}_2;\rangle) = \lambda$*, where the abstract value* λ *can be* $\mathbf{0}$*,* $+$*,* $=$ *or* $\geqslant$*, if action* A *is* $i := 0$*,* $i := i + 1$*,* $a[i] := \mathsf{E}$ *or* $\neg(i < a.length)$ *respectively. Otherwise, if* A *is such that* $\mathsf{mod}(\mathsf{A}) \cap \{a, i\} = \varnothing$*, i.e., if the local variables assigned by* A *are not* a *nor* i*, then* $\lambda = \mathrm{I}$*. In all other cases,* $\lambda = \mathrm{R}$*.*



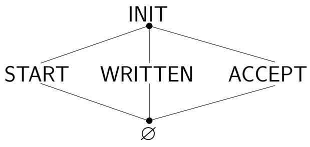


Fig. 3. Automaton detecting fully initialized arrays **Fig. 4.** The abstract domain $\mathbb{A}$

Note that assignments to $a[i]$ are abstracted into $=$ but any assignment to any other element of the array (such as to $a[i+1]$) is considered *irrelevant* (I). This abstraction is syntactical: a command with action $i := 0$ is abstracted into $\mathbf{0}$; another with action $i := 1 - 1$ into R. This does not affect correctness (Theorem 1) but one might simplify and normalize the actions, making the abstraction more semantical and precise. We have not implemented this improvement.

Definition 10 (Abstraction of Traces). *The* abstraction of traces *is given by* $\beta : \mathcal{T} \to \Lambda^*$ *and defined 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* $\beta(\epsilon) = \epsilon$.

Since Λ contains abstractions of commands, the meaning of the states of the automaton, S, becomes clearer. As we formalize below (Def. 11), INIT means that nothing is known about the last executed commands; START means that an assignment $i := 0$ is executed, and potentially followed by an alternation of assignments to $a[i]$ and unitary increments of i; WRITTEN means that an assignment to $a[i]$ has just been executed and the automaton is waiting to match it with a corresponding unitary increment of i; ACCEPT means that the complete initialization of the array can be asserted. An arbitrary number of irrelevant actions can always be executed between relevant actions.

Definition 11 (Abstract Domain $\mathbb{A}$). *The states of the automaton in Fig. 3 correspond to the following sets of traces defined by regular expressions over Λ:* $\mathsf{INIT} = \{\tau \in \mathcal{T} \mid \beta(\tau) \in \Lambda^*\} = \mathcal{T}$, $\mathsf{START} = \{\tau \in \mathcal{T} \mid \beta(\tau) \in \Lambda^*\mathbf{0}\mathrm{I}^*((=\mathrm{I}^*)^+ + \mathrm{I}^*)^*\}$, $\mathsf{WRITTEN} = \{\tau \in \mathcal{T} \mid \beta(\tau) \in \Lambda^*\mathbf{0}\mathrm{I}^*((=\mathrm{I}^*)^+ + \mathrm{I}^*)^*(=\mathrm{I}^*)^+\}$, $\mathsf{ACCEPT} = \{\tau \in \mathcal{T} \mid \beta(\tau) \in \Lambda^*\mathbf{0}\mathrm{I}^*((=\mathrm{I}^*)^+ + \mathrm{I}^*)^*(=\mathrm{I}^*)^* \geqslant\}$, *where* $*$ $(^+)$ *means zero or more* (*at least one*) *repetitions. We define the set* $\mathbb{A} = \{\mathsf{INIT}, \mathsf{START}, \mathsf{WRITTEN}, \mathsf{ACCEPT}, \varnothing\}$.

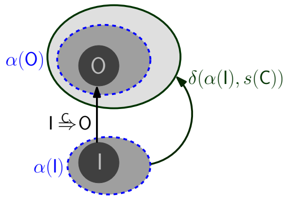
Proposition 1. $\mathbb{A}$ *is a Moore family of* $\wp(\mathcal{T})$ *i.e., it is an abstract domain ordered by set inclusion (Fig. 4). As standard for Moore families, the induced abstraction map* $\alpha : \wp(\mathcal{T}) \to \mathbb{A}$ *is* $\alpha(T) = \bigcap_{A \in \mathbb{A}, T \subseteq A} A$, *for every* $T \subseteq \mathcal{T}$.

Proof. The last relevant instruction of any $\tau \in \mathcal{T}$ is $\mathbf{0}$ or $+$, if $\tau \in \mathsf{START}$; is $=$, if $\tau \in \mathsf{WRITTEN}$; is $\geqslant$, if $\tau \in \mathsf{ACCEPT}$. The intersection of these elements is $\varnothing \in \mathbb{A}$. Since $\mathsf{INIT} = \mathcal{T}$, we have that $\forall a \in \mathbb{A}$, $a \cap \mathsf{INIT} = a$. Hence $\mathbb{A}$ is a Moore family. □

Lemma 1 states a consistency or correctness relation [4] between the operational semantics and the transitions of the automaton in Fig. 3.

Lemma 1. *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. Let $\tau \in \mathsf{O}$. By Def. 8, there exist $\tau' \in \mathsf{I}$ and $e \in \mathcal{E}$ s.t. $\tau = \tau' \circ \langle e, \mathsf{C}\rangle$, and therefore $\beta(\tau) = \beta(\tau')s(\mathsf{C})$. We proceed by case analysis. If $\alpha(\mathsf{I}) = \mathsf{START}$ and $s(\mathsf{C}) = \mathbf{0}$, then $\tau' \in \mathsf{I} \subseteq \alpha(\mathsf{I}) = \mathsf{START}$ and therefore $\beta(\tau) = \beta(\tau')s(\mathsf{C}) \in \Lambda^*\mathbf{0}\mathrm{I}^*((=\mathrm{I}^*)^+ + \mathrm{I}^*)^* s(\mathsf{C}) = \Lambda^*\mathbf{0}\mathrm{I}^*((=\mathrm{I}^*)^+ + \mathrm{I}^*)^*\mathbf{0} \subseteq \Lambda^*\mathbf{0} \subseteq \Lambda^*\mathbf{0}\mathrm{I}^*((=\mathrm{I}^*)^+ + \mathrm{I}^*)^*$. Hence, $\tau \in \mathsf{START}$. Since τ is arbitrary, $\mathsf{O} \subseteq \mathsf{START}$ and hence $\alpha(\mathsf{O}) \subseteq \alpha(\mathsf{START}) = \mathsf{START} = \delta(\mathsf{START}, \mathbf{0}) = \delta(\alpha(\mathsf{I}), s(\mathsf{C}))$. All other cases are proved similarly, see [10] for the full proof. $\square$



$\alpha(\mathsf{O})$
O
$\delta(\alpha(\mathsf{I}), s(\mathsf{C}))$
$\mathsf{I} \overset{\subseteq}{\Rightarrow} \mathsf{O}$
$\alpha(\mathsf{I})$
I

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 $\alpha(\mathsf{O})$.

4 The Static Analysis Algorithm

```
1: for all $\mathsf{C} \in \mathbb{C}$ do
2:     $\varphi(\mathsf{C}) := \varnothing$;
3: end for
4: $\mathsf{ws} := [\langle \mathsf{C}_{init}, \mathsf{INIT}\rangle]$;
5: $\varphi(\mathsf{C}_{init}) := \{\mathsf{INIT}\}$;
6: while (!ws.isEmpty()) do
7:     $\langle \mathsf{C}, \sigma^\sharp \rangle := \mathsf{ws.pop}()$;
8:     for all $\mathsf{C}_1$ such that $\mathsf{suc}(\mathsf{C}) = \mathsf{ini}(\mathsf{C}_1)$ do
9:         $\sigma_1^\sharp := \delta(\sigma^\sharp, s(\mathsf{C}))$;
10:        if $(\sigma_1^\sharp \notin \varphi(\mathsf{C}_1))$ then
11:            $\mathsf{ws.push}(\langle \mathsf{C}_1, \sigma_1^\sharp \rangle)$;
12:            $\varphi(\mathsf{C}_1) := \varphi(\mathsf{C}_1) \cup \{\sigma_1^\sharp\}$;
13:        end if
14:    end for
15: end while
```

Fig. 5. The ArrayInit algorithm

We describe here a static analysis that determines a subset of those commands 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, intraprocedural but aware of interprocedural side-effects, is designed for a specific pair $\langle a, i\rangle$ of variables. Its result lets us compute an under-approximation of the points where a has been initialized through a loop with index variable i. We repeat the analysis for each pair $\langle a, i\rangle$, but in practice, a pair $\langle a, i\rangle$ is significant only when a and i occur in actions $a[i] := \mathsf{E}$, which drastically reduces the number of pairs to consider.

Our analysis is formalized by the working set-based fixpoint algorithm in Fig. 5. When the working set (ws) is empty (line 6), a fixpoint is reached. The algorithm starts by applying the automaton in Fig. 3 from C_{init} and the INIT state (line 4). It reads and *executes* commands in any order allowed by the labels of the program. Consequently, the state of the automaton evolves and the algorithm records it just before executing a command. We use a map φ to that purpose, initially empty (line 2) and updated at each command (line 12).

Example 2. We show the application of ArrayInit over the Java loop and its corresponding transition system given in Example 1. We assume the working set implemented as a stack. We write I, S, W and A for INIT, START, $\mathsf{WRITTEN}$ and

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	$\langle\mathsf{C}_0,\mathsf{I}\rangle$	I	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	$\langle\mathsf{C}_1,\mathsf{S}\rangle,\langle\mathsf{C}_2,\mathsf{S}\rangle$	I	S	S	∅	∅	∅	∅	∅	∅	∅	∅
2	$\langle\mathsf{C}_2,\mathsf{S}\rangle,\langle\mathsf{C}_3,\mathsf{S}\rangle,\langle\mathsf{C}_4,\mathsf{S}\rangle$	I	S	S	S	S	∅	∅	∅	∅	∅	∅
3	$\langle\mathsf{C}_3,\mathsf{S}\rangle,\langle\mathsf{C}_4,\mathsf{S}\rangle,\langle\mathsf{C}_{10},\mathsf{A}\rangle$	I	S	S	S	S	∅	∅	∅	∅	∅	A
4	$\langle\mathsf{C}_4,\mathsf{S}\rangle,\langle\mathsf{C}_{10},\mathsf{A}\rangle,\langle\mathsf{C}_5,\mathsf{S}\rangle$	I	S	S	S	S	S	∅	∅	∅	∅	A
5	$\langle\mathsf{C}_{10},\mathsf{A}\rangle,\langle\mathsf{C}_5,\mathsf{S}\rangle,\langle\mathsf{C}_6,\mathsf{S}\rangle$	I	S	S	S	S	S	S	∅	∅	∅	A
6	$\langle\mathsf{C}_5,\mathsf{S}\rangle,\langle\mathsf{C}_6,\mathsf{S}\rangle$	I	S	S	S	S	S	S	∅	∅	∅	A
7	$\langle\mathsf{C}_6,\mathsf{S}\rangle,\langle\mathsf{C}_9,\mathsf{W}\rangle$	I	S	S	S	S	S	S	∅	∅	W	A
8	$\langle\mathsf{C}_9,\mathsf{W}\rangle,\langle\mathsf{C}_7,\mathsf{W}\rangle$	I	S	S	S	S	S	S	W	∅	W	A
9	$\langle\mathsf{C}_7,\mathsf{W}\rangle$	I	S	S	S	S	S	S	W	∅	W	A
10	$\langle\mathsf{C}_8,\mathsf{S}\rangle$	I	S	S	S	S	S	S	W	S	W	A

Fig. 6. Application of ArrayInit on an array initialization loop

ACCEPT. Fig. 6 shows the evolution of ws and φ during the iterations. Column C_i stands for the content of $\varphi(\mathsf{C}_i)$. Initially, $\mathsf{ws} = [\langle\mathsf{C}_0,\mathsf{I}\rangle]$ with $\mathsf{C}_0 = \mathsf{C}_{init}$, $\varphi(\mathsf{C}_0) = \{\mathsf{I}\}$ and φ holds the empty set elsewhere. Then we pop $\langle\mathsf{C}_0,\mathsf{I}\rangle$ from ws and compute $\delta(\mathsf{I}, s(\mathsf{C}_0)) = \delta(\mathsf{I},\mathbf{0}) = \mathsf{S}$. Since $\mathsf{suc}(\mathsf{C}_0) = \mathsf{ini}(\mathsf{C}_1) = \mathsf{ini}(\mathsf{C}_2)$, control passes to C_1 and C_2. Since $\mathsf{S} \notin \varnothing = \varphi(\mathsf{C}_1) = \varphi(\mathsf{C}_2)$, we push $\langle\mathsf{C}_1,\mathsf{S}\rangle$ and $\langle\mathsf{C}_2,\mathsf{S}\rangle$ into ws and update φ at C_1 and C_2. Since ws is not empty, the algorithm continues by popping $\langle\mathsf{C}_1,\mathsf{S}\rangle$ from ws and computes $\delta(\mathsf{S}, s(\mathsf{C}_1)) = \delta(\mathsf{S}, \mathrm{I}) = \mathsf{S}$. Since $\mathsf{suc}(\mathsf{C}_1) = \mathsf{ini}(\mathsf{C}_3)$ and $\mathsf{S} \notin \varnothing = \varphi(\mathsf{C}_3)$, we push $\langle\mathsf{C}_3,\mathsf{S}\rangle$ into ws and update φ at C_3. The algorithm continues similarly until the working set is empty. □

Example 3. Fig. 7 shows another Java fragment, its transition system and iterations of our algorithm. ArrayInit can identify even this unusual and non-trivial array initialization as a complete initialization (continues in Example 4). □

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

Proof. We prove a stronger property that entails the thesis: at the end of ArrayInit, for every sequence of application of $\Rightarrow$ of the form $\{\epsilon\} \Rightarrow^{\mathsf{C}_1} T_1 \cdots \Rightarrow^{\mathsf{C}_n} T_n$,

```
...
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	$1: i := 0 \rightarrow 2;$
C_1	$2: i < a.length \div 2 \rightarrow 3;$
C_2	$2: \neg(i < a.length \div 2) \rightarrow 7;$
C_3	$3: a[i] := 7 \rightarrow 4;$
C_4	$4: (i > a[i]) \rightarrow 5;$
C_5	$4: \neg(i > a[i]) \rightarrow 6;$
C_6	$5: a[i] := i \rightarrow 6;$
C_7	$6: i := i + 1 \rightarrow 2;$
C_8	$7: (i < a.length) \rightarrow 8;$
C_9	$7: \neg(i < a.length) \rightarrow 10;$
C_{10}	$8: a[i] := 10 \rightarrow 9;$
C_{11}	$9: i := i + 1 \rightarrow 7;$
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	$\langle\mathsf{C}_0,\mathsf{I}\rangle$	I	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	$\langle\mathsf{C}_1,\mathsf{S}\rangle,\langle\mathsf{C}_2,\mathsf{S}\rangle$	I	S	S	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
2	$\langle\mathsf{C}_2,\mathsf{S}\rangle,\langle\mathsf{C}_3,\mathsf{S}\rangle$	I	S	S	S	∅	∅	∅	∅	∅	∅	∅	∅	∅
3	$\langle\mathsf{C}_3,\mathsf{S}\rangle,\langle\mathsf{C}_8,\mathsf{S}\rangle,\langle\mathsf{C}_9,\mathsf{S}\rangle$	I	S	S	S	∅	∅	∅	∅	S	S	∅	∅	∅
4	$\langle\mathsf{C}_8,\mathsf{S}\rangle,\langle\mathsf{C}_9,\mathsf{S}\rangle,\langle\mathsf{C}_4,\mathsf{W}\rangle,\langle\mathsf{C}_5,\mathsf{W}\rangle$	I	S	S	S	S	S	∅	∅	S	S	∅	∅	∅
5	$\langle\mathsf{C}_9,\mathsf{S}\rangle,\langle\mathsf{C}_4,\mathsf{W}\rangle,\langle\mathsf{C}_5,\mathsf{W}\rangle,\langle\mathsf{C}_{10},\mathsf{S}\rangle$	I	S	S	S	S	S	∅	∅	S	S	S	∅	∅
6	$\langle\mathsf{C}_4,\mathsf{W}\rangle,\langle\mathsf{C}_5,\mathsf{W}\rangle,\langle\mathsf{C}_{10},\mathsf{S}\rangle,\langle\mathsf{C}_{12},\mathsf{A}\rangle$	I	S	S	S	S	S	∅	∅	S	S	S	∅	A
7	$\langle\mathsf{C}_5,\mathsf{W}\rangle,\langle\mathsf{C}_{10},\mathsf{S}\rangle,\langle\mathsf{C}_{12},\mathsf{A}\rangle,\langle\mathsf{C}_6,\mathsf{W}\rangle$	I	S	S	S	S	S	W	∅	S	S	S	∅	A
8	$\langle\mathsf{C}_{10},\mathsf{S}\rangle,\langle\mathsf{C}_{12},\mathsf{A}\rangle,\langle\mathsf{C}_6,\mathsf{W}\rangle,\langle\mathsf{C}_7,\mathsf{W}\rangle$	I	S	S	S	S	S	W	W	S	S	S	∅	A
9	$\langle\mathsf{C}_{12},\mathsf{A}\rangle,\langle\mathsf{C}_6,\mathsf{W}\rangle,\langle\mathsf{C}_7,\mathsf{W}\rangle,\langle\mathsf{C}_{11},\mathsf{W}\rangle$	I	S	S	S	S	S	W	W	S	S	S	W	A
10	$\langle\mathsf{C}_6,\mathsf{W}\rangle,\langle\mathsf{C}_7,\mathsf{W}\rangle,\langle\mathsf{C}_{11},\mathsf{W}\rangle$	I	S	S	S	S	S	W	W	S	S	S	W	A
11	$\langle\mathsf{C}_7,\mathsf{W}\rangle,\langle\mathsf{C}_{11},\mathsf{W}\rangle$	I	S	S	S	S	S	W	W	S	S	S	W	A
12	$\langle\mathsf{C}_{11},\mathsf{W}\rangle$	I	S	S	S	S	S	W	W	S	S	S	W	A

Fig. 7. A pair of loops fully initializing an array and their analysis with ArrayInit

with $T_n \neq \varnothing$ and $\mathsf{C}_1 = \mathsf{C}_{init}$, we have $T_n \subseteq \cup\varphi(\mathsf{C}_n)$ and, during the execution of ArrayInit, there is a step where a pair $\langle \mathsf{C}_n, \sigma^\sharp \rangle$, with $\alpha(T_n) \subseteq \sigma^\sharp$, is pushed on the working set ws. The thesis follows from the definition of @C. We prove this property by induction on the length $n \geqslant 1$ of the sequence of applications of $\Rightarrow$.
Base case: In this case $n = 1$, hence we have a sequence $\{\epsilon\} \Rightarrow^{\mathsf{C}_1} T_1$ with $\mathsf{C}_1 = \mathsf{C}_{init}$. Line 5 of the algorithm and the fact that it never removes states from the range of φ guarantee that at the end of the algorithm $T_1 \subseteq \mathsf{INIT} = \cup\varphi(\mathsf{C}_{init})$. Moreover, $\langle \mathsf{C}_{init}, \mathsf{INIT} \rangle$ is pushed into ws at line 4, and $\alpha(T_1) \subseteq \mathsf{INIT}$.
Induction: Assume that the result holds for some $n \geqslant 1$. We prove it for $n+1$. A sequence of applications of $\Rightarrow$ of length $n+1$ has form $\{\epsilon\} \Rightarrow^{\mathsf{C}_1} T_1 \cdots \Rightarrow^{\mathsf{C}_{n+1}} T_{n+1}$, with $T_{n+1} \neq \varnothing$ and $\mathsf{C}_1 = \mathsf{C}_{init}$. Since $T_{n+1} \neq \varnothing$, we also have $T_n \neq \varnothing$, by definition of $\Rightarrow$. In a sequence $\{\epsilon\} \Rightarrow^{\mathsf{C}_1} T_1 \cdots \Rightarrow^{\mathsf{C}_n} T_n$, by inductive hypothesis we have that, at the end of ArrayInit, $T_n \subseteq \cup\varphi(\mathsf{C}_n)$ and, during the execution of ArrayInit, there is a step at which $\langle \mathsf{C}_n, \sigma^\sharp \rangle$ with $\alpha(T_n) \subseteq \sigma^\sharp$ is pushed into ws. The algorithm terminates only when ws is empty (line 6), so that pair must have been removed from ws at some moment, at line 7. $T_{n+1} \neq \varnothing$, so $\mathsf{suc}(\mathsf{C}_n) = \mathsf{ini}(\mathsf{C}_{n+1})$. Hence C_{n+1} was considered in the loop at line 8, state $\sigma_1^\sharp = \delta(\sigma^\sharp, s(\mathsf{C}_n))$ was computed at line 9 and compared against $\varphi(\mathsf{C}_{n+1})$ at line 10. This might have had two outcomes: **1**) if $\sigma_1^\sharp \notin \varphi(\mathsf{C}_{n+1})$, line 12 adds $\sigma_1^\sharp$ to $\varphi(\mathsf{C}_{n+1})$, where it remains until the end of ArrayInit. No state is ever removed from the range of φ, so $\sigma_1^\sharp \subseteq \cup\varphi(\mathsf{C}_{n+1})$. By extensivity of α [4], Lemma 1 and monotonicity[1] of δ, we have $T_{n+1} \subseteq \alpha(T_{n+1}) \subseteq \delta(\alpha(T_n), s(\mathsf{C}_n)) \subseteq \delta(\sigma^\sharp, s(\mathsf{C}_n)) = \sigma_1^\sharp \subseteq \cup\varphi(\mathsf{C}_{n+1})$. Line 11 pushed $\langle \mathsf{C}_{n+1}, \sigma_1^\sharp \rangle$ into ws and from $T_{n+1} \subseteq \sigma_1^\sharp$ (shown above) we have $\alpha(T_{n+1}) \subseteq \alpha(\sigma_1^\sharp) = \sigma_1^\sharp$ since $\sigma_1^\sharp \in \mathbb{A}$. **2**) if $\sigma_1^\sharp \in \varphi(\mathsf{C}_{n+1})$, then it is still there at the end of ArrayInit. As above, we can prove that $T_{n+1} \subseteq \cup\varphi(\mathsf{C}_{n+1})$ and $\alpha(T_{n+1}) \subseteq \sigma_1^\sharp$. □

Hence our algorithm supports a correct array initialization analysis.

Theorem 1. *Consider a program P, variables a (array) and i (index) and the automaton in Fig. 3 for a and i. At the end of the* ArrayInit *algorithm, for every $\mathsf{C} \in \mathbb{C}$ such that $\varphi(\mathsf{C}) = \{\mathsf{ACCEPT}\}$ we have that $\mathsf{ini}(\mathsf{C})$ is a point of P where all elements of a have been initialized by a loop with index i.*

Proof. By Proposition 2, $@\mathsf{C} \subseteq \cup\varphi(\mathsf{C}) = \mathsf{ACCEPT}$ i.e., every trace τ leading to C is in the language of $\Lambda^*\mathbf{0}\mathrm{I}^*((=\mathrm{I}^*)^+ + \mathrm{I}^*)^*(=\mathrm{I}^*)^*\geqslant$. Hence τ ends with an assignment of 0 to i ($\mathbf{0}$) followed by a repetition of at least an assignment to $a[i]$ ($=$) and then a single increment of i ($+$). At the end of τ, i holds the length of a. Since only irrelevant actions are allowed in τ between those actions, a is definitely completely initialized at the end of the execution represented by τ. □

Example 4. From Fig. 6 we know that $\varphi(\mathsf{C}_{10}) = \{\mathsf{ACCEPT}\}$ at the end of the algorithm. By Theorem 1, the array has been completely initialized when program point $9 = \mathsf{ini}(\mathsf{C}_{10})$ is reached. The same holds for program point 10 in Fig. 7. □

[1] It can be proven easily (see [10]) that $\varnothing \subset \mathsf{p} \subseteq \mathsf{q} \Rightarrow \delta(\mathsf{p}, \lambda) \subseteq \delta(\mathsf{q}, \lambda)$.

5 Dealing with Implicit Upper Bounds and Side-Effects

Although the analysis of Sec. 3 and 4 determines where an array held in a *local variable* is fully initialized, it has strong limitations. It identifies the comparison between a loop index variable i and the size of an array a in a syntactical, explicit way (Def. 9): it must have the form $\neg(i < a.length)$. This is not the case in Fig. 1 and our 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* i.e., fields. The problem with fields goes beyond the extension of our language of expressions (Fig. 2) with a new expression *x.field*: it actually requires careful attention to side-effects. For instance, method `foo` in Fig. 8 recreates the array at each iteration. At the end of the loop, none of its elements is initialized. A naive extension of our analysis to arrays held in fields might easily turn out to be unsound. We discuss below how we overcome these two limitations.

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

Fig. 8. Side-effects hinder the full initialization of `this.a`

Implicit Upper Bounds. We consider some frequent implicit ways of expressing the upper bound of an array. Often, a variable is used, as `numpoints` in Fig. 1. To prove that it holds the array length, we use the *definite expression aliasing* analysis available in Julia, a traditional available expression analysis [1] for bytecode: bindings from variables to expressions are *generated* by assignments, that also *kill* other bindings referring to the old value of the variables. In Fig. 1, the binding `numpoints = this.mOriginal.length` is generated by the first `new` and never killed later, since `this`, `numpoints` and `this.mOriginal` are not updated. Similarly for the binding `numpoints = this.mRotated.length`. Def. 9 is improved: when A is $\neg(i < var)$, its abstraction is $\geqslant$ if, there, we have the binding $var = a.length$ (a can be a local variable or a field).

In other cases, the upper bound is a numerical constant. This includes the case when a final static integer field is used (i.e., a symbolic constant) since compilers usually replace it with its numerical value and Julia analyzes the bytecode. Here, we must be sure that the *same* constant is used for the array length *wherever* it is created, also outside the method where the initialization loop occurs. Since there might be more creation points for the objects stored inside the array variable, 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 be created. This is a concretization of class analysis [11]. Def. 9 is improved: when A is $\neg(i < con)$ and con a numerical constant, its abstraction is $\geqslant$ if all creation points for a (the variable being initialized) have the form `new T[`*con*`]`.

Side-Effects. When the array is stored in a field, as in *x.field*, we must strengthen the notion of irrelevant action A (case I of Def 9): we must require that $\mathsf{mod}(\mathsf{A}) \cap \{x, i\} = \varnothing$ *and* that A does not modify *field*. The only actions that might modify

NAME	VENDOR	LOC	TOTAL LOC	ARRAY INITIALIZATION			TOTAL TIME
				TOTAL	DETECTED	TIME	
AbdTest	Android Distribution	489	56334	1	1	2.36	121.73
AccelerometerPlay	Android Distribution	306	46854	1	1	0.35	71.99
CubeWallpaper	Android Distribution	370	25654	3	3	0.12	28.51
HoneycombGallery	Android Distribution	948	71501	1	0	1.06	157.85
TicTacToe	Android Distribution	607	59040	3	3	0.70	102.65
Snake	Android Distribution	420	57075	1	0	0.36	117.49
Real3D	Android Distribution	1228	74384	2	2	1.06	177.95
ChimeTimer	Moonblink	4095	95781	9	7	0.80	383.45
Dazzle	Moonblink	4376	100271	4	1	1.02	394.44
OnWatch	Moonblink	9746	113368	10	6	2.91	525.15
Tricorder	Moonblink	10410	106100	17	11	1.01	467.58
TestAppv2	Typoweather	377	58365	1	1	0.38	102.34
TxWthr	Typoweather	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. 9. Experiments with our array initialization analysis. LOC is the number of non-blank, non-comment program lines reached and hence analyzed by Julia; TOTAL LOC is the total number of analyzed lines, including `java.*` and `android.*` libraries; TOTAL is the number of reachable loops in those programs (not in libraries) that fully initialize an array, computed by manual check; DETECTED is the number of them 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 time in seconds of our array initialization analysis; it is a small fraction of the TOTAL TIME (in seconds) of the nullness analysis of Julia: the latter includes parsing of the class files, preprocessing, aliasing, sharing, creation points, expression aliasing and side-effects analyses.

field are explicit assignments to *y.field*, for any *y*, and calls to non-pure methods (neither of them is in Fig. 2, but naturally a real language includes both). 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).

6 Experiments

We implemented our analysis in the Julia tool: `http://www.juliasoft.com`. Experiments in Figure 9 were performed on a quad-core Intel Xeon 64 bits machine at 2.66GHz, with 8GB of RAM, Linux 2.6.27 and Sun jdk 1.6. We analyzed the lexical analyzers generator `JFlex` (`http://jflex.de`), the `plume` library by Michael D. Ernst (`http://code.google.com/p/plume-lib`), the non-termination analyzer `nti` by Étienne Payet, programs from Google's Android 3.0 distribution and Android applications from public repositories (`http://code.google.com/p/moonblink` and `http://code.google.com/p/typoweather`).

NAME	NULLNESS	
	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

While Figure 9 shows that our array initialization analysis is fast and precise, the table on the left shows that it is useful to a client analysis. In particular, it considers those programs from Fig. 9 that contain at least a loop initializing an array of reference type, since otherwise the array initialization analysis would be irrelevant for nullness analysis. It reports the number of null-pointer warnings with (ArrInit) and without (~~ArrInit~~) our array initialization analysis. 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. 9).

When Julia fails to spot complete array initialization, the problem is related to weaknesses in the supporting analyses rather than to our array initialization analysis. For instance, there is a complete array initialization in `HoneycombGallery` that Julia fails to spot (Fig. 9). Here it is:

```
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 Sec. 5. 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.

7 Conclusion

Our new automaton-based abstract interpretation detects fully initialized arrays held in local variables or fields. The implementation is efficient, precise and effective to support a client nullness analysis. We observe that our array initialization analysis is not tailored to nullness, but can support any other client analysis.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1986)
2. Bradley, A.R., Manna, Z., Sipma, H.B.: What's Decidable About Arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005)
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the 4th Symp. on Principles of Programming Languages, pp. 238–252. ACM (1977)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of the 6th Symposium on Principles of Programming Languages (POPL 1979), pp. 269–282. ACM (1979)
5. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: Proc. of the 29th Symposium on Principles of Programming Languages (POPL 2002), pp. 178–190. ACM (2002)

6. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proc. of the 38th Symposium on Principles of Programming Languages, New York, USA, pp. 105–118 (2011)
7. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proc. of the 29th Symposium on Principles of Programming Languages (POPL 2002), pp. 191–202. ACM (2002)
8. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
9. Habermehl, P., Iosif, R., Vojnar, T.: What Else Is Decidable about Integer Arrays? In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 474–489. Springer, Heidelberg (2008)
10. Nikolić, D., Spoto, F.: Automaton-based array initialization analysis, `http://profs.sci.univr.it/~nikolic/download/LATA2012/LATA2012Ext.pdf`
11. Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: Proceedings of Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 1991). ACM SIGPLAN Notices, vol. 26(11), pp. 146–161. ACM (1991)
12. Spoto, F.: Precise null-pointer analysis. Software and Systems Modeling 10(2), 219–252 (2011)