Checking Array Bounds by Abstract Interpretation and Symbolic Expressions

Fausto Spoto, Etienne Payet

Checking Array Bounds by Abstract Interpretation and Symbolic Expressions

Étienne Payet[1] and Fausto Spoto[2]

[1] Laboratoire d'Informatique et de Mathématiques, Université de la Réunion, France
[2] Dipartimento di Informatica, Università di Verona, Italy

Abstract. Array access out of bounds is a typical programming error. From the '70s, static analysis has been used to identify where such errors actually occur at runtime, through abstract interpretation into linear constraints. However, feasibility and scalability to modern object-oriented code has not been established yet. This article builds on previous work on linear constraints and shows that the result does not scale, when polyhedra implement the linear constraints, while the more abstract zones scale to the analysis of medium-size applications. Moreover, this article formalises the inclusion of symbolic expressions in the constraints and shows that this improves its precision. Expressions are automatically selected on-demand. The resulting analysis applies to code with dynamic memory allocation and arrays held in expressions. It is sound, also in the presence of arbitrary side-effects. It is fully defined in the abstract interpretation framework and does not use any code instrumentation. Its proof of correctness, its implementation inside the commercial Julia analyzer and experiments on third-party code complete the work.

1 Introduction

Arrays are extensively used in computer programs since they are a compact and efficient way of storing and accessing vectors of values. Array elements are indexed by their integer offset, which leads to a runtime error if the index is negative or beyond the end of the array. In C, this error is silent, with unpredictable results. The Java runtime, instead, mitigates the problem since it immediately recognizes the error and throws an exception. In both cases, a definite guarantee, at compilation time, that array accesses will never go wrong, for all possible executions, is desirable and cannot be achieved with testing, that covers only some execution paths. Since the values of array indices are not computable, compilers cannot help, in general. However, static analyses that find such errors, and report some false alarms, exist and are an invaluable support for programmers.

Abstract interpretation has been applied to array bounds inference, from its early days [8, 5], by abstracting states into linear constraints on the possible values of local variables, typically polyhedra [4, 3]. Such inferred constraints let then one check if indices are inside their bounds. For instance, in the code:

```
public DiagonalMatrix inverse(double[] diagonal) {
```

```
2    double[] newDiagonal = new double[diagonal.length]; // local var.
3    for (int i = 0; i < diagonal.length; i++)
4      newDiagonal[i] = 1 / diagonal[i]; ...  }
```

the analysis in [8, 5], at line 4, infers $0 \leqslant i < \mathit{diagonal} = \mathit{newDiagonal}$; *i.e.*, index `i` is non-negative and is smaller than the length of the array `diagonal`, which is equal to that of `newDiagonal`. This is enough to prove that both accesses `newDiagonal[i]` and `diagonal[i]` occur inside their bounds, always.

Programming languages have largely evolved since the '70s and two problems affect the application of this technique to modern software. First, code is very large nowadays, also because object-oriented software uses large libraries that must be included in the analysis. The actual scalability of the technique, hence, remains unproved. Second, the limitation to constraints on *local variables* (such as `i`, `diagonal` and `newDiagonal` above) is too strict. Current programming languages allow arrays to be stored in expressions built from dynamically heap-allocated object fields and other arrays, which are not local variables. For instance, the previous example is actually a simplification of the following real code from class `util.linalg.DiagonalMatrix` of a program called Abagail (Sec. 7):

```
1  private double[] diagonal; // object field, not local variable
2  public DiagonalMatrix inverse() {
3    double[] newDiagonal = new double[this.diagonal.length];
4    for (int i = 0; i < this.diagonal.length; i++)
5      newDiagonal[i] = 1 / this.diagonal[i]; ...  }
```

The analysis in [8, 5] infers $0 \leqslant i, 0 \leqslant \mathit{newDiagonal}$ at line 5 above, since `this.diagonal` is not a local variable and consequently cannot be used in the constraint. The latter does not entail that the two array accesses are safe now, resulting in two false alarms. Clearly, one should allow expressions such as `this.diagonal` in the constraint and infer $0 \leqslant i < \mathit{this.diagonal} = \mathit{newDiagonal}$. But this is challenging since there are (*infinitely*) many expressions (potentially affecting scalability) and since expressions might change their value by *side-effect* (potentially affecting soundness). In comparison, at a given program point, only *finitely* many local variables are in scope, whose value can *only* be changed by syntactically explicit assignment to the affected variable. Hence, this challenge is both technical (the implementation must scale) and theoretical (the formal proof of correctness must consider all possible side-effects).

One should not think that it is enough to include object fields in the constraints, to improve the expressivity of the analysis. Namely, fields are just examples of expressions. In real code, it is useful to consider also other expressions. For instance, the following code, from class `util.linalg.LowerTriangularMatrix` of Abagail, performs a typical nested loop over a bidimensional array:

```
1    UpperTriangularMatrix result = new UpperTriangularMatrix(...);
2    for (int i = 0; i < this.data.length; i++)
3      for (int j = 0; j < this.data[i].length; j++) {
4        // any extra code could occur here
5        result.set(j, i, this.data[i][j]);  }
```

To prove array accesses safe at line 5 above, one should infer that $0 \leqslant i < \mathit{this.data}, 0 \leqslant j < \mathit{this.data}[i]$. The analysis in [8, 5] cannot do it, since it considers local variables and abstracts away fields (`this.data`) and array elements (`this.data[i]`). Moreover, safeness of these accesses can be jeopardised by extra code at line 4 modifying `this.data` or `this.data[i]`: side-effects can affect soundness. That does not happen for arrays held in local variables, as in [8, 5].

For an example of approximation of even more complex expressions, consider the anonymous inner class of `java.net.sf.colossus.tools.ShowBuilderHexMap` from program Colossus (Sec. 7). It iterates over a bidimensional array:

```
1  clearStartListAction = new AbstractAction(...) {
2    public void actionPerformed(ActionEvent e) {
3      for (int i = 0; i < h.length; i++)
4        for (int j = 0; j < h[i].length; j++)
5          if ((h[i][j] != null) && (h[i][j].isSelected())) {
6            h[i][j].unselect();
7            h[i][j].repaint();  }}};
```

Here, `h` is a field of the outer object *i.e.*, in Java, shorthand for `this.this$0.h` (a field of field `this$0`, the synthetic reference to the outer object); `h[i]` stands for `this.this$0.h[i]` (an element of an array in a field of a field). In order to prove the array accesses at lines 4, 5, 6 and 7 safe, the analyser should prove that the constraint $0 \leqslant i < \mathit{this.this\$0.h}, 0 \leqslant j < \mathit{this.this\$0.h}[i]$ holds at those lines. The analysis in [8, 5] cannot do it, since it abstracts away the expressions `this.this$0.h` and `this.this$0.h[i]`. This results in false alarms. Note that, to prove the access at line 7 safe, an analyser must prove that `isSelected()` and `unselect()` do not affect `h` nor `h[i]`. That is, it must consider side-effects.

The contribution of this article is the extension of [8, 5] with specific program expressions, in order to improve its precision. It starts from the formalisation of [8, 5] for object-oriented languages with dynamic heap allocation, known as path-length analysis [19]. It shows that its extension with expressions, using zones [10] rather than polyhedra [4, 3], scales to real code and is more precise than [8, 5]. This work is bundled with a formal proof of correctness inside the abstract interpretation framework. This analysis, as that in [19], is interprocedural, context and flow-sensitive and deals with arbitrary recursion.

This article is organised as follows. Sec. 2 reports related work. Sec. 3 and 4 recall semantics and path-length from [19] and extend it to arrays. Sec. 5 introduces the approximation of expressions. Sec. 6 defines the new static analysis, with side-effect information for being sound. Sec. 7 describes its implementation inside the Julia analyser and experiments of analysis of open-source Java applications. The latter and the results of analysis can be found in [14]. Analyses can be rerun online at `https://portal.juliasoft.com` (instructions in [14]).

2 Related Work and Other Static Analysers

Astrée [6] is a state of the art analyser that infers linear constraints. For scalability, it uses octagons [12] rather than polyhedra [7]. It targets embedded C software, with clear limitations [6]: no dynamic memory allocation, no unbound recursion, no conflicting side-effects and no use of libraries. Fields and arrays are dealt under the assumption that there is no dynamic memory allocation, which limits the analysis to embedded C software [11]. These assumptions simplify the analysis since, in particular, no dynamic memory allocation means that there is a finite number of fields or array elements, hence they can be statically grouped and mapped into linear variables; and the absence of conflicting side-effects simplifies the generation of the constraints between such variables. However, such assumptions conflict with the reality of Java code. Even a minimal Java program uses dynamic memory allocation and a very large portion of the standard Java library, which is much more pervasive that the small standard C library: for instance, a simple print statement reaches library code for formatting and localization and the collection library contains hundreds of intensively used classes. The recent 2018 competition on software verification showed that a few tools can already perform bound verification for arrays in C, with good results[3].

A type system has been recently defined for array bounds checking in Java [15]. It infers size constraints but uses code annotation. Thus, it is not fully automatic.

Facebook has built the buffer overflow analyser Inferbo[4] on top of Infer[5]. Inferbo uses symbolic intervals for approximating indices of arrays held in local variables. We ran Inferbo on Java code but the results were non-understandable. After personal communication with the Infer team, we have been confirmed that Infer does work also on Java code, but Inferbo is currently limited to C only.

We also ran FindBugs[6] at its *maximal analysis effort*. It spots only very simple array access bounds violations. For instance it warns at the second statement of `int t[] = new int[5]; t[9] = 3`. However, in the programs analysed in Sec. 7, it does not issue any single warning about array bounds violations, missing all the true alarms that our analysis finds.

Previous work [19] defined a *path-length* analysis for Java, by using linear constraints over local variables only. It extends [8, 5] to deal with dynamically heap-allocated data structures and arbitrary side-effects. It uses a bottom-up, denotational fixpoint engine, with no limits on recursion. It was meant to support a termination prover. This article leverages its implementation inside the Julia analyser for Java bytecode [17], by adding constraints on expressions.

This work has been inspired by [1, 2], which, however, has the completely different goal of termination analysis. It identifies fields that are locally constant inside a loop and relevant for the analysis, by using heuristics, then performs a polyvariant code instrumentation to translate such fields into *ghost variables*. That analysis is limited to fields and there is no formalisation of path-length with arrays. In this article, instead, path-length with arrays is formalised, applied to array bounds checking with an evaluation of its scalability. Our analysis

[3] https://sv-comp.sosy-lab.org/2018/results/results-verified/META_MemSafety.table. See, in particular, the results for tools Map2Check, Symbiotic and Ultimate

[4] https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer

[5] http://fbinfer.com

[6] http://findbugs.sourceforge.net

identifies, on-demand, expressions that need explicit approximation and can be much more complex than fields *e.g.*, array elements of variables or fields, or fields of array elements; moreover, it does not need any code instrumentation, since expressions only exist in the abstract domain, where linear variables symbolically stand for expressions. As a consequence, it has a formal correctness proof, completely inside the abstract interpretation framework. As far as we can infer from the papers, the analysis in [1, 2] has no formal correctness proof.

3 Concrete Domain

We extend [19] with arrays. Namely, an array arr of type t and length $n \in \mathbb{N}$ is a mapping from $[0, \ldots, n-1]$ to values of type t; it has type $arr.t$ and length $arr.length$. A *memory* maps locations to *reference values i.e.*, objects or arrays. The set of locations is $\mathbb{L}$, the set of arrays is $\mathbb{A}$. A multidimensional array is just an array of locations, bound to arrays, exactly as in Java. The set of classes of our language is $\mathbb{K}$ and the set of types is $\mathbb{T} = \mathbb{K} \cup \{\texttt{int}\} \cup \mathbb{T}[\,]$. The domain of a function f is $dom(f)$, its codomain or range is $rng(f)$. By $f(x)\downarrow$ we mean that f is defined at x; by $f(x)\uparrow$, that it is undefined at x. The composition of functions f and g is $f \circ g = \lambda x.g(f(x))$, undefined when $f(x)\uparrow$ or $g(f(x))\uparrow$.

A *state* is a triple $\langle l \parallel s \parallel \mu \rangle$ that contains the values of the local variables l, those of the operand stack elements s, and a memory μ. Local variables and stack elements are bound to values compatible with their static type. Dangling pointers are not allowed. The size of l is denoted as $\#l$, that of s as $\#s$. The elements of l and s are indexed as l^k and s^k, where s^0 is the base of the stack and $s^{\#s-1}$ is its top. The set of all states is Σ, while $\Sigma_{i,j}$ is the set of states with exactly i local variables and j stack elements. The concrete domain is $\langle \wp(\Sigma), \subseteq \rangle$ *i.e.*, the powerset of states ordered by set inclusion. Denotations are the functional semantics of a single bytecode instruction or of a block of code.

Definition 1. *A* denotation *is a partial function $\Sigma \to \Sigma$ from a* pre-*state to a* post-*state. The set of denotations is Δ, while $\Delta_{l_i,s_i \to l_o,s_o}$, stands for $\Sigma_{l_i,s_i} \to \Sigma_{l_o,s_o}$. Each instruction* ins*, occurring at a program point p, has semantics $ins_p \in \Delta_{l_i,s_i \to l_o,s_o}$ where l_i, s_i, l_o, s_o are the number of local variables and stack elements in scope at p and at the subsequent program point, respectively. Fig. 1 shows those dealing with arrays and fields. Others can be found in [19].*

Fig. 1 assumes runtime types correct. For instance, $arrayload_p\ t$ finds on the stack a value ℓ which is either `null` or a location bound to an array of type $t' \leqslant t$. Such static constraints must hold in legal Java bytecode [9], hence are not checked in Fig. 1. Others are dynamic and checked in Fig. 1: for instance, index v must be inside the array bounds ($0 \leqslant v < \mu(\ell).length$). Fig. 1 shows explicit types for instructions, when relevant in this article, such as t in $arrayload_p\ t$. They are implicit in real bytecode, for compactness, but can be statically inferred [9]. Fig. 1 assumes that runtime violations of bytecode preconditions stop the Java Virtual Machine, instead of throwing an exception. Exceptions can be accomodated in this fragment (and are included in our implementation), at the

$$
\begin{aligned}
\mathit{store}_p\ k &= \lambda\langle l \parallel v :: s \parallel \mu\rangle.\langle l[k \mapsto v] \parallel s \parallel \mu\rangle\\
\mathit{inc}_p\ k\ c &= \lambda\langle l \parallel s \parallel \mu\rangle.\langle l[k \mapsto l^k + c] \parallel s \parallel \mu\rangle\\
\mathit{newarray}_p\ t\ n &= \lambda\langle l \parallel v_n :: \cdots :: v_1 :: s \parallel \mu\rangle.\langle l \parallel \ell :: s \parallel \mu[\ell \mapsto \mathit{arr}, \ldots]\rangle \quad \text{if } 0 \leqslant v_1, \ldots, v_n\\
&\quad \text{where } \ell \in \mathbb{L} \text{ is fresh and } \mathit{arr} \text{ is an } n\text{-dimensional array of type } t\\
&\quad \text{and lengths } v_1, \ldots, v_n, \text{ initialised to its default value,}\\
&\quad \text{with subarrays, if any, bound in the } \ldots \text{ part to fresh locations}\\
\mathit{arrayload}_p\ t &= \lambda\langle l \parallel v :: \ell :: s \parallel \mu\rangle.\langle l \parallel \mu(\ell)(v) :: s \parallel \mu\rangle \quad \text{if } \ell \neq \texttt{null},\ 0 \leqslant v < \mu(\ell).\mathit{length}\\
\mathit{arraystore}_p\ t &= \lambda\langle l \parallel v_1 :: v_2 :: \ell :: s \parallel \mu\rangle.\langle l \parallel s \parallel \mu[\ell \mapsto \mu(\ell)[v_2 \mapsto v_1]]\rangle\\
&\quad \text{if } \ell \neq \texttt{null} \text{ and } 0 \leqslant v_2 < \mu(\ell).\mathit{length}\\
\mathit{arraylength}_p\ t &= \lambda\langle l \parallel \ell :: s \parallel \mu\rangle.\langle l \parallel \mu(\ell).\mathit{length} :: s \parallel \mu\rangle \quad \text{if } \ell \neq \texttt{null}\\
\mathit{getfield}_p\ f &= \lambda\langle l \parallel \ell :: s \parallel \mu\rangle.\langle l \parallel \mu(\ell)(f) :: s \parallel \mu\rangle \quad \text{if } \ell \neq \texttt{null}\\
\mathit{putfield}_p\ f &= \lambda\langle l \parallel v :: \ell :: s \parallel \mu\rangle.\langle l \parallel s \parallel \mu[\ell \mapsto \mu(\ell)[f \mapsto v]]\rangle \quad \text{if } \ell \neq \texttt{null}\\
(\mathit{call}_p\ \kappa.m)(\delta) &= \text{see [19]}
\end{aligned}
$$

Fig. 1. The concrete semantics of a fragment of Java bytecode, with arrays and fields. The semantics of an instruction is implicitly undefined if its preconditions do not hold.

price of extra complexity. This is explained and formalised in [13]. Instruction newarray allocates an array of n dimensions and leaves its pointer ℓ on top of the stack. When $n > 1$, a multidimensional array is allocated. In that case, the array at ℓ is the spine of the array, while its elements are arrays themselves, held at futher newly allocated locations. Instructions arrayload, arraystore and arraylength operate on the array $\mu(\ell)$, where ℓ is provided on the stack. The first two check if the index is inside its bounds. getfield and putfield are similar, but $\mu(\ell)$ is an object. Objects are represented as functions from field names to field values. call plugs the denotation δ of the callee(s) at the calling point.

4 Path-Length Abstraction with Arrays

Path-length [19] is a property of local and stack variables, namely, the length of the maximal chain of pointers from the variable. It leads to an abstract interpretation of Java bytecode, reported below, after extending path-length to arrays: their path-length is their length. This is consistent with the fact that the length of the arrays is relevant for checking array bounds and for proving the termination of loops over arrays. Hence the elements of an array are irrelevant *w.r.t.* path-length and only the first dimension of a multidimensional array matters.

Definition 2. *Let μ be a memory. For every $j \geqslant 0$, let **1)** $\mathit{len}^j(\texttt{null}, \mu) = 0$, **2)** $\mathit{len}^j(i, \mu) = i$ if $i \in \mathbb{N}$, **3)** $\mathit{len}^j(\ell, \mu) = \mu(\ell).\mathit{length}$ if $\ell \in \mathit{dom}(\mu)$ and $\mu(\ell) \in \mathbb{A}$, **4)** $\mathit{len}^0(\ell, \mu) = 0$ if $\ell \in \mathit{dom}(\mu)$ and $\mu(\ell) \notin \mathbb{A}$, **5)** $\mathit{len}^{j+1}(\ell, \mu) = 1 + \max\left\{\mathit{len}^j(\ell', \mu) \mid \ell' \in \mathit{rng}(\mu(\ell)) \cap \mathbb{L}\right\}$ if $\ell \in \mathit{dom}(\mu)$ and $\mu(\ell) \notin \mathbb{A}$, with the* *assumption that the maximum of an empty set is* 0*. The* path-length of a value v in μ *is* $len(v,\mu) = \lim_{j\to\infty} len^j(v,\mu)$.

In the last case of the definition of len^j, the intersection with $\mathbb{L}$ selects only the non-primitive fields of object $\mu(\ell)$. If $i \in \mathbb{Z}$ then $len(i,\mu) = len^j(i,\mu) = i$ for every $j \geqslant 0$ and memory μ. Similarly, $len(\texttt{null},\mu) = len^j(\texttt{null},\mu) = 0$. Moreover, if location ℓ is bound to a cyclical data-structure, then $len(\ell,\mu) = \infty$.

A state can be mapped into a *path-length assignment i.e.*, into a function specifying the path-length of its variables. This comes in two versions: in the *pre-state version* $\widecheck{len}$, the state is considered as the pre-state of a denotation. In the *post-state version* $\widehat{len}$, it is considered as the post-state of a denotation.

Definition 3. *Let* $\langle l \,\|\, s \,\|\, \mu\rangle \in \Sigma_{\#l,\#s}$*. Its* pre-state path-length assignment *is* $\widecheck{len}(\langle l \,\|\, s \,\|\, \mu\rangle) = [\check{l}^k \mapsto len(l^k,\mu) \mid 0 \leqslant k < \#l] \cup [\check{s}^k \mapsto len(s^k,\mu) \mid 0 \leqslant k < \#s]$*. Its* post-state path-length assignment *is* $\widehat{len}(\langle l \,\|\, s \,\|\, \mu\rangle) = [\hat{l}^k \mapsto len(l^k,\mu) \mid 0 \leqslant k < \#l] \cup [\hat{s}^k \mapsto len(s^k,\mu) \mid 0 \leqslant k < \#s]$*.*

Definition 4. *Let* $l_i, s_i, l_o, s_o \in \mathbb{N}$*. The* path-length constraints $\mathbb{PL}_{l_i,s_i\to l_o,s_o}$ *are all finite sets of integer linear constraints over variables* $\{\check{l}^k \mid 0 \leqslant k < l_i\} \cup \{\check{s}^k \mid 0 \leqslant k < s_i\} \cup \{\hat{l}^k \mid 0 \leqslant k < l_o\} \cup \{\hat{s}^k \mid 0 \leqslant k < s_o\}$ *with the* $\leqslant$ *operator.*

One can also use constraints such as $x = y$, standing for both $x \leqslant y$ and $y \leqslant x$.

A path-length assignment fixes the values of the variables. When those values satisfy a path-length constraint, they are a *model* of the constraint.

Definition 5. *Let* $pl \in \mathbb{PL}_{l_i,s_i\to l_o,s_o}$ *and* ρ *be an assignment from a superset of the variables of* pl *into* $\mathbb{Z} \cup \{\infty\}$*. Then* ρ *is a* model *of* pl*, written as* $\rho \models pl$*, when* $pl\rho$ *holds i.e., when, by substituting, in* pl*, the variables with their values provided in* ρ*, one gets a tautological set of ground constraints.*

The *concretisation* of a path-length constraint is the set of denotations that induce pre- and post-state assignments that form a model of the constraint.

Definition 6. *Let* $pl \in \mathbb{PL}_{l_i,s_i\to l_o,s_o}$*. Its* concretisation $\gamma(pl)$ *is* $\big\{\delta \in \Delta_{l_i,s_i\to l_o,s_o} \mid$ *for all* $\sigma \in \Sigma_{l_i,s_i}$ *such that* $\delta(\sigma)\downarrow$ *we have* $\big(\widecheck{len}(\sigma) \cup \widehat{len}(\delta(\sigma))\big) \models pl\big\}$*.*

In [19] it is proved that γ is the concretisation map of an abstract interpretation [5] and sound approximations are provided for some instructions such as const, dup, new, load, store, add, getfield, putfield, ifeq and ifne, as well as for sequential and disjunctive composition. For instance, there is an abstraction $\mathit{getfield}_p^{\mathbb{PL}}\ f$, sound *w.r.t.* the concrete semantics of getfield f at p (Fig. 1). They remain sound after introducing arrays to the language. Fig. 2 reports the abstraction of array instructions. This defines a denotational fixpoint static analysis of Java bytecode, that approximates the path-length of local variables. We cannot copy from [19] the complete definition of the abstract semantics. We only observe that the analysis uses possible sharing [16] and reachability [13] analyses for approximating the side-effects of field updates and method calls.

Proposition 1. *The maps in Fig. 2 are sound w.r.t. those in Fig. 1.* □

$$
\begin{aligned}
\mathit{newarray}_p^{\mathbb{PL}}\ t\ n &= \mathit{Id}(\#l, \#s-n+1) \cup \{\check{s}^{\#s-1} \geqslant 0, \ldots, \check{s}^{\#s-n} \geqslant 0\} \\
\mathit{arraylength}_p^{\mathbb{PL}}\ t &= \mathit{Id}(\#l, \#s) \\
\mathit{arrayload}_p^{\mathbb{PL}}\ t &= \begin{cases} \mathit{Id}(\#l, \#s-2) \cup \{0 \leqslant \check{s}^{\#s-1} < \check{s}^{\#s-2}\} & \text{if } t = \texttt{int} \\ \mathit{Id}(\#l, \#s-2) \cup \{0 \leqslant \check{s}^{\#s-1} < \check{s}^{\#s-2}, \hat{s}^{\#s-2} \geqslant 0\} & \text{otherwise} \end{cases} \\
\mathit{arraystore}_p^{\mathbb{PL}}\ t &= \mathit{Id}(\#l, \#s-3) \cup \{0 \leqslant \check{s}^{\#s-2} < \check{s}^{\#s-3}\}
\end{aligned}
$$

Fig. 2. Path-length abstraction of the bytecodes from Fig. 1 that deal with arrays. $\#l, \#s$ are the number of local variables and stack elements at program point p. $\mathit{Id}(x, y) = \{\check{l}^i = \hat{l}^i \mid 0 \leqslant i < x\} \cup \{\check{s}^i = \hat{s}^i \mid 0 \leqslant i < y\}$.

We do not copy the abstract method call from [19], since it is complex but irrelevant here. Given the approximation pl of the body of a method m of class κ, it is a constraint $(\mathit{call}_p^{\mathbb{PL}} \kappa.m)(pl)$, sound *w.r.t.* $\mathsf{call}\ \kappa.m$ at program point p. Method calls in object-oriented languages can have more dynamic target methods, hence pl is actually the disjunction of the analysis of all targets. A restricted subset of targets can be inferred for extra precision [18].

Sequential composition of path-length constraints pl_1 and pl_2 matches the post-states of pl_1 with the pre-states of pl_2, through temporary, overlined variables. Disjuctive composition is used to join more execution paths.

Definition 7. *Let* $pl_1 \in \mathbb{PL}_{l_i, s_i \to l_t, s_t}$*,* $pl_2 \in \mathbb{PL}_{l_t, s_t \to l_o, s_o}$ *and* $T = \{\bar{l}^0, \ldots, \bar{l}^{l_t-1}, \bar{s}^0, \ldots, \bar{s}^{s_t-1}\}$*. The* sequential composition $pl_1 ;^{\mathbb{PL}} pl_2 \in \mathbb{PL}_{l_i, s_i \to l_o, s_o}$ *is the constraint* $\exists_T\, (pl_1[\hat{v} \mapsto \bar{v} \mid \bar{v} \in T] \cup pl_2[\check{v} \mapsto \bar{v} \mid \bar{v} \in T])$*. Let* $pl_1, pl_2 \in \mathbb{PL}_{l_i, s_i \to l_o, s_o}$*. Their* disjunctive composition $pl_1 \cup^{\mathbb{PL}} pl_2$ *is the polyhedral hull of* pl_1 *and* pl_2*.*

5 Path-Length with Expressions

Def. 4 defines path-length as a domain of numerical constraints over local or stack elements, which are the only program expressions that one can use in the constraints. That limitation can be overcome by adding variables that stand for more complex expressions, that allow the selection of fields or array elements.

Definition 8. *Given* $l \geqslant 0$*, the set of* expressions *over* l *local variables is* $\mathbb{E}_l = \{l^k \mid 0 \leqslant k < l\} \cup \{e.f \mid e \in \mathbb{E}_l \text{ and } f \text{ is a field name}\} \cup \{e_1[e_2] \mid e_1, e_2 \in \mathbb{E}_l\}$*. Given also* $s \geqslant 0$*, the expressions or stack elements are* $\mathbb{ES}_{l,s} = \mathbb{E}_l \cup \{s^i \mid 0 \leqslant i < s\}$*. When we want to fix a maximal depth* $k > 0$ *for the expressions, we use the set* $\mathbb{E}_l^k = \{e \in \mathbb{E}_l \mid e \text{ has depth at most } k\}$*.*

Definition 9. *Given* $\sigma = \langle l \parallel s \parallel \mu \rangle \in \Sigma_{\#l,\#s}$ *and* $e \in \mathbb{ES}_{\#l,\#s}$*, the* evaluation $[\![e]\!]\sigma$ of e in σ *is defined as* $[\![l^k]\!]\sigma = l^k$ *and* $[\![s^k]\!]\sigma = s^k$ *(*l^k *and* s^k *is an expression on the left and the value of the* k*th local variable or stack element on the right); moreover,* $[\![e.f]\!]\sigma = \mu([\![e]\!]\sigma)(f)$ *if* $[\![e]\!]\sigma \in \mathbb{L}$ *(undefined, otherwise);* $[\![e_1[e_2]]\!]\sigma = \mu([\![e_1]\!]\sigma)([\![e_2]\!]\sigma)$ *if* $[\![e_1]\!]\sigma \in \mathbb{L}$ *and* $[\![e_2]\!]\sigma \in \mathbb{Z}$ *(undefined, otherwise).*

Sec. 4 can now be generalised. Path-length assignments refer to all possible expressions, not just to local variables and stack elements (compare with Def. 3).

Definition 10. *Let* $\sigma = \langle l \,\|\, s \,\|\, \mu\rangle \in \Sigma_{\#l,\#s}$*. Its* pre-state path-length assignment *is* $\widecheck{len}(\sigma) = [\check{e} \mapsto len([\![e]\!]\sigma, \mu) \mid e \in \mathbb{ES}_{\#l,\#s}]$*. Its* post-state path-length assignment *is* $\widehat{len}(\sigma) = [\hat{e} \mapsto len([\![e]\!]\sigma, \mu) \mid e \in \mathbb{ES}_{\#l,\#s}]$*.*

Path-length can now express constraints over the value of expressions (compare with Def. 4); such expressions are actually numerical variables of the constraints.

Definition 11. *Let* $l_i, s_i, l_o, s_o \in \mathbb{N}$*. The set* $\mathbb{PL}_{l_i,s_i \to l_o,s_o}$ *of the* path-length constraints *contains all finite sets of integer linear constraints over the variables* $\{\check{e} \mid e \in \mathbb{ES}_{l_i,s_i}\} \cup \{\hat{e} \mid e \in \mathbb{ES}_{l_o,s_o}\}$*, using only the* $\leqslant$ *comparison operator.*

Def. 5, 6 and 7 remain unchanged; the abstractions in Fig. 2 and from [19] work over this generalised path-length domain, but do not exploit the possibility of building constraints over expressions. Such expressions must be selected, since $\mathbb{E}_{l,s}$ is infinite, in general. The analysis adds expressions on-demand, as soon as the analysed code uses them. Namely, consider the abstractions of the instructions that operate on the heap. They are refined by introducing expressions, as follows, by using definite aliasing, a minimum requirement for a realistic static analyser: $e_1 \sim_p e_2$ means that e_1 and e_2 are definitely alias at program point p.

Definition 12. *Let* $k > 0$ *be a maximal depth for the expressions considered below. From now on, the approximations on* $\mathbb{PL}$ *of getfield*$_p^{\mathbb{PL}}$ f *and putfield*$_p^{\mathbb{PL}}$ f *from [19] and arrayload*$_p^{\mathbb{PL}}$ *and arraystore*$_p^{\mathbb{PL}}$ *from Fig. 2 will be taken as refined by adding the following constraints:*

$$\textit{to getfield}_p^{\mathbb{PL}}\ f : \{\hat{s}^{\#s-1} = \widecheck{e.f} = \widehat{e.f} \mid e.f \in \mathbb{E}^k_{\#l} \textit{ and } e \sim_p s^{\#s-1}\}$$

$$\textit{to putfield}_p^{\mathbb{PL}}\ f : \left\{ \check{s}^{\#s-1} = \widehat{e.f} \;\middle|\; \begin{array}{l} e.f \in \mathbb{E}^k_{\#l},\ f \textit{ does not occur in } e, \\ e \sim_p s^{\#s-2} \textit{ and } f \textit{ has type } \texttt{int} \textit{ or array} \end{array} \right\}$$

$$\textit{to arrayload}_p^{\mathbb{PL}}\ t : \left\{ \hat{s}^{\#s-2} = \widecheck{e_1[e_2]} = \widehat{e_1[e_2]} \;\middle|\; \begin{array}{l} e_1[e_2] \in \mathbb{E}^k_{\#l}, \\ e_1 \sim_p s^{\#s-2} \textit{ and } e_2 \sim_p s^{\#s-1} \end{array} \right\}$$

$$\textit{to arraystore}_p^{\mathbb{PL}}\ t : \left\{ \check{s}^{\#s-1} = \widehat{e_1[e_2]} \;\middle|\; \begin{array}{l} e_1[e_2] \in \mathbb{E}^k_{\#l},\ e_1 \sim_p s^{\#s-3},\ e_2 \sim_p s^{\#s-2}, \\ e_1, e_2 \textit{ do not contain array subexpressions} \end{array} \right\}$$

Def. 12 states that getfield f pushes on the stack the value of $e.f$, where e is a definite alias of its receiver. Bytecode putfield f stores the top of the stack in $e.f$, where e is, again, a definite alias of its receiver. Similarly for arrayload and arraystore. Bytecodes putfield and arraystore avoid the introduction of expressions whose value might be modified by their same execution.

Proposition 2. *The maps in Def. 12 are sound w.r.t. those in Fig. 1.* □

Example 1. In the snippet of code from `util.linarg.DiagonalMatrix` at page 2, the compiler translates the expression `this.diagonal.length` at line 4 into

```
load 0               // load local variable this
getfield diagonal    // load field diagonal of this
arraylength double   // compute the length of this.diagonal
```

At the beginning $\#s = 1$, local 0 is `this`, local 1 is `newDiagonal` and local 2 is `i`, hence the latter is a definite alias of stack element 0, going to be compared against the value of `this.diagonal.length`. The next table reports the number $\#s$ of stack elements ($\#l = 3$ always), definite aliasing just before the execution of each instruction (self-aliasing is not reported) and its resulting abstraction:

instruction	$\#s$	definite aliasing	abstraction
load 0	1	$\{l^2 \sim s^0\}$	$\{\check{l}^0 = \hat{l}^0 = \hat{s}^1, \check{l}^1 = \hat{l}^1, \check{l}^2 = \hat{l}^2, \check{s}^0 = \hat{s}^0\}$
getfield diagonal	2	$\{l^2 \sim s^0, l^0 \sim s^1\}$	$\left\{\begin{array}{l} \check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{l}^2 = \hat{l}^2, \check{s}^0 = \hat{s}^0 \\ \hat{s}^1 = \widecheck{l^0.diagonal} = \widehat{l^0.diagonal} \end{array}\right\} (pl_1)$
arraylength double	2	$\{l^2 \sim s^0\}$	$\left\{\begin{array}{c} \check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{l}^2 = \hat{l}^2 \\ \check{s}^0 = \hat{s}^0, \check{s}^1 = \hat{s}^1 \end{array}\right\} (pl_2)$

The abstraction of `getfield diagonal` uses the definite aliasing information $l^0 \sim s^1$ to introduce the constraint $\hat{s}^1 = \widecheck{l^0.diagonal} = \widehat{l^0.diagonal}$ on expression `this.diagonal` (Def. 12). The sequential composition of the three constraints approximates the execution of the three bytecode instructions: $\check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{l}^2 = \hat{l}^2, \check{s}^0 = \hat{s}^0$. Unfortunately, it loses information about `this.diagonal`.

The approximation in Ex. 1 is imprecise since pl_1 (see Ex. 1) refers to $\widehat{l^0.diagonal}$, but pl_2 does not refer to $l^0.diagonal$ at all: hence their sequential composition does not propagate any constraint about it. To overcome this imprecision, one can include frame constraints in the abstraction of each instruction ins, stating, for *each* expression e whose value is not *affected* by ins, that its path-length does not change: $\check{e} = \hat{e}$. But this is impractical since, in general, there are infinitely many such expressions. Next section provides an alternative, finite solution.

6 Expressions and Side-Effects Information

Let us reconsider the sequential composition of pl_1 and pl_2 from Ex. 1. Since pl_1 refers to the expression $l^0.diagonal$, not mentioned in pl_2, we could define $pl_2' = pl_2 \cup \{\widecheck{l^0.diagonal} = \widehat{l^0.diagonal}\}$ and compute $pl_1;^{\mathbb{PL}} pl_2'$ instead of $pl_1;^{\mathbb{PL}} pl_2$. The composition will then propagate the constraints on $\widehat{l_0.diagonal}$. This redefinition of $;^{\mathbb{PL}}$ is appealing since it adds the frame condition $\widecheck{l^0.diagonal} = \widehat{l^0.diagonal}$ only for $l^0.diagonal$ *i.e.*, for the expressions that are introduced on-demand during the analysis. However, it is unsound when pl_2 is the abstraction of a piece of code that affects the value of $l^0.diagonal$ (for instance, it modifies l^0 or $diagonal$): the constraint $\widecheck{l^0.diagonal} = \widehat{l^0.diagonal}$ would not hold for all its concretisations. This leads to the addition of side-effect information to $\mathbb{PL}$.

Side-effects are modifications of leftvalues, that is, local variables, object fields or array elements. A local variable is modified when its value changes. A field f is modified when at least an object in memory changes its value for f. An array of type t is modified when at least an array of type t in memory changes.

Definition 13. *Let* $\delta \in \Delta_{l_i,s_i \to l_o,s_o}$ *and* $\sigma = \langle l \,\|\, s \,\|\, \mu \rangle \in \Sigma_{l_i,s_i}$. *Then* δ modifies local k in σ *iff*[7] $\sigma' = \langle l' \,\|\, s' \,\|\, \mu' \rangle = \delta(\sigma)\downarrow$ *and either* l'^k *does not exist or* $l^k \neq l'^k$. *It* modifies f in σ *iff* $\sigma' = \langle l' \,\|\, s' \,\|\, \mu' \rangle = \delta(\sigma)\downarrow$ *and there exists* $\ell \in dom(\mu)$ *where* $\mu(\ell)$ *is an object having a field* f *and either* $\mu'(\ell)\uparrow$, *or* $\mu'(\ell)$ *is not an object having a field* f, *or* $\mu(\ell)(f) \neq \mu'(\ell)(f)$. *It* modifies an array of type t in σ *iff* $\sigma' = \langle l' \,\|\, s' \,\|\, \mu' \rangle = \delta(\sigma)\downarrow$ *and there exists* $\ell \in dom(\mu)$ *where* $\mu(\ell)$ *is an array of type* t *and either* $\mu'(\ell)\uparrow$, *or* $\mu'(\ell)$ *is not an array of type* t, *or* $\mu(\ell).length \neq \mu'(\ell).length$, *or* $\mu(\ell)(i) \neq \mu'(\ell)(i)$ *for some index* $0 \leqslant i < \mu(\ell).length$.

It is now possible to define a more concrete abstract domain than in Def. 11, by adding information on local variables, fields and arrays that might be modified.

Definition 14. *Let* $l_i, s_i, l_o, s_o \in \mathbb{N}$. *The abstract domain* $\mathbb{PLSE}_{l_i,s_i \to l_o,s_o}$ *for path-length and side-effects contains tuples* $\langle pl \,\|\, L \,\|\, F \,\|\, A \rangle$ *where* $pl \in \mathbb{PL}_{l_i,s_i \to l_o,s_o}$ *(Def. 11),* L *is a set of local variables,* F *is a set of fields and* A *is a* minimal *set of types i.e., for all* $t, t' \in A$ *it is never the case that* $t < t'$.

A tuple $\langle pl \,\|\, L \,\|\, F \,\|\, A \rangle$ represents denotations that are allowed to modify locals in L, fields in F and arrays whose elements are compatible with some type in A:

Definition 15. *Let* $\langle pl \,\|\, L \,\|\, F \,\|\, A \rangle \in \mathbb{PLSE}_{l_i,s_i \to l_o,s_o}$. *Its* concretisation function *is* $\gamma(\langle pl \,\|\, L \,\|\, F \,\|\, A \rangle) = \{\delta \in \Delta_{l_i,s_i \to l_o,s_o} \mid$ **1)** $\delta \in \gamma(pl)$ *[Def. 6],* ***2)*** *if* δ *modifies local* k *in* σ *then* $l^k \in L$, ***3)*** *if* δ *modifies* f *in* σ *then* $f \in F$, ***4)*** *if* δ *modifies an array of type* t *in* σ *then* $t \leqslant t'$ *for some* $t' \in A\}$.

Proposition 3. $\mathbb{PLSE}_{l_i,s_i \to l_o,s_o}$ *is a lattice and the map* γ *of Def. 15 is the concretisation of a Galois connection from* $\Delta_{l_i,s_i \to l_o,s_o}$ *to* $\mathbb{PLSE}_{l_i,s_i \to l_o,s_o}$. □

The abstract semantics uses now Fig. 2, [19] and Def. 12 and adds side-effects. For method calls, callees in Java cannot modify the local variables of the caller.

Definition 16. *The approximations* $ins^{\mathbb{PLSE}}$ *are defined as* $ins_p^{\mathbb{PLSE}}\ k = \langle ins_p^{\mathbb{PL}} \,\|\, \{l^k\} \,\|\, \varnothing \,\|\, \varnothing \rangle$ *if* ins *is* store k *or* inc k c; $putfield_p^{\mathbb{PLSE}}\ f = \langle putfield_p^{\mathbb{PL}}\ f \,\|\, \varnothing \,\|\, \{f\} \,\|\, \varnothing \rangle$; $arraystore_p^{\mathbb{PLSE}}\ t = \langle arraystore_p^{\mathbb{PL}}\ t \,\|\, \varnothing \,\|\, \varnothing \,\|\, \{t\} \rangle$; $(call_p^{\mathbb{PLSE}}\ \kappa.m)(\langle pl \,\|\, L \,\|\, F \,\|\, A \rangle) = \langle (call_p^{\mathbb{PL}}\ \kappa.m)(pl) \,\|\, \varnothing \,\|\, F \,\|\, A \rangle$; $ins_p^{\mathbb{PLSE}} = \langle ins_p^{\mathbb{PL}} \,\|\, \varnothing \,\|\, \varnothing \,\|\, \varnothing \rangle$ *for all other* ins.

Proposition 4. *The maps in Def. 16 are sound w.r.t. those in Fig. 1.* □

Definition 17. *Let* $a = \langle pl \,\|\, L \,\|\, F \,\|\, A \rangle \in \mathbb{PLSE}_{l_i,s_i \to l_o,s_o}$. *Then* $e \in \mathbb{E}_{l_i}$ *is* affected *by* a *iff 1)* $e = l^k$ *and* $l^k \in L$, *or 2)* $e = e'.f$ *and* $f \in F$ *or* e' *is affected by* a, *or 3)* $e = e_1[e_2]$, *the type of* $e_1 \leqslant t \in A$ *or* e_1 *or* e_2 *is affected by* a.

Abstract compositions over $\mathbb{PLSE}$ use side-effect information to build frame conditions for expressions used in one argument and not affected by the other.

[7] In Java bytecode, local variables are identified by number and their amount varies across program points. Source code variable names are not part of the bytecode.

Program	Category	LoC	LoC w. Libs	Watchpoints	True Alarms
Snake&Ladder	game	794	17818	15	1
MediaPlayer	entertainment	2634	87368	28	2
EmergencySNRest	web service	3663	42540	36	2
FarmTycoon	game	4005	69659	1998	2
Abagail	mach. learn.	12270	49243	2986	126
JCloisterZone	game	19340	116858	590	49
JExcelAPI	scientific	34712	67944	2031	162
Colossus	game	77527	194994	1988	173

Fig. 3. The programs analyzed. **LoC** are the non-blank non-commented lines of source code; **LoC w. Libs** includes the lines of the libraries reachable and analyzed; **Watchpoints** is the number of arrayload or arraystore, whose bounds must be checked; **True Alarms** are index bound violations found by the analysis (*i.e.*, actual bugs).

Definition 18. *Let abstract elements* $a_1 = \langle pl_1 \parallel L_1 \parallel F_1 \parallel A_1 \rangle \in \mathbb{PLSE}_{l_i,s_i \to l_t,s_t}$*,* $a_2 = \langle pl_2 \parallel L_2 \parallel F_2 \parallel A_2 \rangle \in \mathbb{PLSE}_{l_t,s_t \to l_o,s_o}$*,* $U_1 = \{\check{e} = \hat{e} \mid e \in \mathbb{E}_{l_i}$ *is used in* pl_2 *and not affected by* $a_1\}$ *and* $U_2 = \{\check{e} = \hat{e} \mid e \in \mathbb{E}_{l_t}$ *is used in* pl_1 *and not affected by* $a_2\}$*. The* sequential composition $a_1 ;^{\mathbb{PLSE}} a_2 \in \mathbb{PLSE}_{l_i,s_i \to l_o,s_o}$ *is* $\langle (pl_1 \cup U_1) ;^{\mathbb{PL}} (pl_2 \cup U_2) \parallel L_1 \cup L_2 \parallel F_1 \cup F_2 \parallel \mathsf{maximize}(A_1 \cup A_2) \rangle$*, where* $\mathsf{maximize}(A) = \{t \in A \mid \neg \exists t' \in A \text{ such that } t < t'\}$*. Let* $a_1 = \langle pl_1 \parallel L_1 \parallel F_1 \parallel A_1 \rangle$*,* $a_2 = \langle pl_2 \parallel L_2 \parallel F_2 \parallel A_2 \rangle$ *be in* $\mathbb{PLSE}_{l_i,s_i \to l_o,s_o}$ *and* U_1, U_2 *be as above. The* disjunctive composition $a_1 \cup^{\mathbb{PLSE}} a_2$ *is* $\langle (pl_1 \cup U_1) \cup^{\mathbb{PL}} (pl_2 \cup U_2) \parallel L_1 \cup L_2 \parallel F_1 \cup F_2 \parallel \mathsf{maximize}(A_1 \cup A_2) \rangle$.

Proposition 5. *The compositions in Def. 18 are sound w.r.t. the corresponding concrete compositions on denotations [19].* □

Example 2. In Ex. 1, no instruction has side-effects, hence the last case of Def. 16 applies. The abstraction of getfield diagonal is now $a_1 = \langle pl_1 \parallel \varnothing \parallel \varnothing \parallel \varnothing \rangle$. That of the subsequent arraylength double is now $a_2 = \langle pl_2 \parallel \varnothing \parallel \varnothing \parallel \varnothing \rangle$ (pl_1 and pl_2 are given in Ex. 1). Expression $l^0.diagonal$ is used in pl_1 and is not affected by a_2. Hence (Def. 18) $U_1 = \varnothing$, $U_2 = \{\widecheck{l^0.diagonal} = \widehat{l^0.diagonal}\}$ and $a_1 ;^{\mathbb{PLSE}} a_2 = \langle pl_1 \parallel \varnothing \parallel \varnothing \parallel \varnothing \rangle ;^{\mathbb{PLSE}} \langle pl_2 \cup \{\widecheck{l^0.diagonal} = \widehat{l^0.diagonal}\} \parallel \varnothing \parallel \varnothing \parallel \varnothing \rangle = \{\check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{l}^2 = \hat{l}^2, \check{s}^0 = \hat{s}^0, \hat{s}^1 = \widecheck{l^0.diagonal} = \widehat{l^0.diagonal}\}$. The result refers to $l^0.diagonal$ now: the imprecision in Ex. 1 is overcome.

7 Experiments

Implementation. $\mathbb{PLSE}$ (Def. 14) needs to implement its elements $\langle pl \parallel L \parallel F \parallel A \rangle$, its abstract operations (Def. 16) and a fixpoint engine for denotational, bottom-up analysis. We have used the Julia analyser [17] and its fixpoint engine. Elements of $\mathbb{PLSE}$ use bitsets for L, F and A, since they are compact and with fast union (Def. 18). The pl component has been implemented twice: as bounded differences of variable pairs, by using zones (Ch. 3 of [10]) and as a

Program	**Zones Only**			**Zones + Poly**			**Zones + Exps**		
	Alarms	**Time**	**Mem**	**Alarms**	**Time**	**Mem**	**Alarms**	**Time**	**Mem**
Snake&Ladder	1	13	1.7	1	14	1.7	1	14	1.7
MediaPlayer	5	52	4.8	5	64	4.8	5	52	4.9
EmergencySNRest	2	26	3.0	2	30	3.0	2	26	3.1
FarmTycoon	17	51	4.3	17	54	4.7	9	53	4.4
Abagail	664	69	4.3	662	202	11.2	339	81	7.2
JCloisterZone	116	108	7.1	116	127	8.3	90	108	7.4
JExcelAPI	1061	95	4.8	1045	317	13.8	786	141	9.7
Colossus	597	312	9.7	*out of memory*			477	328	11.5

Fig. 4. Analysis results for the programs in Fig. 3. **Zones Only** uses zones only; **Zones+Poly** zones and polyhedra; **Zones+Exps** zones with expressions (Sec. 5 and 6). Julia issues index bound **Alarms**, bounded by **Watchpoints** in Fig. 3. **Time** is full analysis time, in seconds. **Mem** is the peak memory usage, in gigabytes.

hybrid implementation of zones and polyhedra, by using the Parma Polyhedra Library [3] for polyhedra. We use zones rather than the potentially more precise octagons, only for engineering reasons: zones are already available, tested and optimised in Julia. They cannot accomodate constraints such as those for `add` or `sub`, that refer to three variables (two operands and the result) and are dropped with zones. This keeps the analysis sound but reduces its precision. In the hybrid representation, instead, polyhedra represent them. A fixpoint is run for each strongly-connected code component. Polyhedra and zones have infinite ascending chains, hence widening [5, 3, 10] is used after 8 iterations. The cost of operations on polyhedra and zones depends on their dimensions *i.e.*, variables (locals, stack elements and expressions, in pre-state ($\check{v}$), post-state ($\hat{v}$) and overlined $\overline{v}$, see Def. 7). We have limited zones to 200 dimensions and polyhedra to 110; variables beyond that limit are projected away. This does not mean that the analysed programs have only up to 200 (or 110) variables: the limit applies at each given program point, not to the program as a whole. Since there are infinitely many expressions (Def. 12), we fixed a limit of 9. This does not mean that the analysis of a program considers 9 expressions only: it applies to each given program point. We fixed $k = 3$ in Def. 12. When, nevertheless, abstraction (Def. 12) or composition (Def. 18) generate more than 9 expressions, the implementation prefers those from a_2 in $a_1;^{\mathbb{PLSE}} a_2$ and drops those beyond the 9th.

Results. We used an Intel 8-core i7-6700HQ at 2.60 Ghz, OpenJDK Java 1.8.0_151 and 15 GB of RAM. Small to medium-size open-source third-party programs have been analysed, up to 195000 lines of code, cloneable from [14]. Fig. 3 reports their size, characteristics and number of index bound violations found by the analysis. The reachable libraries have been included and analysed, together with the application code. This is needed for the approximation of method calls to the library. However, warnings have been generated only on the application code. Fig. 4 reports the results. Programs have been first analysed as in [8, 5], with zones only (column **Zones Only**). Each alarm has been manually classified as true (*i.e.*, an array index bug) or false. True alarms range from 1 to 173 per program. If classification was impossible, since we do not fully understand the logic of the code or its invariants, alarms have been conservatively classified as false. Thus, column **True Alarms** in Fig. 3 is a lower bound on actual bugs. Comparing **Alarms** of **Zones Only** with **True Alarms** shows a major precision gap. To close it, we tried to exploit the extra precision of polyhedra through the hybrid use of zones *and* polyhedra. Column **Alarms** of **Zones+Poly** in Fig. 4 deceives our hopes: polyhedra hardly improve the precision, at the price of higher analysis time and memory footprint, up to an out of memory. Instead, columns **Zones+Exps** show that the technique of Sec. 6, with zones only, scales to all programs, with fewer alarms: precision benefits more from expressions than from polyhedra and expressions are cheaper than polyhedra *w.r.t.* memory usage. Fig. 4 reports full analysis times and peak memory usage during parsing of the code, construction of the control-flow graph and of the strongly-connected components, heap, aliasing and path-length analysis. The alarms are in [14], annotated as `TA` when they classify as true alarms. Note that the analysis has false positives but no false negatives (true bugs that the analysis does not find), since it is provably sound.

False Alarms that Disappear by Using Expressions. `Zones Only` issues false alarms for all examples in Sec. 1. They disappear with `Zones + Exps`. In the first example, the analyser uses a variable for the expression `this.diagonal`; in the second, for `this.data` and `this.data[i]`; in the third, for `this.this$0.h` and `this.this$0.h[i]`. Expressions are chosen automatically and on-demand.

True Alarms. In `jxl.biff.BaseCompoundFile` of JExcelAPI, Julia issues a true alarm at line 3 below[8], since the constructor is public and its argument `d` is arbitrary, hence might have less than `SIZE+1` elements[9]:

```
1 public PropertyStorage(byte[] d) {
2   this.data = d;
3   int s = IntegerHelper.getInt(this.data[SIZE], this.data[SIZE+1]); }
```

Julia issues a true alarm at line 4 of class `domain.Farm` of FarmTycoon, for a public method whose argument `options` is hence arbitrary. Very likely, the programmer should have written `options[pos]` here, instead of `options[1]`:

```
1 public static void objPrinter(String[] options) {
2   Storm[] objStorm = new Storm[options.length];
3   for (int pos = 0; pos < options.length; pos++)
4     objStorm[pos] = new Storm(Long.parseLong(options[1])); }
```

Julia issues a true alarm at line 2 of `edu.cmu.sv.ws.ssnoc.common.logging.Log` in EmergencySNRest, since the stack trace might be shorter than 4 elements (the documentation even allows `getStackTrace()` to be empty):

```
1 private static Logger getLogger() {
2   ... = Thread.currentThread().getStackTrace()[3].getClassName(); }
```

[8] Line numbers, conveniently starting at 1, do not correspond to the actual line numbering of the examples, which are simplified and shortened *w.r.t.* their original code.

[9] We assume that public entries can be called with any values, as also done in [15].

Julia issues true alarms from line 5 of `java.net.sf.colossus.webclient.WebClientSocketThread` in Colossus, where `fromServer` comes from a remote server and might contain too few tokens: it should be sanitised first:

```
1  String fromServer = getLine();
2  String[] tokens = fromServer.split(sep, -1)
3  String command = tokens[0]; // ok: split() returns at least one token
4  if (command.equals(IWebClient.userInfo)) {
5    int loggedin = Integer.parseInt(tokens[1]);
6    int enrolled = Integer.parseInt(tokens[2]);
7    ... String text = tokens[6]; ... }
```

False Alarms: Limitations of the Analysis. In `func.svm.SingleClassSequentialMinimalOptimization` of Abagail, Julia issues false alarms at line 8:

```
1  public SingleClass...Optimization(DataSet examples, ..., double v) {
2    v = Math.min(v, 1); ...
3    this.a = new double[examples.size()];
4    this.vl = v * examples.size(); int ivl = (int) this.vl;
5    int[] indices = ABAGAILArrays.indices(examples.size());
6    ABAGAILArrays.permute(indices);
7    for (int i = 0; i < ivl; i++)
8      this.a[indices[i]] = 1 / vl; }
```

It is $0 \leqslant \mathtt{i} < \mathtt{ivl} = \lfloor \mathtt{v} * \mathtt{examples.size()} \rfloor \leqslant \mathtt{examples.size}()$ and `ABAGAILArrays.indices(x)` yields an array of size `x`. Thus `indices[i]` is safe. Also `this.a[indices[i]]` is safe, since the elements of `ABAGAILArrays.indices(x)` range from 0 to `x` (excluded) and `permute()` shuffles them. Such reasonings are beyond the capabilities of our analysis.

Julia issues false alarms at lines 3 and 4 of `net.sf.colossus.util.StaticResourceLoader` in Colossus:

```
1  while (r > 0) {
2    byte[] temp = new byte[all.length + r];
3    for (int i = 0; i < all.length; i++) temp[i] = ...;
4    for (int i = 0; i < r; i++) temp[i + all.length] = ...; }
```

Here, Julia builds a constraint $\mathtt{temp} = \mathtt{all} + \mathtt{r}$. Since $\mathtt{r} > 0$, then `i` in the first loop is inside `temp`; since $0 \leqslant \mathtt{i} < \mathtt{r}$, the same holds in the second loop. Zones cannot express a constraint among three variables. Polyhedra can do it, but do not scale to the analysis of Colossus (Fig. 4).

8 Conclusion

The extension of path-length to arrays (Sec. 4) scales to array index bounds checking of real Java programs, but only with weaker abstractions than polyhedra, such as zones. Precision improves with explicit information about some expressions (Sec. 5 and 6). Experiments (Sec. 7) are promising. The analysis has limitations: it is unsound with unconstrained reflection or side-effects due to concurrent threads, as it is typical of the current state of the art of static analysers for full Java; also remaining false alarms (Sec. 7) show space for improvement.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Field-Sensitive Value Analysis by Field-Insensitive Analysis. In *World Congress on Formal Methods (FM)*, pages 370–386, The Netherlands, 2009.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. V. Ramírez-Deantes. From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In *Static Analysis Symposium (SAS)*, pages 100–116, Perpignan, France, 2010.
3. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
4. R. Bagnara, P. M. Hill, and E. Zaffanella. Applications of Polyhedral Computations to the Analysis and Verification of Hardware and Software Systems. *Theoretical Computer Science*, 410(46):4672–4691, 2009.
5. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252, 1977.
6. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée Analyzer. In *European Symposium on Programming (ESOP)*, pages 21–30, 2005.
7. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does Astrée Scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.
8. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Principles of Programming Languages (POPL)*, pages 84–96, Tucson, Arizona, USA, January 1978.
9. T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java^{TM} Virtual Machine Specification.* Financial Times/Prentice Hall, 2013.
10. A. Miné. *Weakly Relational Numerical Abstract Domains.* PhD thesis, École Polytechnique, Paris, France, 2004.
11. A. Miné. Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 54–63, Ottawa, Ontario, Canada, 2006.
12. A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
13. D. Nikolic and F. Spoto. Reachability Analysis of Program Variables. *Transactions on Programming Languages and Systems*, 35(4):14:1–14:68, 2013.
14. É. Payet and F. Spoto. Index Checking Experiments. Available at https://github.com/spoto/Index-Checker-Experiments.git, 2017.
15. J. Santino. Enforcing Correct Array Indexes with a Type System. In *Foundations of Software Engineering (FSE)*, pages 1142–1144, Seattle, WA, USA, 2016. ACM.
16. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *Static Analysis Symposium (SAS)*, pages 320–335, London, UK, September 2005.
17. F. Spoto. The Julia Static Analyzer for Java. In *Static Analysis Symposium (SAS)*, pages 39–57, Edinburgh, UK, 2016.
18. F. Spoto and T. P. Jensen. Class Analyses as Abstract Interpretations of Trace Semantics. *Transactions on Programming Languages and Systems*, 25(5):578–630, 2003.
19. F. Spoto, F. Mesnard, and É. Payet. A Termination Analyzer for Java Bytecode Based on Path-Length. *Transactions on Programming Languages and Systems*, 32(3):8:1–8:70, 2010.