Analyzing Usage of Data in Array Programs

Jan Haltermann

Dept. of Computer Science, Paderborn University, Germany
jfh@mail.upb.de

1 Introduction

Most static dataflow analyses, like constant propagation or live variable analysis, are designed for variables having primitive data types and are not applicable to arrays. For large arrays or in case that the length of an array is not known in advance, creating variables for each array index (array expansion) is infeasible or impossible. Treating the array as a single variable (array smashing) leads to imprecise results, when not all array elements are treated in the same way.

2 Approach

In this paper, we tackle the problem of analyzing programs containing arrays of unknown length and computing properties for their array elements by presenting two novelties: First, we provide a framework to analyze arrays by extending arbitrary program analyses for primitive variables to full arrays, based on the array segmentation domain presented by Cousot et al. [3]. Second, we develop the $\mathcal{C}_{\mathbb{LU}}$ analysis for identifying non-used array elements, as instance of the framework. To illustrate the framework and the analysis, we use the example given in Figure 1.

2.1 Framework for Analyzing Arrays

To infer properties for array elements in programs, we make use of the array segmentation domain defined by Cousot et al. [3]. Intuitively, an array segmentation splits an array into several segments, such that each element from the array is associated to exactly one segment. The information computed by an analysis is also associated with one segment and hence the full array is covered The array segmentation domain D is defined as follows:

$$D = [E : (\underbrace{\{e_1^0, \dots\}}_{B_0}\ p_0[?]_0\ B_1\ p_1[?]_1\ \ \dots\ \ p_{n-1}[?]_{n-1}\ \underbrace{\{e_1^n, \dots\}}_{B_n} \cup \square) \cup (\bot, \top)]^n$$

The bottom element is used to indicate unreachable path, the top element to avoid under-approximation analysis results. A segment describes an interval in the array, having a lower bound B_i of the segmentation being included and the upper bound B_{i+1} that is not included. Moreover, it contains some analysis information p_i from the underlying domain forming a complete lattice and a flag $'?'_i$, indicating if the segment may be empty. The segment bounds are simple binary expressions over integers, are ordered ascending and all expressions present in the same segment bound are assumed to haven an equal value. For example, the segmentation $\{0\}\ p_0?\ \{i\}\ p_1\ \{a.len()\}$ may be computed during the analysis of the example program shown in Figure 1. It states that the information p_0 holds for the interval $[0, i[$, that might be empty and p_1 for all elements in the interval $[i, a.len()[$, containing at least one element.

The transfer relation used to compute successor segmentation when analyzing a program mainly retains the relation between the expressions present in the segment bounds. When merging or comparing two segmentation, we apply a unification algorithm. It ensure that both

```
int sum(int[] a)
    int res = 0;
    for( int i = 0; i < a.len()−1 ; i++)
        res = res + a[i];
    return res;
```

Figure 1: A program, computing the sum of all array elements except the last one

segmentation contain identical segment bounds, to be able to apply the merge or comparison segment-wise, making use of the underlying domain. The unification algorithm tries to maintain as many segment bounds as possible while guaranteeing an unique ordering of the segment bounds in both segmentation. Moreover, it guarantees that the number of segment bounds in a segmentation is bounded above. Formal definitions for enhanced versions of both, transfer function and unification algorithm can be found in the Masters Thesis of Haltermann, where the basic ideas are taken from Cousot [4].

To compute more precises results, we extended Cousot et al.'s domain in several ways: First, we add the explicit symbol $\Box$ to indicate, that the array can be empty. Thereby, more precise results are computed by removing doubtful $'?'$'s. Second, we introduce the split-condition E together with using multiple segmentation for one program location. A split condition states in general an assumption on the array length, e. g. if it is less, equal or greater than a constant. By analyzing a segmentation w. r. t. the split condition, an analysis computes multiple different array segmentation for a single program location. Applying splitting allows us to improve the results, i. e. when analyzing programs containing branches having constant values in their conditions. Third, we added a strengthening mechanism using the path formula introduced by Beyer et al. [2] to enhance the performance of the analysis. Thereby, we can again remove doubtful $'?'$'s, reducing the over-approximation of the analysis for the computed results.

2.2 $\mathcal{C}_{\mathbb{LU}}$ Analysis: Instantiation to Analyze Array Content Usage

The combined location and usage analysis $\mathcal{C}_{\mathbb{LU}}$ is an instantiation of our framework detecting unused array elements. An array index at position i is considered as used, if there are two program executions started with same values for all input elements except the value at index i and returning different values. The underlying domain detecting usage consists of the two elements **U**sed and **N**ot-used, where $\bot = N \sqsubseteq U = \top$. The transfer function and merge operator designed for single elements are given in [4]. The analysis is designed as a may analysis, hence the set of used array elements is over-approximated. Initially, every $\mathcal{C}_{\mathbb{LU}}$ analysis is started with the empty segmentation $\{0\}N?\{a.len()\}$, meaning that all elements are not used or the array is empty. When applied to the running example from Figure 1, we obtain the segmentation $\{\{0\}\ U?\ \{a.len()-1,\ i\}\ N\ \{a.len()\},\ \Box\}$. Hence, we can state that all array elements in the potentially empty interval $[0, a.len()-1[$ are used and one element in the interval $[a.len()-1, a.len()[$ is not considered during computation, if the array is not empty.

In addition, we prove that any instantiation of the framework terminates, when providing a monotone transfer function for the underlying domain. We show that the analysis can only produce a finite number of segmentation, even for programs containing loops. During computation, a segmentation's length is bounded above by the number of statements on the longest, loop-free path leading to a location plus a constant value. This property of the framework is guaranteed by the unification procedure. Moreover, we show that the $\mathcal{C}_{\mathbb{LU}}$-analysis is correct in the sense that all results computed in fact over-approximates the set of used variables.

3 Implementation

We implemented the framework as a configurable analysis and the $\mathcal{C}_{\mathbb{LU}}$ analysis as instantiation in the CPAchecker[1] [1]. First evaluations on small, hand crafted examples containing loops and branches like in Figure 1 and patterns extracted by Xiao et al. [5] during their study of reducers, has lead to precise results. For instance, when applied to the running example, the $\mathcal{C}_{\mathbb{LU}}$ analysis computed a precise result in around one second.

4 Conclusion and Further Work

We presented a lightweight framework usable to easily lift analyses developed for single variables to arrays, additionally guaranteeing termination. The framework is based on the array segmentation domain introduced by Cousot et al., heavily extended to obtain more precise results. Moreover, we presented an instantiation of the framework to detect usage of array elements and demonstrated it on a small example. Both, the framework as well as the $\mathcal{C}_{\mathbb{LU}}$ analysis are implemented and evaluated using the CPAchecker.

In the future, we want to enhance the $\mathcal{C}_{\mathbb{LU}}$ analysis to be able to detect non-commutative usage of array elements. Moreover, we plan to investigate if using further analyses like interval analysis or constant value analysis, can either lead to more precise results or speed up the computation time.

References

[1] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In Werner Damm and Holger Hermanns, editors, *Proc. of 19th Int. Conf. on Computer-Aided Verification, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007.

[2] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. of 10th Int. Conf. on Formal Methods in Computer-Aided Design, FMCAD 2010*, pages 189–197. IEEE, 2010.

[3] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proc. of 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '11*, pages 105–118. ACM, 2011.

[4] Jan Haltermann. Analyzing data usage in array programs. Master's thesis, Paderborn University, 2019.

[5] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. Nondeterminism in mapreduce considered harmful? An empirical study on non-commutative aggregators in MapReduce programs. In *Proc. of 36th Int. Conf. on Software Engineering, ICSE '14*, pages 44–53. ACM, 2014.

[1] https://cpachecker.sosy-lab.org/