Inference of Progressive Loop Invariants for Array Programs

Master's Thesis Description

Nils Becker
Supervisors: Jérôme Dohrau, Alexander J. Summers, Prof. Dr. Peter Müller

Department of Computer Science, ETH Zurich, Switzerland

September 2019

1 Introduction

In order to guide the verification of a program many program verification tools require the user to provide *specifications*. These either allow the verifier to assume specific properties at a certain point in a program or instruct it to verify that a specific property holds at a certain point in a program. One form of specification that is oftentimes used to verify loops is an *inductive loop invariant*. The tool checks that this invariant holds before the loop is entered and that each iteration of the loop preserves this invariant. If both of these checks succeed we can then, by induction, assume that the invariant holds in any state that can be reached by executing arbitrarily many loop iterations.

In addition to *functional specifications* (e.g. constraints on the value of each array element), tools based on separation logic, such as the Viper tools [4], require *framing specifications*, in the form of *permissions*, indicating which memory locations are potentially accessed by a certain part of a program. A method for verifying the framing of certain categories of array-accessing loops without the need for user-supplied loop invariants was proposed by Dohrau et al. [3]. This approach first analyzes the permissions needed for a single loop iteration as a function of its environment (e.g. the value of an iteration variable). Based on these results it then generates a precondition for the loop, indicating which permissions are required to execute the entire loop, as well as a postcondition, indicating what permissions are left after executing the entire loop.

In contrast to many similar analyses this approach does not rely on a fixed-point analysis to determine the amounts of permissions required for different memory locations (though a fixed-point analysis is used to obtain constraints on the values of local variables). In such a fixed-point analysis the program state would be represented in an *abstract domain* and the program would be executed symbolically until a fixed-point is reached. To guarantee that a fixed-point is always reached precision beyond a certain point is sacrificed. E.g. if our abstract domain tracks the constraint $0 \leq i \leq 5$ and we execute a single loop iteration in which we increment i by 1 we obtain the new constraint $1 \leq i \leq 6$. To continue executing the loop we need to merge the original constraint with the new one.

```
1  var a = get_int_array(); var old_a = a.copy(); var i = 0
2  while (i < a.length)
3    invariant ∀x. x < i ? a[x] == old_a[x]+1 : a[x] == old_a[x]
4    invariant ∀x. i ≤ x < a.length ⟹ write_access(a[x])
5  {
6    a[i] += 1
7    i += 1
8  }
```

Fig. 1. A simple example of a loop iterating over an array and possible corresponding progressive loop invariants for that program. The invariant on line 3 summarizes the effects of iterations that have already been executed while the one on line 4 provides an upper bound on the iterations that will still be executed.

Instead of generating $0 \leq i \leq 6$ we may choose to drop the upper bound and generate $0 \leq i$ whereafter subsequent iterations of the loop will not change the constraint anymore i.e. we have reached a fixed-point.

Because it does not utilize a fixed-point analysis, one of the main advantages of the approach described above is that it does not require an abstract domain that can handle permission predicates to be designed. The analysis can, however, only infer pre- and postconditions for loops. Invariants, which would e.g. be able to express how the amount of permissions held after each iteration changes, cannot be inferred. These may e.g. be useful when combining its results with user supplied functional specifications or the results of other analyses. Moreover, it is imprecise for programs that access arrays at indices that are heap-dependent or that cannot be expressed in Presburger arithmetic. It is, also, oblivious to the order in which loop iterations are executed and, hence, requires a *soundness condition* to be satisfied that ensures that no loop iteration `exhale`s (gives away) permissions that are needed by any other iteration. The algorithm generates preconditions based on the results for a single loop iteration by requiring the maximum amount of permissions that may be needed for each array location, thus, combining the requirements for different iterations into a single expression. This approach cannot easily be extended to functional specifications since there is no easy way for combining the effects of array updates across multiple iterations (without knowing the order in which they are executed).

The goal of this thesis is to design an analysis to capture information about how the program state evolves at different points during the execution of a loop. The information captured by such an analysis will be more fine grained than that captured by the analysis described by Dohrau et. al and will, therefore, allow us to reason about the order in which the effects of a loop take place instead of just summarizing the overall effects in the form of a postcondition. In particular this analysis will infer *progressive loop invariants*, invariants that capture the effects of loop iterations that have already been executed as well as information about iterations that will happen in the future. Lines 3 and 4 of Fig. 1 show examples of what such invariants may look like. Similar to what Dohrau et al. described

the analysis will generalize the results for a single iteration over the part of the loop that has already been executed or the part that will happen in the future, as necessary. In addition to the direct applications of this analysis (e.g. obtaining information about the values in an array or checking loop termination by proving that the amount of permissions necessary to execute the remainder of the loop constantly decreases with each iteration) this analysis can be combined with the one described by Dohrau et al. to overcome some of the limitations described above (e.g. weakening the soundness condition described above by ensuring that permissions that are `exhale`d are only used by earlier loop iterations).

2 Core Goals

- Design a mechanism for inferring progressive loop invariants.
  - Manually explore feasibility of different approaches for approximating the effect of past loop iterations and underestimating the iterations that will still be executed.
  - Formalize the most successful approach.
- Explore use-cases of the analysis. Possible applications include:
  - Analyzing how array values develop during the execution of a loop and inferring corresponding loop invariants.
  - Analyzing what permissions are available at different points during loop execution and inferring corresponding loop invariants.
  - Checking loop termination.
- Implement at least one of theses analyses as an extension of the existing inference mechanism for loop pre- and postconditions.
- Evaluate the analysis on examples and well-known benchmarks (e.g. [1]).

3 Extension Goals

- Design and implement a mechanism for inferring constraints about array values and compare the results to existing works such as [2].
- Design and implement a mechanism for inferring inductive loop invariants as framing specifications of loops iterating over arrays and explore how the resulting information about the order in which iterations are executed can be used to apply the analysis to additional programs that do not satisfy the soundness condition described by Dohrau et al.
- Explore how the precision of the analysis can be improved for programs that access arrays at indices that are heap-dependent or that cannot be expressed in Presburger arithmetic.

References

1. SV-Benchmarks - Collection of verification tasks. https://github.com/sosy-lab/sv-benchmarks.

2. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In A. D. Gordon, editor, *Programming Languages and Systems*, pages 246–266, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
3. J. Dohrau, A. J. Summers, C. Urban, S. Münger, and P. Müller. Permission inference for array programs. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification (CAV)*, volume 10982 of *LNCS*, pages 55–74. Springer-Verlag, 2018.
4. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.