Automating Verification of Loops by Parallelization

Reiner Hähnle
Chalmers University of Technology, Sweden
reiner@chalmers.se

Coauthors: Tobias Gedell (Chalmers University of Technology, Sweden)

It is generally agreed upon that loops and recursive calls are the main bottleneck in formal software verification. The source of the problem is that loops and recursion are proof theoretically handled either with invariant rules or with induction. In both cases, it is necessary in general to strengthen invariants and induction hypotheses in order to make proofs go through. There are also many technicalities with those rules that make their application difficult. A number of heuristic techniques have been developed to guide induction proofs and to find appropriate induction hypotheses.

The context of the present work is formal verification of functional properties of sequential Java programs. Here the situation is aggravated by the fact that heuristic techniques have been developed for relatively simple functional programming languages and are not readily applicable to a complex, imperative, object-based language such as Java (similar comments apply to C, C++, or C#). Hence, not only is there a lack of heuristic techniques that help to automate proofs about loops in Java, but due to the complexity of loop rules in imperative languages user interaction involves a high amount of technical knowledge and is extremely expensive.

A recent divide-and-conquer technique for decomposition of induction proofs works for imperative programs, but it is targeted at simplifying user interaction rather than eliminating it. In order to deal automatically with loops in verification of Java-like languages there are not many options at present: abstraction and approximation are incomplete and in some scenarios even unsound. They impose also limits on what can be expressed in specifications. If the number of loop iterations is known and small then it is possible to use symbolic execution with finite unwinding. The state of art in Java verification is, however, that complex user interaction is unavoidable for most loops.

We present an automatic deductive verification technique that is applicable to many loops occurring in practically relevant Java programs. Like any automatic method it cannot handle all loops, but it is seamlessly integrated with a complete interactive verification system. In addition, it computes useful information even when it fails. Let us take a look at an example (where `e(i)` is a Java expression with an occurrence of `i`):

```
for (int i = 0; i < a.length; i++) a[i] = e(i);
```

The effect of this piece of code is simply to initialize the array `a` with the expression `e(i)` at index `i`. Since the length of `a` is in general unknown, it is not possible to deal with this loop by finite unwinding. An abstraction of this program has difficulties to record that the value `a.length` depends on `a`. On the other hand, in most cases it is overkill to use induction on such a simple problem. In order to describe the effect of such loops it is usually sufficient to be able to quantify universally over state update expressions that are performed in parallel. From a proof theoretic point of view, such quantified state modifiers can be handled by skolemization and simplification, hence, they are amenable to automated proof search.

In general, the initialization, guard and step expressions, as well as the loop body could be more complicated than in the example above. Our technique does not rely on the target program being in a particular syntactic form but, of course, we need to ensure that the effect of a loop is expressable as a quantified parallel update. This problem is closely related to loop vectorization and parallelization and it is possible to use notions developed in those fields. The main point is to exclude certain data dependencies. For example, in the case where `e(i)` is `a[i - 1]` the code above cannot be transformed into a quantified update, because the updates for each i cannot be performed in parallel.

The contribution of this paper is a deductive verification method for treating loops based on the ideas just sketched. Its main properties are:

[Robustness] The target program needs not to be in a particular syntactic form. This is achieved by computing the accumulated effect of the expressions and statements occurring in the loop by symbolic execution before checking the dependencies in the loop body.

[Soundness] There is an efficient test based on a static analysis that guarantees sound applicability.

[Automation] Proof theoretic treatment of the effect of loops is not by induction but by universally quantified state modification and is automatic.

[Integration] The method is seamlessly integrated with a complete interactive verification system. Even when a loop fails to be parallelizable, our method computes a symbolic constraint that characterizes when this is the case. This constraint can greatly simplify the remaining user interaction.

[Relevance] The method applies not only to a few academic examples, but to a substantial number of loops in realistic programs. An experimental evaluation of a number of realistic Java Card programs confirms this.