Static verification of program running time

CIS 673 course project report

Caleb Stanford

December 2016

Contents

1 Introduction

1.1 Total Correctness is Not Enough

At the most general level, the goal of formal verification is to prove that a given program, C, adheres to some specification S. Thus, in general the nature of verification is dependent on both the programming language (syntax of C) and the nature of the specification (syntax of S).

In the Hoare style[1], the specification S consists of a *pre-condition* P and a *post-condition* Q, which are both predicates (boolean functions) of the program state at a fixed time. We say that C is *correct* with respect to S if, when if the program is started in a program state satisfying the pre-condition P, and the program is executed, the post-condition Q is satisfied. If C does not halt, we distinguish between *total correctness* and *partial correctness*. Let h be the event that C halts. We may notate the difference between them as follows:

$$\textbf{Partial Correctness: } (P \wedge h) \to Q.$$
$$\textbf{Total Correctness: } P \to (h \wedge Q).^{2}$$

Less formally:

$$\textbf{Total Correctness} \equiv \textbf{Partial Correctness} + \textbf{Termination}.$$

Partial correctness is sufficient to catch all bugs in which the program, on a well-formed input, produces an incorrect output (where "incorrect" is relative to the specific application). In contexts where performance is a concern, however, partial correctness is not enough. A classic example is that a program which doesn't halt can be proven to satisfy any specification. In the following program, the programmer intends to add the numbers from 1 to `X`, but forgets to increment the loop counter `I`.

```
1  Total := 0
2  I := 1
3  while I <= X :
4     Total := Total + I
5  return Total
```

Unfortunately, this program still satisfies partial correctness with respect to its specification, say

$$\{\texttt{X} = n \wedge n > 0\} \quad C \quad \left\{\texttt{Total} = \frac{n(n+1)}{2}\right\}.$$

In fact, the program is partially correct with respect to *any* specification, because it never halts. Total correctness, in contrast, cannot be proven except for the correct specification. So total correctness has the advantage (and motivation) of catching unintended infinite loops in a program.

However, total correctness (while providing a stronger guarantee than partial correctness), does not protect against performance bugs in general. In practice, the difference between non-termination and termination in an unreasonable amount of time is negligible (i.e., not empirically observable). Consider a different program:

```
1  Total := 0
2  I := 1
3  while I <= 2000000000 :
4     if (I <= X) :
5        Total := Total + I
6     I := I + 1
7  return Total
```

[1] See Hoare's original paper, [1].

[2] Technically, one has to decide whether to take Q to be true or false in an execution which does not terminate. However, the formulas $(P \wedge h) \to Q$ and $P \to (h \wedge Q)$ evaluate to the same thing in either case.

It is not too difficult to prove that this new program is totally correct with respect to the same specification as before (if we assume that $n \leq 2000000000$). But the correctness in this case is not useful, because the program will not terminate quickly.

In practice, hidden blowup in the runtime of a program will not be so artificial. Even the difference between linear and quadratic runtime (such as might be caused by an accidental linear-time operation within a performance-critical loop) may be the difference between a program which terminates instantaneously and one which does not seem to terminate at all. However, such programs will still be totally correct with respect to their specification. In summary, if we are interested in proving termination at all, then we should really be interested not just in proving termination, but in proving *termination within a specified amount of time*. This argument is summarized in [2] by the following slogan:

Partial Correctness + Time > Total Correctness.

1.2 Project Description

For the project, I looked at two specific papers which use verification techniques to (statically) prove properties of program running time.

My main focus was the question of automatically inferring upper bounds on the runtime of programs. In practice, it is infeasible to require programmers to state and prove runtime bounds explicitly; performance guarantees of core software are usually not formally proven. Automatically generating correct and precise bounds on program runtime is a foolproof way to check for performance bugs at no additional cost to the user. The authors in [3] introduce a framework and methodology for automatic generation of these bounds, and implement their ideas in a tool called SPEED. In section 2, I have summarized the paper's results as I understand their relevance, and commented on how their framework might be applied to future research.

Another context where runtime guarantees are important is in the context of security, when a computation is carried out on sensitive data. Such computations can be subjected to *timing-based attacks*, which use the time the computation takes to terminate to get some information about the control flow that must have occured, and then deduce something about the sensitive data. The problem is, given an (allegedly safe) algorithm, to automatically verify that the runtime does not in fact depend on the sensitive parameters. I read a paper [4] which cleverly addresses this problem.

The techniques of [4] end up being quite different than those of [3]. A bound on runtime is unhelpful in protecting against timing-based attacks; instead, it is necessary to prove that the *exact* runtime doesn't depend on certain parameters. However, in fact finding the exact runtime is unnecessary, and [4] sidesteps this problem entirely; instead, the paper develops a model of what it means for a program to leak information, and defines security as a program which does not leak information. In fact, instead of dealing with the running time, they consider the *actual control flow* of a program to be leaked, and proving security means proving that given the actual control flow an attacker cannot deduce sensitive information.

Because the second paper [4] did not relate as much as I had hoped to the first paper [3], and because summarizing and interpreting the results of both [3] and [4] (without really being able to combine the two summaries) would have been difficult in only a few pages, I decided to focus on [3] exclusively for the remainder of this report.

2 SPEED

First without any automation, here is the general strategy to prove an upper bound $T \leq B$ on the runtime T of a program, where B is an expression in the input parameters. First, introduce increments to T as appropriate in the program (in general, T can be incremented by any amount at every command, but in practice it will be sufficient to only increment T at every iteration of each loop, so we are really bounding the number of loop iterations). Then, prove loop invariants on every loop such that every loop invariant logically implies $T \leq B$.

The hard part of this, that needs human input in general, is specifying the bound B, and specifying the loop invariants. In [3], the authors provide a way of automatically inferring a relatively tight bound B (with a proof of correctness). The method is implemented in the tool SPEED. Since *automatically* inferring and

proving the runtime of a program will not be possible in general, the method sacrifices a lot of generality for heuristics and a simple way of generating bounds that often works well in practice.

2.1 Goals for automatic bound generation

The goal of SPEED is to be an algorithm which automatically infers a bound on the number of loop iterations in simple programs. Specifically, instead of loop iterations we speak of **back-edges**. A program has a **back-edge** every time the control goes from a lower point in the program to a higher point. In other words, every time it jumps to the top of a loop. If there are multiple if branches inside a loop, we put a back-edge at the end of each one.

```
1   def f1(x0, y0, m, n) :
2     x := x0; y := y0
3     while (x < m) :
4       if (y < n) :
5         y++
6       else :
7         x++
```

Back-edges: In this code there are two, from line 5 to line 3, and from line 7 to line 3. We may denote them by (5,3) and (7,3).

Goal: *Given a program P, automatically infer an upper bound B in terms of the input parameters on the total number of times a back-edge is traversed.*

Note that whatever algorithm we come up with, it will fail on some program inputs, because deciding a bound in general will be undecidable. But we would like the algorithm to work well in practice. Specifically, it should satisfy these competing specific goals:

1. **Correct** We should never output an incorrect bound. (But we are allowed to fail.)
2. **Successful** The algorithm should not fail on too many programs in practice; it should produce a bound for a significant percentage of programs.

   Note that stating bounds for most programs requires use of max, and for programs with greater than linear runtime, use of $\cdot$. So a minimum requirement is that we are able to infer bounding expressions which use $\max, \cdot, +$.
3. **Precise** More than just producing a bound, it should *produce a tight bound.* The bound should not be too much higher than the actual runtime of a program.
4. **Efficient** Our algorithm must run quickly enough to work on real code.

2.2 Procedure

Generating the bound

Given a program, the first step is to assign a counter to each back-edge. We may use the same counter multiple times, but allowing multiple counters is sometimes necessary for a linear invariant to be generated. This is true of the example program `f1`.

`f1` has two back-edges; we modify it by assigning to the back-edges counters c_1 and c_2, set them to 0 at the start of the program, and increment them at the corresponding back-edge. From here, the linear invariant generation tool gives us invariants $\{y_0 + c_2 = y \wedge y \leq n\}$ and $\{x_0 + c_1 = x \wedge x \leq m\}$ which hold at the two back-edges. These two invariants imply bounds on c_1 and c_2 at the back-edges, namely $c_2 \leq n - y_0$ and $c_1 \leq m - x_0$, respectively.

```
def f1(x0, y0, m, n) :
   x := x0; y := y0; c1 := 0; c2 := 0
   while (x < m) :
      if (y < n) :
         y++; c2++ {y0 + c2 = y ∧ y ≤ n}
                ⟹ {c2 ≤ n − y0}
      else :
         x++; c1++ {x0 + c1 = x ∧ x ≤ m}
                ⟹ {c1 ≤ m − x0}
```

Why do we need two counters? With only one counter c, the linear invariant generation tool could find that $x_0 + y_0 + c = x + y$ holds at both back-edges, as does $x \leq m$. But $y \leq n$ only holds at the second back-edge; at the first back-edge, we instead need $y \leq \max(n, y_0)$. But the linear invariant generation tool cannot come up with $y \leq \max(n, y_0)$ since that is not a linear constraint on y.

So we use two separate counters to make the linear generation tool able to give us bounds. Altogether, we want to bound the number of back-edges traveled total, $T = c_1 + c_2$. Note that we cannot say $c_1 \leq m - x_0$ unless that back-edge is traversed at least once, so $T \leq (m - x_0) + (n - y_0)$ is not a correct bound. But we can say that either c_1 is 0, or it is bounded by $m - x_0$, i.e. it is bounded by $\max(0, m - x_0)$. So the bound we automatically infer is

$$T \leq \max(0, m - x_0) + \max(0, n - y_0).$$

(Note that there are better bounds possible, for example $\max(0, m - x_0 + \max(0, n - y_0))$. Since the method is automatic, it doesn't consider a more general form for the bound.)

This approach is all that we need to generate bounds if the runtime of the program is linear in the input. However, SPEED is able to generate multiplicative bounds, like $\max(0, m) \cdot \max(0, n)$. The way to do this is more clever. SPEED considers the possibility of adding to the program *counter dependencies*: if c_j depends on c_i, then every time c_i is incremented c_j is set back to 0 again. This is where products come in. If we are able to bound c_i by B_i whenever it is incremented, and c_j by B_j whenever it is incremented, then overall c_i back-edges are traversed at most $\max(0, B_i)$ times in total, and c_j back-edges are traversed at most

$$(\max(0, B_i) + 1) \cdot \max(0, B_j)$$

times in total.

Here is an example of this:

```
def f3(m, n) :
x := 0; y := 0; c1 := 0; c2 := 0
while (x < m) :
   if (y < n) :
      y++; c2++ {y = c2 ∧ y ≤ n}
             ⟹ {c2 ≤ n}
   else :
      y = 0
      x++; c1++; c2 = 0 {x = c1 ∧ x ≤ m}
                     ⟹ {c1 ≤ m}
```

The bound $c_2 \leq n$ holds whenever it is incremented, and $c_1 \leq m$ holds whenever it is incremented. So the overall bound on the program running time is

$$\max(m, 0) + (\max(m, 0) + 1) \cdot \max(n, 0).$$

Search space

The procedure given up to now explains how to get a bound given assignments of counters to every back-edge, and dependencies between the counters. This defines a **search space** of all possible counter-assignments and counter-dependencies. Given any state in the search space, the procedure given either generates a bound, or fails (if the linear invariant generation tool fails). The goal is to search through the search space and find the *best* bound. The size of the space is exponential, so we must search through the space efficiently.

However, given two bounds generated by the procedure, since they involve max it is not easy to tell which is a better bound. The authors of SPEED instead choose to go off of two heuristics: (1) fewer counters tends to lead to a better bound; (2) fewer dependencies tends to lead to a better bound. They then simply search through the search space for an assignment of counters to back-edges and dependencies between counters for which the linear invariant generation tool succeeds, and for which the number of counters and dependencies are minimal. The algorithm for doing this is greedy.

2.3 Key Techniques

This procedure, implemented in SPEED, achieves a good balance between the 4 goals given in 2.2. While the techniques in the paper are introduced in order to create SPEED, they are based on much more general high-level ideas. This same general framework could be used to design other tools.

Here is how each of the goals was addressed:

1. To achieve **correctness**, SPEED makes use of a linear invariant generation tool, and only provide bounds which are implied by the invariants generated by the tool. SPEED will fail if the invariant generation tool fails to find an invariant, or if the invariant found does not imply a bound.

   It seems to me that using an invariant generation tool of some sort (or requiring invariants from the programmer) is unavoidable. In order for a generated bound on the program's runtime to be correct, it is necessary and sufficient that the bound is true at every back-edge. If the bound is to be proven correct, it seems unavoidable that we need a loop invariant at every back-edge.

2. One big key to SPEED being **successful** (able to generate a bound at all) is that it actually incorporates the sizes of data structures into its bounds. For example, SPEED internally has a semantic specification of `Len` L for lists L–that it increases by one when you append an element, and decreases by one when you remove an element. The invariant generation tool is able to use this specification to incoporate `Len` into generated loop invariants, and in turn into the generated bounds. I did not discuss this in the procedure.

3. To achieve **precision**, we saw that SPEED uses a multiple-counter technique that generates bounds involving max, $+$, and $\cdot$. This form of the bounds generated is general enough to be precise in practice.

4. Finally, **efficiency**. Recall that the context is this:

   - We have some **search space** of modified programs that we look through.
   - For each modified program, we are able to produce a bound, or fail.
   - We try to find an optimal bound in the search space.

   Efficiency was achieved in two ways: (i) by restricting the search space so that it is manageable enough (but it will still be exponential), and (ii) instead of finding the optimal bound, which would require searching the entire space and being able to compare any two bounds, using a heuristic to find a bound that is likely close to optimal in practice, but not necessarily actually optimal. (The heurstic was, namely, fewer counters and fewer dependencies.)

2.4 Research ideas

What I am most interested in is using this same general framework to automatically generate bounds in a different way. Specifically, the framework is:

1. Define some search space of counter-modified programs (perhaps allowing different sorts of relationships between the counters, rather than just the basic dependencies);

2. Use an invariant tool to generate bounds on the counters, given the counter-modified program;

3. Use a heuristic to search through the space of all counter-modified programs and find a close-to-optimal bound.

With respect to (1), can we generalize the counter-placement to produce bounds of a more general form? For example, how would a bound like $\max(0, m + \max(0, n))$ arise? It does not come out of the counter placement used by SPEED. What about a bound like $n \log n$ – how might we automatically infer that with counter placement?

With respect to (2), can we use a more sophisticated invariant generation tool? Alternatively, we can drop the requirement that the whole process be automatic, and require some user input for the loop invariants. This would open up the way for more much sophisticated bounds to be generated.

With respect to (3), is there a better heuristic, or a way of actually comparing bounds, instead of just using a heuristic? For instance, it might be natural to plug in some random specific values, like $m, n = 100, 1000, 10000$ into two bounding expressions and just see which one seems to be lower, and that would be extremely an efficient heuristic.

References

[1] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[2] Eric CR Hehner. Specifications, programs, and total correctness. *Science of Computer Programming*, 34(3):191–205, 1999.

[3] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *ACM SIGPLAN Notices*, volume 44, pages 127–139. ACM, 2009.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. *URL: https://fdupress. net/files/ctverif. pdf*, 2016.