

Proving Termination

Aziem Chawdhary

Queen Mary, University of London

Introduction

Program termination is crucial to ensuring that systems code can always react. For example, it is absolutely essential that a device driver dispatch routine eventually returns to its caller. Automatically proving the termination of programs has proved to be very challenging. Of course it is well known that proving termination of a program is in general undecidable. However recent work has suggested that for certain classes of practical programs proving termination is a viable task [3, 4]. In this short abstract we outline some work we did in [1]

We have developed an algorithm which can take any program analysis and build a termination analyser using its results. Termination analysers are tools that prove as many termination lemmas as possible as quickly as possible. These termination lemmas are of the form:

“This program point is not visited infinitely often”

Rather than looking at the whole program and asking whether it terminates, we go line by line asking if this line is visited infinitely often. This makes the task of proving termination much easier. With our method, we can take any abstract interpretation based program analysis and produce a termination analysis using the results of the program analysis. Because we can use any abstract interpretation, we are producing many different termination analysers, by converting invariance analyses into termination analyses.

1. Some Background Information

It has been known for some time that mapping a program to a well-founded relation is a way of proving program termination. However taking an entire program and trying to map it to a well-founded relation is a difficult task, which is why total correctness in Hoare logics is difficult for example. However, recent results have managed to break down the problem of finding well-founded relations as noted below. A relation is well-founded if and only if it contains no countable infinite descending chains

1.1 Podelski-Rybalchenko Result

We use a recent result [6] as the foundation of our termination analysers. We view the program as a relation. To prove termination we need to show that this relation is well-founded. Traditionally, proving the a relation is well-founded is difficult, but the Podelski-Rybalchenko result makes it easier for us. The result states :

A relation R is well-founded iff
$$R^+ \subseteq T_1 \cup \dots \cup T_n$$
where T_i are all well-founded

So to check if a relation R is well-founded, we just need to find a series of other smaller well-founded relations T_i , such that the transitive closure of R is contained within them. This breaks

down the problem of checking well-foundedness into checking well-foundedness of smaller relations, which is much easier.

To build these T_i relations we are going to use the results of a program analysis.

Program Analysis

Program analysis, or static analysis, is a procedure which can calculate properties of a program without actually executing the program. A simple example is calculating the parity of every variable at each program point of a program:

```
1: x=12;  
2: y=0;  
3: while z>0 do  
4:   x=x+1;  
5:   y=y-1;  
6:   z=z-1  
7: done
```

Loc	Inv
1	x {+}
2	x {+}, y {0},
3	x {+}, y {-,0}, z {+}
4	x {+}, y {-,0}, z {+}
5	x {+}, y {-}, z {+}
6	x {+}, y {-}, z {+,0}

A more interesting analysis may calculate numerical information about variables . The examples in this paper use the Octagon Domain [5]

Loc	Inv
1	$x = 12$
2	$x = 12, y = 0$
3	$x \geq 12, y \leq 0, z \geq 1$
4	$x \geq 13, y \leq 0, z \geq 1$
5	$x \geq 13, y \leq -1, z \geq 1$
6	$x \geq 13, y \leq -1, z \geq 0$

Slightly more formally a program analysis uses

- a domain D to represent concrete states of memory. For examples $x = 1, y = 2$ might be represented as $x \geq 0 \wedge y \geq 0$ in a numerical domain.
- We also have transfer functions $F : D \rightarrow D$, which symbolically simulate what each command in the program does by operating on the abstract domain. The transfer functions can be seen as abstract versions of program statements. They mimic what the concrete commands do on concrete states of memory, but operate on the abstract domains.
- The program analysis starts at some initial abstract state $d_0 \in D$ and keeps applying the transfer functions to produce an abstract state at each program point until the abstract states are no longer changed by repeated applications of the transfer functions. This process is guaranteed to terminate due the mathematical properties of the domains and transfer functions.

The Algorithm

We now outline the algorithm very briefly. Given a program P and an program analysis D_a on domain D , we perform the following:

1. Run the analysis D_a on P
2. For every cutpoint in the program

- (a) We seed the analysis result at that cutpoint. Seeding basically converts an invariant into a relation between states. We do this by adding auxiliary variables and equalities into the invariant. For example, if we have an invariant $x - y \geq 0$, then seeding will introduce two new variables x_s, y_s and add the equalities $x_s = x \wedge y_s = y$ to get $x_s = x \wedge y_s = y \wedge x - y \geq 0$. The intuition is that x_s, y_s record the variables value before we do the next step. The seeded invariant can be viewed as a relation in the following way:

$$\{(s, t) \mid s(x) = t(x) \wedge s(y) = t(y) \wedge t(x) - t(y) \geq 0\}$$

s, t are two program states, where they have the same value for x and y , with $x - y \geq 0$.

- (b) Having seeded, we now run the analysis again on the seeded invariant to get another invariant. The values of x, y may have changed, but we know the relationship between their current values and their previous values because of the auxiliary variables we introduced. Thus we have a relation from the previous state at the cutpoint and the current state.
- (c) We then check that the second generated invariant is well-founded. If any of the seeded invariants are not well-founded, then we can conclude that the program may diverge. On the other hand, if all the seeded invariants are well-founded, then the program must be terminating.

Example

We now present a small example. We will prove the termination of the following small program. According to our intuition, this program fragment should terminate, as either x or y is decremented (the means non-deterministic choice) until either one is 0 or less.

```

1 while (x>0 and y>0) do
2   if (*) then
3     x = x - 1;
4   else
5     y = y - 1;
6 done

```

1. We will first perform an analysis to get numerical information for each variable after each program point:

1	$x \geq 1, y \geq 1$
2	$x \geq 1, y \geq 1$
3	$x \geq 0, y \geq 1$
4	$x \geq 1, y \geq 1$
5	$x \geq 1, y \geq 0$
6	?

2. Notice that we only really need to check that line 2 is not visited infinitely often. If we can prove that, then we have proven that the code fragment terminates.

We take the invariant at line 2, and convert it into a relation by seeding it: $x \geq 1, y \geq 1, x_s = x, y_s = y$

We then run the analysis again using the above state as a starting point. This will affect the values of x and y , but we took a snapshot of their previous values by using x_s and y_s

3. The results we get are:

1	...
2	$x \geq 1, y \geq 1$ $x_s - x \geq 1, y_s - y \geq 0$
	$x \geq 1, y \geq 1$ $x_s - x \geq 0, y_s - y \geq 1$
3	...

Notice that we know have two invariants are line 2, this is because one invariant has captured the information that x has been decremented and the other that y has been decremented.

4. We need to check whether the two invariants are well-founded.
 - $x \geq 1, y \geq 1, x_s - x \geq 1, y_s - y \geq 0$ The assertion $x_s - x \geq 1$ states that the value of x has gone down, as it is now definitely lower than the old seeded value x_s . So this relation states that x always goes down, but the assertion $x \geq 1$ states that x cannot go lower than 1, so the relation is well-founded.
 - A similar argument holds for y in the second invariant.

Conclusion

We have presented a very brief overview of the algorithm we have developed in [1]. We can take any program analysis and produce a termination analyser. We have produced two tools, with one being based on the Octagon Domain program analysis [5]. The tools we have produced are, as far as we know, the fastest termination analysers to date. We have analysed various algorithms, including algorithms taken from Windows device drivers. The full table of results is available in our paper.

References

- [1] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Invariance analyses from variance analyses. In *Principles of Programming Languages 2007*, 2007.
- [2] Josh Berdine, Byron Cook, Dino Distefano, and Peter O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV'06: Conference on Computer Aided Verification*, 2006.
- [3] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI'06: Programming Language Design and Implementation*, 2006.
- [4] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *CAV'06: Conference on Computer Aided Verification*, 2006.
- [5] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006. (to appear).
- [6] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS'2004: Logic in Computer Science*, pages 32–41. IEEE, 2004.

□