Proving Termination and Computational Complexity of Computer Programs

Anton Dergunov*

CSEdays 2013

In computability theory, the *halting (or termination) problem* can be stated as follows: "given a description of a computer program decide whether the program will always finish running or could potentially execute forever." A termination proof plays a critical role in formal verification. *Partial correctness* requires that an output of an algorithm is correct. *Total correctness* additionally requires that an algorithm terminates.

Alan Turing proved that a general algorithm to solve the halting problem can not exist. However, this does not mean that we are *always unable* to prove termination. New tools that are able to automatically prove the correctness of software are being developed.

We represent a computer program by a set of its states S, a set of its possible initial states $I \subseteq S$ and a transition relation $R \subseteq S \times S$ between states that the program can make during execution. Formally we prove a program termination by proving that its transition relation is *well-founded*, meaning that it does not permit infinite sequences of states $s = s_1, s_2, ...$ where $s_i \in S$ and $R(s_i, s_{i+1})$ [1]. The popular method to show that a relation $R \subseteq S \times S$ is well founded is to find a structure preserving map (homeomorphism) from structure (R, S) to some *well-ordered set* $(\geq, T)$. The structure $(\geq, T)$ forms a well order iff it is a total order and every nonempty subset of S has a least element. Such maps are called *ranking functions*.

There are several tools to prove termination of computer programs. TERMINATOR program [2] automatically searches for the relevant ranking function to prove termination, while in DAFNY it must be specified explicitly. DAFNY [3] is a programming language with built-in specification constructs that are used to verify correctness of programs. It employs *decreases annotations* to prove that programs terminate. A decreases annotation specifies a ranking function (*termination measure*) which values become strictly smaller each time a loop is traversed or a recursive method is called. This value is bounded so that it does not decrease forever. Several kinds of values can be used in decreases annotations, such as natural numbers or sets of values which have natural lower bounds (0 and empty set correspondingly). DAFNY proves that the termination measure gets smaller on each iteration. In simple cases DAFNY is able to guess termination measure, otherwise it must be specified explicitly, as in a method to search in a binary tree:

```
method Search(x: int) returns(found: bool)
    decreases ReachableNodes;
{
    if (x = data)                        {found := true;}
    else if (left ≠ null ∧ x < data)     {found := left.Search(x);}
    else if (right ≠ null ∧ x > data)    {found := right.Search(x);}
    else                                 {found := false;}
}
```

*Independent Researcher

In this example `ReachableNodes` is a member variable of a tree node class that stores a set of nodes reachable from the that node. The search method is guaranteed to terminate, because by construction the number of nodes reachable from the child nodes is strictly smaller than the number of nodes reachable from the parent node. DAFNY proves that the body of that method satisfies the decreases annotation.

The contribution of this article is the proposal to generalize proving termination to proving computational complexity in DAFNY. The idea is to verify not only that the termination measure decreases, but also the pace of its change. No special annotations exist in DAFNY for this functionality. But we can use existing annotations. For example, we can prove that the worst case time complexity of binary search algorithm is logarithmic:

```
0  method binarySearch(a: array<int>, key: int) returns(index: int)
1      requires a ≠ null;
2  {
3      var low      := 0; var high     := a.Length;
4      var old_low := 0; var old_high := high * 2;
5
6      while (low < high)
7          invariant high ≤ a.Length;
8          decreases (high − low);
9          invariant high = low ∨ (old_high − old_low) / (high − low) ≥ 2;
10     {
11         old_low  := low; old_high := high;
12
13         var mid := low + (high − low) / 2;
14         if (key > a[mid])       {low := mid + 1;}
15         else if (key < a[mid]) {high := mid;}
16         else                    {index := mid; return;}
17     }
18
19     index := −1;
20 }
```

Line 8 specifies termination measure that becomes strictly smaller each time a loop is traversed. In line 9 we specify that the termination measure reduces at least twice each iteration. DAFNY verifies that the body of this method satisfies the specification, thus proving the complexity of the method. Special keywords can be added to DAFNY to reduce boilerplate code to prove computational complexity.

Acknowledgements. This article is based on results of a student project with Omer Subasi, Krasnoshtan Dmytro, Georgiy Savchenko and Pavel Ajtkulov at Microsoft Summer School in Software Engineering and Verification held in Moscow in 2011. The project was supervised by Ben Livshits and Stephan Tobies.

References

[1] Byron Cook. Principles of program termination. *Engineering Methods and Tools for Software Safety and Security*, 22:161, 2009.

[2] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 415–426, New York, NY, USA, 2006. ACM.

[3] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer Berlin Heidelberg, 2010.