# Proving program termination

In contrast to popular belief, proving termination is not always impossible

Byron Cook Andreas Podelski Andrey Rybalchenko

## ABSTRACT

After Turing proved the halting problem undecidable in 1936, many considered the dream of automatic termination proving to be impossible. While not refuting Turing's original result, recent research advances make practical termination proving tools possible.

## Introduction

The *program termination problem*, also known as the *uniform halting problem*, can be defined as follows:

> Using only a finite amount of time, determine whether a given program will always finish running or could execute forever.

This problem rose to prominence before the invention of the modern computer, in the era of Hilbert's *Entscheidungsproblem*[1]: the challenge to formalize all of mathematics and use algorithmic means to determine the validity of all statements. In hopes of either solving Hilbert's challenge, or showing it impossible, logicians began to search for possible instances of *undecidable* problems. Turing's proof [38] of termination's undecidability is the most famous of those findings.[2]

The termination problem is structured as an infinite set of queries: to solve the problem we would need to invent a method capable of accurately answering either "terminates" or "doesn't terminate" when given any program drawn from this set. Turing's result tells us that any tool that attempts to solve this problem will fail to return a correct answer on at least one of the inputs. No number of extra processors nor terabytes of storage nor new sophisticated algorithms will lead to the development of a true oracle for program termination.

Unfortunately, many have drawn too strong of a conclusion about the prospects of automatic program termination proving and falsely believe that that we are *always unable* to prove termination, rather than more benign consequence that we are *unable to always* prove termination. Phrases like *"but that's like the termination problem"* are often used to end discussions that might otherwise have led to viable partial solutions for real but undecidable problems.

While we cannot ignore termination's undecidability, if we develop a slightly modified problem statement we can build useful tools. In our new problem statement we will still require that a termination proving tool always return answers that are correct, but we will not necessarily require an answer. If the termination prover cannot prove or disprove termination, it should return "unknown".

> Using only a finite amount of time, determine whether a given program will always finish running or could execute forever, *or* return the answer "unknown".

This problem can clearly be solved, as we could simply always return "unknown". The challenge is to solve this problem while keeping the occurrences of the answer "unknown" to within a tolerable threshold, in the same way that we hope web browsers will *usually* succeed to download webpages, although we know that they will sometimes fail. Note that the principled use of "unknown" in tools attempting to solve undecidable or intractable problems is increasingly common in computer science, *e.g.*, in program analysis, type systems, and networking.

In recent years, powerful new termination tools have emerged that return "unknown" infrequently enough that they are useful in practice [35]. These termination tools can automatically prove or disprove termination of many famous complex examples such as Ackermann's function or McCarthy's 91 function as well as moderately-sized industrial examples such as Windows device drivers. Furthermore, entire families of industrially useful termination-like properties—called *liveness properties*—such as "Every call to `lock` is eventually followed by a call to `unlock`" are now automatically provable using termination proving techniques [12,29]. With every month, we now see more powerful applications of automatic termination proving. As an example, recent work has demonstrated the utility of automatic termination proving to the problem of showing concurrent algorithms to be non-blocking [20]. With further research and development, we will see more powerful and more scalable tools.

We could also witness a shift in the power of software, as techniques from termination proving could lead to tools for other problems of equal difficulty. Whereas in the past a software developer hoping to build practical tools for solving something related to termination might have been frightened off by a colleague's retort *"but that's like the termination problem"*, perhaps in the future the developer will instead adapt techniques from within modern termination provers in order to develop a partial solution to the problem of interest.

The purpose of this article is to familiarize the reader with the recent advances in program termination proving,

[1]In English: "decision problem".

[2]There is a minor controversy as to whether or not Turing proved the undecidability in [38]. Technically he did not, but termination's undecidability is an easy consequence of the result that is proved. A simple proof can be found in [36].

and catalog the underlying techniques for those interested in adapting them techniques to other domains. We also discuss current work and possible avenues for future investigation. Concepts and strategies will be introduced informally, with citations to original papers for those interested in more detail. Several sidebars are made available for readers with backgrounds in mathematical logic.

```
1    x := input();
2    y := input();
3    while $x > 0$ and $y > 0$ do
4        if input() = 1 then
5            x := x − 1;
6            y := y + 1;
7        else
8            y := y − 1;
9        fi
10   done
```

**Figure 1: Example program. User-supplied inputs are gathered via calls to the function input(). We assume that the variables range over integers with arbitrary precision (in other words, not 64-bit or 32-bit integers). Assuming that the user always eventually enters in a value when prompted via input(), does the program terminate for all possible user-supplied inputs? (The answer is provided in a footnote below.)**

## Disjunctive termination arguments

Thirteen years after publishing his original undecidability result, Turing proposed the now classic method of proving program termination [39]. His solution divides the problem into two parts:

**Termination argument search:** Find a potential *termination argument* in the form of a function that maps every program state to a value in a mathematical structure called a well-order. We will not define well-orders here, the reader can assume for now that we are using the natural numbers (*a.k.a.* the positive integers).

**Termination argument checking:** prove the termination argument to be valid for the program under consideration by proving that result of the function decreases for every possible program transition. That is, if $f$ is the termination argument and the program can transition from some state $s$ to state $s'$, then $f(s) > f(s')$.

The reader with a background in logic may be interested in the formal explanation contained in the sidebar.

A well-order can be thought of as a terminating program—in the example of the natural numbers, the program is one that counts from some initial value in the natural numbers down to 0. Thus, no matter which initial value is chosen the program will still terminate. Given this connection between well-orders and terminating programs, in essence Turing is proposing that we search for a map from the program we are interested in proving terminating into a program known to terminate such that all steps in the first program have analogous steps in the second program. This map to a well-order is usually called a *progress measure* or a *ranking function* in the literature. Until recently, all known methods of proving termination were in essence minor variations on the original technique.

The problem with Turing's method is that finding a single, or *monolithic*, ranking function for the whole program is typically difficult, even for simple programs. In fact, we are often forced to use ranking functions into well-orders that are much more complex than the natural numbers. Luckily, once a suitable ranking function has been found, checking validity is in practice fairly easy.

The key trend that has led towards current progress in termination proving has been the move away from the search for a *single* ranking functions and towards a search for a *set* of ranking functions. We think of the set as a choice of ranking functions and talk about a *disjunctive* termination argument. This terminology refers to the proof rule of *disjunctively well-founded transition invariants* [31]. The recent approaches for proving termination for general programs [3,4,9,12,14,26,32] are based on this proof rule. The proof rule itself is based on Ramsey's theorem [34], and it has been developed in the effort to give a logical foundation to the termination analysis based on size-change graphs [24]. The principle it expresses appears implicitly in previously developed termination algorithms for rewrite systems, logic and functional programs, see [10,15,17,24].

The advantage to the new style of termination argument is that it is usually easier to find, because it can be expressed in small, mutually independent pieces. Each piece can be found separately or incrementally using various known methods for the discovery of monolithic termination arguments. As a trade-off, when using a disjunctive termination argument, a more difficult validity condition must be checked. This difficulty can be mitigated thanks to recent advances in assertion checking tools (as discussed in a later section).

*Example using a monolithic termination argument.* Consider the example code fragment in Figure 1. In this code the collection of user-provided input is performed via the function input(). We will assume that the user always enters in a new value when prompted. Furthermore, we will assume for now that variables range over possibly-negative integers with arbitrary precision (that is, mathematical integers as opposed to 32-bit words, 64-bit words, etc.). Before reading further, please answer the question: "Does this program terminate, no matter what values the user gives via the input() function?". The answer is given below.[3]

Using Turing's traditional method we can define a ranking function from program variables to the natural numbers. One ranking function that will work is $2x + y$, though there are many others. Here we are using the formula $2x + y$ as shorthand for a function that takes a program configuration as its input and returns the natural number computed by looking up the value of x in the memory, multiplying that by 2 and then adding in y's value—thus $2x+y$ represents a mapping from program configurations to natural numbers. This ranking function meets the constraints required to prove termination: the valuation of $2x+y$ when executing at line 9 in the program will be strictly one less than its valuation during the same loop iteration at line 4. Furthermore, we know that the function always produces natural numbers (thus it is a map into a well-order), as $2x + y$ is greater than 0 at

[3] The program does terminate.

**Note for the reader with a background in logic**

Formally, proving program termination amounts to proving the program's transition relation $R$ to be well-founded. If $(S, \geq)$ is a well-order then $>$ is a well-founded relation. Furthermore, any map $f$ into $S$ defines a well-founded relation, by lifting $>$ via $f$, i.e., $\{(s,t) \mid f(s) > f(t)\}$. Turing's method [39] of proving a program's transition relation $R$ well-founded amounts to finding a map $f$ into a well-order, which defines a termination argument $T = \{(s,t) \mid f(s) > f(t)\}$. To prove the validity of $T$ we must show $R \subseteq T$. From the well-foundedness of $T$ and the fact that every sub-relation of a well-founded relation is well-founded follows that $R$ is well-founded.

In this article we are using the phrase *disjunctive termination argument* to refer to a disjunctively well-founded transition invariant [31]. This is a finite union $T_1 \cup \ldots \cup T_n$ of well-founded relations that contains $R^+$, which is the transitive closure of the transition relation of the program, as a superset, i.e.,

$$R^+ \subseteq T_1 \cup \ldots \cup T_n \ .$$

Usually, each $T_1, \ldots, T_n$ will be constructed as above via some map into a well-order. Note that the non-reflexive transitive closure (the $+$ in $R^+$) is crucial. It is not sufficient to show that $R \subseteq T_1 \cup \ldots \cup T_n$, as the union of well-founded relations is not guaranteed to be well-founded. It is the transitive closure that makes checking the subset inclusion more difficult in practice.
The recent approaches for proving termination for general programs [3, 4, 9, 12, 14, 32] are based on the proof rule of disjunctively well-founded transition invariants. The proof rule itself is based on Ramsey's theorem [34], and it has been developed in the effort to give a logical foundation to the termination analysis based on size-change graphs [24]. The principle expressed by the proof rule appears implicitly already in previously developed termination algorithms for rewrite systems and logic and functional programs, see [10, 15, 17, 24].

lines 4 through 9.

Automatically proving the validity of a monolithic termination argument like $2\mathsf{x} + \mathsf{y}$ is usually easy using tools that check verification conditions (*e.g.* SLAM [2]). However, as mentioned above, the actual search for a valid argument is famously tricky. As an example, consider the case in Figure 2, where we have replaced the command "y := y + 1;" in Figure 1 with "y := input();". In this case no function into the natural numbers exists that suffices to prove termination; instead we must resort to a *lexicographic* ranking function (a ranking function into ordinals, a more advanced well-order than the naturals).

*Example using a disjunctive termination argument.* Following the trend towards the use of disjunctive termination arguments, we could also prove the termination of Figure 1 by defining an argument as the unordered finite collection of measures x and y. The termination argument in this case should be read as

x goes down by at least 1 and is larger than 0

```
1   x := input();
2   y := input();
3   while $x > 0$ and $y > 0$ do
4       if input() = 1 then
5           x := x − 1;
6           y := input();
7       else
8           y := y − 1;
9       fi
10  done
```

**Figure 2: Example program, similar to Figure 1 where the command "y := y + 1;" replaced with "y := input();". No ranking function into the natural numbers exists that can prove the termination of this program.**

*or*

y goes down by at least 1 and is larger than 0

We have constructed this termination argument with two ranking functions: x and y. The use of "*or*" is key: the termination argument is modular because it is easy to enlarge using additional measures via additional uses of "*or*". As an example, we could enlarge the termination argument by adding "*or* $2\mathsf{w} - \mathsf{y}$ goes down by at least 1 and is greater than 1000". Furthermore, as we will see in a later section, independently finding these pieces of the termination argument is easier in practice than finding a single monolithic ranking function.

The expert reader will notice the relationship between our disjunctive termination argument and complex lexicographic ranking functions. The advantage here is that we do not need to find an order on the pieces of the argument, thus making the pieces of the argument independent from one another.

The difficulty with disjunctive termination arguments in comparison to monolithic ones is that they are more difficult to prove valid: for the benefit of modularity we pay the price in the fact that the termination arguments must consider the transitions in all possible loop unrollings and not just single passes through a loop. That is to say: the disjunctive termination argument must hold not only between the states before and after any single iteration of the loop, but before and after *any number* of iterations of the loop (1 iteration, 2 iterations, 3 iterations, etc). This is a much more difficult condition to automatically prove. In the case of Figure 1 we can prove the more complex condition using techniques described later.

Note that this same termination argument now works for the tricky program in Figure 2, where we replaced "y := y + 1;" with "y := input();". On every possible unrolling of the loop we will still see that either x or y has gone down and is larger than 0.

To see why we cannot use the same validity check for disjunctive termination arguments as we do for monolithic ones, consider the slightly modified example in Figure 3. For every single iteration of the loop it is true that either x goes down by at least one and x is greater than 0 or y goes down by at least one and y is greater than 0. Yet, the program does not guarantee termination. As an example input sequence that triggers non-termination, consider 5, 5, followed by 1, 0, 1, 0, 1, 0, .... If we consider *all possible unrollings* of the loop, however, we will see that after two iterations it is possible (in the case that the user supplied the inputs 1 and 0 during the two loop iterations) that neither x nor y went down, and thus the disjunctive termination argument is not valid for the program in Figure 3.

```
x := input();
y := input();
while x > 0 and y > 0 do
        if input() = 1 then
            x := x − 1;
            y := y + 1;
        else
            x := x + 1;
            y := y − 1;
        fi
done
```

**Figure 3: Another example program. Does it terminate for all possible user-supplied inputs?**

## Argument validity checking

```
1  if y ≥ 1 then
2      while x > 0 do
3          assert(y ≥ 1);
4          x := x − y;
5      done
6  fi
```

**Figure 4: Example program with an assertion statement in line 3.**

While validity checking for disjunctive termination arguments is more difficult than checking for monolithic arguments, we can adapt the problem statement such that recently developed tools for proving the validity of assertions in programs (*i.e.* SLAM [2]).

An assertion statement can be put in a program to check if a condition is true. For example, **assert**$(y \geq 1)$; checks that $y \geq 1$ after executing the command. are violated, but We can use an assertion checking tool to formally investigate at compile time whether the conditions passed to assertion statements always evaluate to true. For example, most assertion checking tools will be able to prove that the **assert** statement at line 3 in Figure 4 never fails. Note that compile-time assertion checking is itself an undecidable problem, although it is technically in an easier class of difficulty than termination.[4]

The reason that assertion checking is so important to termination is that the validity of disjunctive termination arguments can be encoded as an assertion statement, where the statement fails only in the case that the termination argument is not valid. Once we are given an argument of the form $T_1$ *or* $T_2$ *or* ... *or* $T_n$, to check validity we simply want to prove the following statement.

[4]Checking validity of an assertion statement is an undecidable but co-recursively enumerable problem, whereas termination is neither r.e. nor co-r.e. problem.

```
copied := 0;
x := input();
y := input();
while x > 0 and y > 0 do
    if copied = 1 then
        assert(oldx ≥ x + 1 and oldx > 0);
    elsif input() = 1 then
        copied := 1;
        oldx := x;
        oldy := y;
    fi
    if input() = 1 then
        x := x − 1;
        y := y + 1;
    else
        y := y − 1;
    fi
done
```

**Figure 5: Encoding of termination argument validity using the program from Figure 1 and the termination argument "x goes down by at least one and is larger than 0". The black code comes directly from Figure 1. The code in red implements the encoding of validity with an assertion statement.**

> Each time an execution passes through one state and then through another one, $T_1$ *or* $T_2$ *or* ... *or* $T_n$ holds between these two states. That is, there *does not exist* a pair of states, one being reachable from the other, possibly via the unrolling of a loop, such that neither $T_1$ nor $T_2$ nor ... $T_n$ holds between this pair of states.

This statement can be verified a program transformation where we introduce new variables into the program to record the state before the unrolling of the loop and then use an assertion statement to check that the termination argument always holds between the current state and the recorded state. If the assertion checker can prove that the assert cannot fail, it has proved the validity of the termination argument. We can use encoding tricks to force the assertion checker to consider *all* possible unrollings.

To see such an example, look at Figure 5, where we have used the termination argument "x goes down by at least one and x is greater than 0" using the encoding given in [14]. The new code (introduced as a part of the encoding) is given in red, whereas the original program from Figure 1 is in black. We make use of an extra call to input() to decide when the unrolling begins. The new variables oldx and oldy are used for recording a state. Note that the assertion checker must consider all values possibly returned by input() during its proof, thus the proof of termination is valid for any starting position. This has the effect of considering any possible unrolling of the loop. After some state has been recorded, from this point out the termination argument is checked using the recorded state and the current state. In this case the assertion can fail, meaning that the termination argument is not valid.

If we were to attempt to check this condition in a naive way (for example, by simply executing the program) we would never find a proof for all but the most trivial of cases. Thus, assertion checkers must be cleverly designed to find proofs about all possible executions without actually executing all of the paths. A plethora of recently developed techniques now make this possible. Many recent assertion checkers are designed to produce a path to a bug in the case that the assertion statement cannot be proved. For example, a path leading to the assertion failure is $1 \to 2 \to 3 \to 4 \to 5 \to 7 \to 8 \to 9 \to 10 \to 11 \to 12 \to 16 \to 17 \to 4 \to 5 \to 6$. This path can be broken into parts, each representing a different phases of the execution: the prefix-path $1 \to 2 \to 3 \to 4$ is the path from the program initial state to the recorded state in the failing pair of states. The second part of the path $4 \to 5 \to \ldots 5 \to 6$ represents how we reached the current state from the recorded one. That is: this is the unrolling found that demonstrates that the assertion statement can fail. What we know is that the termination argument does not currently cover the case where this path is repeated forever.

See Figure 6 for a version using the same encoding, but with the valid termination argument:

x goes down by at least 1 and is larger than 0

*or*

y goes down by at least 1 and is larger than 0.

This assertion cannot fail. The fact that it cannot fail can be proved by a number of assertion verification tools.

```
1   copied := 0;
2   x := input();
3   y := input();
4   while x > 0 and y > 0 do
5       if copied = 1 then
6           assert( (oldx ≥ x + 1 and oldx > 0)
7                           or
8                   (oldy ≥ y + 1 and oldy > 0)
9                 );
10      elsif input() = 1 then
11          copied := 1;
12          oldx := x;
13          oldy := y;
14      fi
15      if input() = 1 then
16          x := x − 1;
17          y := y + 1;
18      else
19          y := y − 1;
20      fi
21  done
```

**Figure 6: Encoding of termination argument validity using the program from Figure 1 and the termination argument "x goes down by at least one and is larger than 0 or y goes down by at least one and is larger than 0". The black code comes directly from Figure 1. The code in red implements the encoding of validity with an assertion statement.**

## Finding termination arguments

In the previous section we saw how we can check a termination argument's validity via a translation to a program with an assertion statement. We now discuss known methods for finding *monolithic* termination arguments.

*Rank function synthesis.* In some cases simple ranking functions can be automatically found. We call a ranking function simple if it can be defined by a linear arithmic expression (e.g., $-3x - 2y + 100$). The most popular approach for finding this class of ranking function uses a result from Farkas [16] together with tools for solving linear constraint systems. See [11] or [30] for examples of tools using Farkas' lemma. Many other approaches for finding ranking functions for different classes of programs have been proposed—see [1, 6–8, 19, 37]. Tools for the synthesis of ranking functions are sometimes applied directly to programs, but more frequently they are used (on small and simplified program fragments) internally within termination proving tools for suggesting the single ranking functions that appear in a disjunctive termination argument.

*Termination analysis.* Numerous approaches have been developed for finding disjunctive termination arguments in which—in effect—the validity condition for disjunctive termination arguments is almost guaranteed to hold by construction. In some cases— [3] for example—to prove termination we need only check that the argument indeed represents a set of measures. In other cases, such as [24] or [26], the tool makes a one-time guess as to the termination argument and then checks it using techniques drawn from abstract interpretation.

Consider the modified program in Figure 7. The termination strategy described in [3,32] essentially builds a program like this and then applies a custom program analysis to find the following candidate termination argument:

$$(\mathsf{copied} \neq 1)\ \textbf{or}$$
$$(\mathsf{oldx} \geq \mathsf{x} + 1,\ \mathsf{oldx} > 0,\ \mathsf{oldy} > 0,\ \mathsf{x} \geq 0,\ \mathsf{y} > 0)\ \textbf{or}$$
$$(\mathsf{oldx} \geq \mathsf{x},\ \mathsf{oldy} \geq \mathsf{y} + 1,\ \mathsf{oldx} > 0,\ \mathsf{oldy} > 0,\ \mathsf{x} > 0,\ \mathsf{y} \geq 0)$$

for the program at line 4—meaning that we could pass this complex expression to the assertion at line 4 in Figure 7 and know that the assertion cannot fail. We know that this statement is true of any unrolling of the loop in the original Figure 1. What remains is to prove that each piece of the candidate argument represents a measure that decreases—here we can use rank function synthesis tools to prove that $\mathsf{oldx} \geq \mathsf{x} + 1$ **and** $\mathsf{oldx} > 0 \ldots$ represents the measure based on x. If each piece between the **or**s in fact represents a measure (with the exception of $\mathsf{copied} \neq 1$ which comes from the encoding) then we have proved termination.

One difficulty with this style of termination proving is that, in the case that the program doesn't terminate, the tools can only report "unknown", as the techniques used inside the abstract interpretation tools have lost so much detail that it is impossible to find a non-terminating execution from the failed proof and then prove it non-terminating. The advantage when compared to other known techniques is that it is much faster.

*Finding arguments by refinement.* Another method for discovering a termination argument is to follow the approach of [14] or [9] and search for counterexamples to (possibly invalid) termination arguments and then refine them based on new ranking functions found via the counterexamples.

```
1    copied := 0;
2    x := input();
3    y := input();
4    while x > 0 and y > 0 do
5        if copied = 1 then
6            skip;
7        elsif input() = 1 then
8            copied := 1;
9            oldx := x;
10           oldy := y;
11       fi
12       if input() = 1 then
13           x := x − 1;
14           y := y + 1;
15       else
16           y := y − 1;
17       fi
18   done
```

**Figure 7: Program prepared for abstract interpretation**

Recall Figure 5, which encoded the invalid termination argument for the program in Figure 1, and the path leading to the failure of the assertion: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 16 \rightarrow 17 \rightarrow 4 \rightarrow 5 \rightarrow 6$. Recall that this path represents two phases of the program's execution: the path to the loop, and some unrolling of the loop such that the termination condition doesn't hold. In this case the path $4 \rightarrow 5 \rightarrow \ldots 5 \rightarrow 6$ represents how we reached the second failing state from the first. This is a counterexample to the validity of the termination argument, meaning that the current termination argument does not take this path and others like it into account.

If the path can be repeated forever during the program's execution then we have found a real counterexample. Known approaches ( [21], for example) can be used to try and *prove* that this path can be repeated forever. In this case, however, we know that the path cannot be repeated forever, as y is decremented on each iteration through the path and also constrained via a conditional statement to be positive. Thus this path is a *spurious counterexample* to termination and can be ruled out via a refinement to the termination argument. Again, using rank function synthesis tools we can automatically find a ranking function that demonstrates the spuriousness of this path. In this case a rank function synthesis tool will find y, meaning that the reason that this path cannot be repeated for ever is that "y always goes down by at least one and is larger than 0". We can then refine the current termination argument used in Figure 5:

x goes down by at least 1 and is larger than 0

with the larger termination argument:

x goes down by at least 1 and is larger than 0

*or*

y goes down by at least 1 and is larger than 0

We can then check the validity of this termination argument using a tool such as Impact on program in Figure 6. Impact can prove that this assertion never fails, thus proving the termination of the program in Figure 1.

**Note for the reader with a background in logic**

We give a brief summary of implementation strategies based on disjunctive termination arguments deployed by the recent termination checkers:

**Refinement [9, 14]:** In [14], the termination argument begins with $\emptyset$. We first attempt to prove that $R^+ \subseteq \emptyset$. When this proof fails, rank function synthesis is applied to the witness, thus giving a refinement $T_1$ to the argument, which is then rechecked $R^+ \subseteq \emptyset \cup T_1$. This process is repeated until a valid argument is found or a real counterexample is found.

In [9], the termination argument $T$ is constructed following the structure of the transition relation $R = R_1 \cup \cdots \cup R_m$ by using a ranking function synthesis procedure, which is used to compute a well-founded over-approximation $\mathit{WF}(X)$ of a binary relation $X$. The initial candidate $T = \mathit{WF}(R_1) \cup \cdots \cup \mathit{WF}(R_m)$ is extended with $\mathit{WF}(\mathit{WF}(R_i) \circ R_j)$ and so on until the fixpoint is reached.

**Variance analysis [3, 32]:** As described in some detail in this article, the approach from [3, 32] uses program transformations and abstract interpretation for invariants to compute an over approximation $T_1, T_2, \ldots T_n$ such that $R^+ \subseteq T_1 \cup T_2 \cup \ldots \cup T_n$. It then uses rank function synthesis to check that each $T_i$ is well founded.

In contrast to the refinement-based methods, variance analysis always terminates, but may return "don't know" in cases when a refinement-based method succeeds.

## Further directions

With fresh advances in methods for proving the termination of sequential programs that operate over mathematical numbers we are now in the position to begin proving termination of more complex programs, such as those with dynamically allocated data-structures, or multi-threading. Furthermore, these new advances open up new potential for proving properties beyond termination, and finding conditions which would guarantee termination. We now discuss these avenues of future research and development in some detail.

*Dynamically allocated heap.* Consider the C loop in Figure 8, which walks down a list and removes links with data elements equaling 5. Does this loop guarantee termination? What termination argument should we use?

The problem here is that there are no arithmetic variables in the program from which we can begin to construct an argument—instead we would want to express the termination argument over the lengths of paths to `NULL` via the `next` field. Furthermore, the programmer has obviously intended for this loop to be used on acyclic singly-linked lists,

```
c = head;
while (c != NULL) {
    if (c−>next != NULL && c−>next−>data == 5) {
        t = c−>next;
        c−>next = c−>next−>next;
        free(t);
    }
    c = c−>next;
}
```

**Figure 8: Example C loop over a linked-list data-structure with fields `next` and `data`.**

```
x := 10;
while x > 9 do
        x := x − 2^32;
done
```

**Figure 9: Example program demonstrating non-termination when variables range over fixed-width numbers. The program terminates if x ranges over arbitrary size integers, but repeatedly visits the state where x = 10 in the case that x ranges over 32-bit unsigned numbers.**

but how do we know that the lists pointed to by `head` will always be acyclic? The common solution to these problems is to use *shape analysis* tools (which are designed to automatically discover the shapes of data-structures) and then to create new auxiliary variables in the program that track the sizes of those data structures, thus allowing for arithmetic ranking functions to be more easily expressed—examples include [4, 5, 25]. The difficultly with this approach is that we are now dependent on the accuracy and scalability of current shape analysis tools—to date the best known shape analysis tool [40] supports only lists and trees (cyclic and acyclic, singly- and doubly-linked) and scales only to relatively simple programs of size less than 30,000 LOC. Furthermore, the auxiliary variables introduced by methods such as [25] sometimes do not track enough information in order to prove termination (for example, imagine a case with lists of lists in which the sizes of the nested lists are important). In order to improve the state-of-the-art for termination proving of programs using data structures, we must develop better methods of finding arguments over data structure shapes, and we must also improve the accuracy and scalability of existing shape analysis tools.

*Bit vectors.* In the examples used until now we have considered only variables that range over mathematical numbers. The reality is that most programs use variables that range over fixed-width numbers, such as 32-bit integers or 64-bit floating-point numbers, with the possibility of overflow or underflow. If a program uses only fixed-width numbers and does not use dynamically allocated memory, then termination proving is decidable (though still not easy). In this case we simply need to look for a repeated state, as the program will diverge if and only if there exists some state that is repeated during execution. Furthermore, we cannot ignore the fixed-width semantics, as overflow and underflow can cause non-termination in programs that would otherwise terminates, an example is included in Figure 9. Another complication when considering this style of program is that of bit-level operations, such as left- or right-shift.

*Binary executables.* Until now we have discussed proving termination of programs at their source level, perhaps in C or Java. The difficulty with this strategy is that the compilers that then take these source programs and convert them into executable artifacts can introduce termination bugs that do not exist in the original source program. Several potential strategies could help mitigate this problem: 1) we might try to prove termination of the executable binaries instead of the source level programs, 2) we might try to equip the compiler with the ability to prove that the resulting binary program preserves termination, perhaps by first proving the termination of the source program and then finding a map from the binary to the source-level program and proving that the composition with the source-level termination argument forms a valid termination argument for the binary-level program.

*Non-linear systems.* Current termination provers largely ignore non-linear arithmetic. When non-linear updates to variables do occur (for example $\mathsf{x} := \mathsf{y} * \mathsf{z};$), current termination provers typically treat them as if they were the instruction $\mathsf{x} := \mathsf{input}();$. This modification is sound—meaning that when the termination prover returns the answer "terminating", we know that the proof is valid. Unfortunately, this method is not precise: the treatment of these commands can lead to the result "unknown" for programs that actually terminate. Termination provers are also typically unable to find or check non-linear termination arguments ($\mathsf{x}^2$, for example) when they are required. Some preliminary efforts in this direction have been made [1,6], but these techniques are weak. To improve the current power of termination provers, further developments in non-linear reasoning are required.

*Concurrency.* Concurrency adds an extra layer of difficultly when attempting to prove program termination. The problem here is that we must consider all possible interactions between concurrently executing threads. This is especially true for modern fine-grained concurrent algorithms, in which threads interact in subtle ways through dynamically allocated data structures. Rather than attempting to explicitly consider all possible interleavings of the threads (which does not scale to large programs) the usual method for proving concurrent programs correct is based on *rely-guarantee* or *assume-guarantee* style of reasoning, which considers every thread in isolation under assumptions on its environment and thus avoids reasoning about thread interactions directly. Much of the power of a rely-guarantee proof system such as [22, 28] comes from the cyclic proof rules, where we can assume a property of the second thread while proving property of the first thread, and then assume the recently proved property of the first thread when proving the assumed property of the second thread. This strategy can be extended to liveness properties using induction over time, e.g. [20, 27].

As an example, consider the two code fragments in Figure 10. Imagine that we are executing these two fragments concurrently. To prove the termination of the left thread we must prove that it does not get stuck waiting for the call to **lock**. To prove this we can assume that the other thread will always eventually release the lock—but to prove this of the code on the right we must assume the analogous property of the thread on the left, etc. In this case we can certainly just consider all possible interleavings of the threads, thus turning the concurrent program into a sequential model representing its executions, but this approach does not scale well to larger programs. The challenge is to develop automatic methods of finding non-circular rely-guarantee termination arguments. Recent steps [20] have developed heuristics that work for non-blocking algorithms, but more general techniques are still required.

```
1  while x > 0 do
2          x := x − 1;
3          lock(lck)
4          b := x;
5          unlock(lck)
6  done
```

```
1  while y > 0 do
2          lock(lck)
3          y := b;
5          unlock(lck)
6  done
```

**Figure 10: Example of multi-threaded terminating producer/consumer program. To prove that the thread on the left terminates we must assume that the thread on the right always calls unlock when needed. To prove that the thread on the right always calls unlock when needed, we must prove that the thread on the left always calls unlock when needed, etc.**

*Advanced programming features.* The industrial adoption of high-level programming features such as virtual functions, inheritance, higher-order functions, or closures make the task of proving industrial programs more of a challenge. With few exceptions (such as [18]), this area has not been well studied.

Untyped or dynamically typed programs also contribute difficulty when proving termination, as current approaches are based on statically discovering data-structure invariants and finding arithmetic measures in order to prove termination. Data in untyped programs is often encoded in strings, using pattern matching to marshal data in and out of strings. Termination proving tools for JavaScript would be especially welcome, given the havoc that non-terminating JavaScript causes daily for web browsers.

*Finding preconditions that guarantee termination.* In the case that a program does not guarantee termination from *all* initial configurations, we may want to automatically discover the conditions under which the program does guarantee termination. That is, when calling some function provided by a library: what are the conditions under which the code is guaranteed to return with a result? The challenge in this area is to find the right precondition: the empty precondition is correct but useless, whereas the weakest precondition for even very simple programs can often be expressed only in complex domains not supported by todays tools. Furthermore, they should be computed quickly (the weakest precondition expressible in the target logic may be too expensive to compute). Recent work [13, 33] has shown some preliminary progress in this direction.

*Liveness.* We have alluded to the connection between liveness properties and the program termination problem. Formally, liveness properties expressed in temporal logics can be converted into questions of *fair termination*—termination proving were certain non-terminating executions are deemed *unfair* via given fairness constraints, and thus ignored. Current tools, in fact, either perform this reduction, or simply require the user to express liveness constraints directly as the set of fairness constraints [12, 29]. Neither approach is optimal: the reduction from liveness to fairness is inefficient in the size of the conversion, and fairness constraints are difficult for humans to understand when used directly. An avenue for future work would be to directly prove liveness properties, perhaps as an adaption of existing termination proving techniques.

*Dynamic analysis and crash dumps for liveness bugs.* In this article we have focused only on *static*, or *compile-time*, proof techniques rather than techniques for diagnosing divergence during execution. Some effort has been placed into the area of automatically detecting deadlock during execution time. With new developments in the area of program termination proving we might find that automatic methods of discovering *livelock* could also now be possible. Temporary modifications to scheduling, or other techniques, might be also be employed to help programs not diverge even in cases where they do not guarantee termination or other liveness properties. Some preliminary work has begun to emerge in this area (see [23]) but more work is needed.

*Scalability, performance, and precision.* Scalability to large and complex programs is currently a problem for modern termination provers—today's techniques are known, at best, to scale to simple systems code of 30,000 lines of code. Another problem we face is one of precision. Some small programs currently cannot be proved terminating with existing tools. Turing's undecidability result, of course, states that this will always be true, but this does preclude us from improving precision for various classes of programs and concrete examples. The most famous example is that of the Collatz' problem, which amounts to proving the termination or non-termination of the program in Figure 11. Currently no proof of this program's termination behavior is known.

```
1  while x > 1 do
2          if x is divisible by 2 then
3              x := x/2;
4          else
5              x := 3x + 1;
6          fi
7  done
```

**Figure 11: Collatz program. We assume that x ranges over all natural numbers with arbitrary precision (that is, neither 64-bit vectors nor 32-bit vectors). A proof of this program's termination or non-termination is not known.**

## Conclusion

This article has surveyed recent advances in program termination proving techniques for sequential programs, and pointed towards on-going work and potential areas for future development. The hope of many tool builders in this area is that the current and future termination proving techniques will become generally available for developers wishing to directly prove termination or liveness. We also hope that termination-related applications—such as detecting livelock at runtime or Wang's tiling problem—will also benefit from these advances.

## Acknowledgments

The authors would like to thank Lucas Bourdeaux, Abigail See, Tim Harris, Ralf Herbrich, Peter O'Hearn, and Hongseok Yang for their reading of early drafts of this article and suggestions for improvement.

## 1. REFERENCES

[1] D. Babic, A. J. Hu, Z. Rakamaric, and B. Cook. Proving termination by divergence. In *SEFM*, 2007.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, 2006.

[3] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *POPL*, 2007.

[4] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, 2006.

[5] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, 2006.

[6] A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.

[7] A. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV*, 2005.

[8] A. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *ICALP*, 2005.

[9] A. Chawdhary, B. Cook, S. Gulwani, M. Sagiv, and H. Yang. Ranking abstractions. In *ESOP*, 2008.

[10] M. Codish, S. Genaim, M. Bruynooghe, J. Gallagher, and W. Vanhoof. One loop at a time. In *WST*, 2003.

[11] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *TACAS*, 2001.

[12] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Vardi. Proving that programs eventually do something good. In *POPL*, 2007.

[13] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *CAV*, 2008.

[14] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[15] N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Appl. Algebra Eng. Commun. Comput.*, 2001.

[16] J. Farkas. Über die Theorie der einfachen Ungleichungen. *Journal für die reine und angewandte Mathematik*, 1902.

[17] A. Geser. Relative termination. PhD dissertation, 1990.

[18] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In *RTA*, 2006.

[19] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *RTA*, 2004.

[20] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, 2009.

[21] A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. Proving non-termination. In *POPL*, 2008.

[22] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 1983.

[23] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[24] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL*, 2001.

[25] S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, 2007.

[26] P. Manolios and D. Vroon. Termination analysis with calling context graphs. In *CAV*, 2006.

[27] K. L. McMillan. Circular compositional reasoning about liveness. In *CHARME*, 1999.

[28] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 1981.

[29] A. Pnueli, A. Podelski, and A. Rybalchenko. Separating fairness and well-foundedness for the analysis of fair discrete systems. In *TACAS*, 2005.

[30] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.

[31] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, 2004.

[32] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, 2005.

[33] A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *CAV*, 2008.

[34] F. Ramsey. On a problem of formal logic. *London Math. Soc.*, 1930.

[35] G. Stix. Send in the Terminator. *Scientific American Magazine*, November 2006.

[36] C. Strachey. An impossible program. *Computer Journal*, 1965.

[37] A. Tiwari. Termination of linear programs. In *CAV*, 2004.

[38] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *London Mathematical Society*, 1936.

[39] A. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, 1949.

[40] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.

**Byron Cook** is a Senior Researcher at Microsoft's research laboratory at Cambridge University, and Professor of Computer Science at Queen Mary, University of London.
**Andreas Podelski** is Professor of Computer Science at the University of Freiburg.
**Andrey Rybalchenko** leads the Verification Systems group at the Max Planck Institute for Software Systems.