Proving program termination

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

Byron Cook

ABSTRACT

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

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 potentially 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 computational means to determine the validity of all statements. In hopes of either solving or showing Hilbert's challenge impossible, logicians began to search for possible instances of *undecidable* problems, Turing's proof [46][2] of the termination's undecidability is the most famous of those findings.

The termination problem is structured as an infinite set of queries: to solve the problem we would need to invent a method capabable 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 halting problem"* or *"but that's undecidable"* are often used to end discussions that could otherwise have led to viable partial solutions for real but technically undecidable problems.

While we of course 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 necissarily 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 potentially 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.

In recent years, powerful new termination tools have emerged that return "unknown" infrequently enough that they are useful in practice [43]. 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 drawn from the Windows operating system or NASA's Mars rover. Furthermore, entire families of industrially useful termination-like properties—called *liveness properties* [1]—such as "Every call to `lock` is eventually followed by a call to `unlock`" are now automatically provable using termination proving techniques [17]. 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 [29]. 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 undecidable problems. Problems such as Wang's tiling problem or Diophantine equation solving are reducible to termination—in fact their proofs of undecidability are via reductions to termination. Thus, advances in termination proving could be potentially adapted for other applications. 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 halting problem"*, perhaps in the future the developer will instead adapt techniques from within modern termination provers in order to develop a partial solution to

[1] In English: "decision problem"

[2] There is a minor controversy as to whether or not Turing proved the undecidability in [46]. 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 [44].

```
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 terminate for all possible user-supplied inputs? (Answer provided in a footnote below)

the problem of interest.

The purpose of this article is to familiarize the reader with the recent advances in termination proving, and catalog the underlying techniques for those interested in adapting known termination proving techniques to other related 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 optional sidebars are made available for readers with backgrounds in mathematical logic.

Modular termination arguments

Thirteen years after publishing his original undecidability result, Turing proposed the now classic method of proving program termination [47]. His solution divides the problem into two parts:

Termination argument search: Find a *termination argument* in the form of a function that maps every program state to a value in a mathematical structure called a well-order [14]. 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).

Validity 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 configuration s to t, then $f(s) > f(t)$.

The reader with a background in logic may be interested in the formal explaination 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 enumerates 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 essense Turing is proposing that we search for a map from the program we are interested in proving termination of into a program known to terminate such that all steps in one program have analogous steps in the other program. This map to a well-order is usually called a *measure* or a *ranking function* in the literature. After the publication of [47], for the next 40 years all known methods of proving termination were in essence minor variations on the same 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. Once a suitable ranking function has been found, checking validity (while itself undecidable in the presence of nested loops and complex program invariants) is in practice is fairly easy.

The key trend in termination proving that has led towards progress has been the move away from the search for monolithic ranking functions and towards a search for *modular termination arguments*, as spearheaded in [23, 26, 34, 37, 40]. The soundness of the approaches proposed in these papers is usually established via an application of Koening's lemma [33] or Ramsey's theorem [42]. This change in strategy has opened up numerous new algorithmic approaches for proving termination. Examples can be found in [4, 5, 11, 15, 17, 19, 34, 38].

The advantage to a modular style of termination argument is that they can be expressed in small and easy-to-understand pieces and they are usually easier to find, as each piece of the argument can be found in parallel or incrementally using various known methods. Unfortunately, when using modular termination arguments, a more difficult validity condition must be checked. This difficulty can be mitigated thanks to recent advances in assert proving tools (as discussed in a later section).

Example using a single 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 configuation 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 $2x + y$ is representing a proper 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 lines 4 through 9.

Automatically proving the validity of a monolithic termination argument like $2x + y$ is usually quite easy using tools

[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 we know that $>$ is a well-founded relation. Furthermore, we know that any map f into S gives rise to a well-founded relation, by lifting $>$ via f: $\{(s,t) \mid f(s) > f(t)\}$. Turing's method [47] of proving a program's transition relation R well-founded is to find a map, f, into a well-order defining a termination argument $T = \{(s,t) \mid f(s) > f(t)\}$. To prove the validity of T we must show $R \subseteq T$. We know that T is well-founded, and as every sub-relation of a well-founded relation is itself well-founded, we then know that R is well-founded.

In this article we are using the phrase *modular termination argument* to refer to a finite disjunction of well-founded relations $T_1 \cup T_2 \cup \ldots \cup T_n$, where usually each T_i will be constructed as above via some map into a well-order. To prove the validity of this style of argument we must show that

$$R^+ \subseteq T_1 \cup T_2 \cup \ldots \cup T_n$$

Note that the non-reflexive transitive closure (the $+$ in R^+) is important: it is not suffecient to show that $R \subseteq T_1 \cup T_2 \cup \ldots \cup T_n$, as the union of well-founded relations is not guarenteed to be well-founded. It is the transitive closure that makes checking the subset inclusion more difficult in practice. The proof of the soundness of above approach is based on Ramsey's theorem [42]. Numerous papers have reported on results similar to the above [23, 26, 34, 40].

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

such as SLAM [3], BLAST [31], or IMPACT [36]. 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 *lexiographic* ranking function (a ranking function into a more advanced well-order).

Example using a modular termination argument. Following the trend towards the use of modular 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

or

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

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* 2w − 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 advanced reader will notice the relationship between our modular termination argument and complex lexiographic 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 independant from one another.

The difficulty with modular 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 all possible loop unrollings and not just single passes through a loop. That is to say: the modular 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,

```
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?

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

Figure 4: Example program with assert statement.

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 tricks 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 modular 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 modular termination argument is not valid for the program in Figure 3.

Argument validity checking

While validity checking for modular arguments is more difficult than checking for monolithic arguments, we can adapt the problem such that recently developed tools for proving the validity of **assert** statements in programs such as IMPACT [36] can be applied[4].

An **assert** statement can be used in a program to check that a condition is true. For example, **assert**($y \geq 1$) would ensure that $y \geq 1$ after executing the command. In practice, programs will raise exceptions when conditions fail, but we can use an assert verifier to formally prove at compile time that the conditions passed to **assert** statements always evaluate to true. Most assert verifiers will, for example, be able to prove that the **assert** statement at line 3 in Figure 4 will never fail. Note that assert verification is itself an undecidable problem, although it is technically in an easier class of difficulty than termination[5].

[4] We are using the term *assert statement checking* synonymously with safety checking, safety proving, invariance checking , invariance proving, etc.

```
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 assert statement.

The reason that assert-checking is so important to termination is that the validity of modular termination arguments can be encoded as an **assert** 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

> There *does not exist* a state such that another state is reachable via a finite unrolling of the loop such that T_1 doesn't hold between the pre-state and post-state, nor does T_2 hold between the pre-state and post-state, etc.

That is, we introduce new variables into the program to remember a previous state before the unrolling of the loop and then use an **assert** to check that the termination argument always holds between the current state and the remembered state, if the assert prover can prove that the assert cannot fail, it has proved the validity of the termination argument. We can use encoding tricks such that the assert verifier must 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 [19]. The new code (introduced as a part of the encoding) is given in

[5] Assert verification for infinite-state systems is undecidable but co-NP, whereas termination is not co-NP.

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 to record the pre-state. Note that the assert 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 the pre-state state has been recorded, from this point out the termination argument is checked using the pre-state and post-state. In this case the **assert** can fail, meaning that the termination argument is invalid.

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, assert verifiers must 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 assert verifiers are designed to produce a path to a bug in the case that the assert statement cannot be proved. For example, a path to the assert failure found by the assert prover IMPACT [36] 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's initial state to the first of the two failing states. The second part of the path $4 \to 5 \to \ldots 5 \to 6$ represents how we reached the second failing state from the first. That is: this is the unrolling found that demonstrates that the assert 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 **assert** cannot fail. The fact that it cannot fail can be proved by a number of assert verification tools, including IMPACT.

```
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 assert 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 assert. We now discuss known methods for finding termination arguments.

Monolithic rank function synthesis. In some cases simple monolithic ranking functions for example programs can be automatically found. For example, if we consider unnested loops with conditional checks and assignment statements using only linear arithmetic over the real numbers or rationals, the search for ranking functions expressed in linear arithmetic is decidable [39]. The most popular approach for finding this class of ranking function uses a result from Farkas [25] together with tools for solving linear constraint systems (such as Z3 [22] or Yices [24]). See [16] or [39] for examples of tools using Farkas' lemma. Approaches for finding more complex classes of ranking functions for restricted systems have also been proposed—see [2, 8, 9, 10, 12, 11, 8, 13, 28, 45]. These tools are sometimes applied directly to programs, but more frequently they are used internally within termination proving tools on representations of parts of the software during the search for modular termination arguments.

Termination analysis. Numerous approaches have been developed for finding modular termination arguments in which—in effect—the validity condition for modular termination arguments is almost guaranteed to hold by construction. In some cases—[4] for example—to prove termination we need only check that the argument indeed represents a set of measures. In other cases, such as [34] or [38], the tool makes a one-time guess as to the termination argument and then checks it using techniques drawn from abstract interpretation [21].

Consider the modified program in Figure 7. The termination strategy described in [4] essentially builds a program like this and then applies a custom program analysis to find the following candidate termination argument:

$$(\mathsf{copied} \neq 1) \ \textbf{or}$$

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

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

for the program at line 4—meaning that we could pass this complex expression to **assert** at line 4 in Figure 7 and know that the assert 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 oldx $\geq$ x +

```
copied := 0;
x := input();
y := input();
while x > 0 and y > 0 do
    if copied = 1 then
        skip;
    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 7: Program prepared for abstract interpretation

1 **and** 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 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. Another advantage to these approaches is that they will not diverge themselves, they are guarenteed to either produce the answer "terminates" or "unknown".

Finding arguments by refinement. Another method for discovering a termination argument is to follow the approach of [19] or [15] and search for counterexamples to (possibly invalid) termination arguments and then refine them based on new ranking functions found via the counterexamples.

Recall Figure 5, which encoded the invalid termination argument for the program in Figure 1, and the path leading to the failure of the assert: $1 \to 2 \to 3 \to 4 \to 5 \to 7 \to 8 \to 9 \to 10 \to 11 \to 12 \to 16 \to 17 \to 4 \to 5 \to 6$. 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 $4 \to 5 \to \ldots 5 \to 6$ 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 ([30], 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 **assert** never fails, thus proving the termination of the program in Figure 1.

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 the 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 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, 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 [35, 7, 5]. 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 [49] 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 [35] 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

Note for the reader with a background in logic

In some precise detail, here a brief summary of the known implementation strategies based on modular termination arguments:

Variance analysis [4]: As described in some detail in this article, the approach from [4] uses program transformations and abstract interpretion 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.

Size-change [34]: This technique abstracts R, determining a computable overapproximation to R^+. It then guesses at a termination argument using each variable in the program: T_1 equals the set of states ordered on variable x_1, T_2 equals the set of states ordered on variable x_2, etc. Each T_i is thus well-founded by construction. The size-change approach has been implemented in numerous tools and extended in various ways.

Induction [15]: The strategy here is to find an *inductive* termination argument, $T = T_1 \cup T_2 \cup \ldots \cup T_n$ such that $R \subseteq T$ and $T; R \subseteq T$, thus proving $R^+ \subseteq T$. The advantage to this approach is we do not need to implement assert-checking strategies which support transitive closure. The disadvantage is that finding inductivly valid arguments is more challenging than simply finding valid arguments

Refinement [19]: In this approach 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. This method is not guarenteed to return an answer, but answers can be verifiably correct.

```
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.

```
while (x != NULL && y<1073741824) {
    if (input() == 1) {
        x = x->next;
    } else {
        y = y << 1;
    }
}
```

Figure 10: Example program using both bit-vectors and unbounded heap. Existing techniques for proving termination of programs with heap, and programs with bit-vectors do not mix well in current frameworks.

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)[6]. 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 [6]. 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 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.

When programs mix fixed-width numbers with the heap or unbounded numbers (perhaps introduced for reasoning about the heap), more difficulties arise. Good techniques that support such mixed programs are currently not known, and should be developed in the future. Consider, for example, the code in Figure 10, where x is used as a pointer into the heap, but y ranges only over bit-vectors. In this case we need to find a termination argument that fits into the modular termination argument framework, but is accurate for programs with fixed-width integers.

Binary executables. Until now we have discussed proving termination of programs at their source level, perhaps in

[6] As a consequence of [20], we also know this result holds even in the precense of recursive procedures.

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, perhaps using recent advances in binary analysis; 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 (x^2, for example) when they are required. Some preliminary efforts in this direction have been made [2, 8], 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* reasoning, which considers every thread in isolation under assumptions on its environment and thus avoids reasoning about thread interactions directly. Much of power of a rely-guarantee proof system such as [48] 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 works for assert-verification, but not not termination (it is unsound for termination and liveness).

As an example, consider the two code fragments in Figure 11. 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 [29] have developed heuristics that work for non-blocking algorithms, but more general techniques are still required.

```
while x > 0 do
      x := x − 1;
      lock(lck)
      b := x;
      unlock(lck)
done
```

```
while y > 0 do
      lock(lck)
      y := b;
      unlock(lck)
done
```

Figure 11: 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 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 [27]), 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 programs is often encoded in strings, using pattern matching to marshal data in and out of strings. Termination proving tools for 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 [18, 41] has shown some preliminary progress in this direction.

Liveness. We have alluded to the connection between liveness properties [1] and the program termination problem. Formally, liveness properties expressed in temporal logics such as LTL 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,

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

Figure 12: Collatz program. We assume that x ranges over all natural numbers with arbitrary precision (that is, not 64-bit vectors nor 32-bit vectors). A proof of this program's termination or non-termination is not known.

or simply require the user to express liveness constraints directly as the set of fairness constraints [17]. 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 [32]) 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 12. Currently no proof of this program's termination behavior is 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

todo

1. REFERENCES

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[2] D. Babic, A. J. Hu, Z. Rakamaric, and B. Cook. Proving termination by divergence. In *SEFM*, pages 93–102, 2007.

[3] 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: European Systems Conference*, pages 73–85, 2006.

[4] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *POPL: Programming Language Design and Implementation*, 2007.

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

[6] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *FMICS: Formal Methods for Industrial Critical Systems*, volume 66(2) of *ENTCS*, 2002.

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

[8] A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*, pages 113–129. Springer, 2005.

[9] A. R. Bradley and Z. Manna. Termination and invariance analysis of loops. In *ATVA*, page 2, 2005.

[10] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV*, pages 491–504, 2005.

[11] A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *ICALP*, pages 1349–1361, 2005.

[12] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination analysis of integer linear loops. In *CONCUR*, pages 488–502, 2005.

[13] M. Braverman. Termination of integer linear programs. In *CAV*, pages 372–385, 2006.

[14] G. Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers*. Dover, 1955.

[15] A. Chawdhary, B. Cook, S. Gulwani, M. Sagiv, and H. Yang. Ranking abstractions. In *ESOP: European Symposium on Programming*, 2008.

[16] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 67–81. Springer, 2001.

[17] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Vardi. Proving that programs eventually do

something good. In *POPL: Programming Language Design and Implementation*, 2007.

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

[19] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI: Programming Language Design and Implementation*, 2006.

[20] B. Cook, A. Podelsksi, and A. Rybalchenko. CFL-termination. *Formal Methods in System Design*, To appear 2009.

[21] P. Cousto and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL: Principles of Programming Languages*, 1977.

[22] L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[23] N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Appl. Algebra Eng. Commun. Comput.*, 12(1/2):117–156, 2001.

[24] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T)*. In *CAV*, 2006.

[25] J. Farkas. Uber die theorie der einfachen ungleichungen. *Journal fur die Reine und Angewandte Mathematik*, 124:1–27, 1902.

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

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

[28] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *RTA: Rewriting Techniques and Applications*, volume 3091 of *LNCS*, pages 210–220. Springer, 2004.

[29] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL: Programming Language Design and Implementation*, 2009.

[30] A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. Proving non-termination. In *POPL: Principles of Programming Languages*, 2008.

[31] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL: Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

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

[33] D. Koenig. Sur les correspondances multivoques des ensembles. *Fundamenta Mathematicae*, 8:114–134, 1926.

[34] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL: Principles of Programming Languages*, 2001.

[35] S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS: Static Analysis Symposium*, 2007.

[36] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[37] J. Misra. On the union of well-founded relations: An application of koenig's lemma. Unpublished note, 1996.

[38] D. V. Panagiotis Manolios. Termination analysis with calling context graphs. In *CAV: Conference on Computer Aided Verification*, 2006.

[39] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI: Verification, Model Checking, and Abstract Interpretation*, pages 239–251, 2004.

[40] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS: Logic in Computer Science*, 2004.

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

[42] F. Ramsey. On a problem of formal logic. *London Math. Soc.*, 30:264–286, 1930.

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

[44] C. Strachey. An impossible program. *Computer Journal*, 7(4):313, 1965.

[45] A. Tiwari. Termination of linear programs. In *CAV*, pages 70–82, 2004.

[46] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *London Mathematical Society*, 42(2):230–265, 1936.

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

[48] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

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

Byron Cook is a researcher at Microsoft's research laboratory at Cambridge University, and Professor of Computer Science at Queen Mary, University of London.