Under consideration for publication in Formal Aspects of Computing

Proving Termination of Nonlinear Command Sequences

Domagoj Babić[1], Byron Cook[2], Alan J. Hu[3], Zvonimir Rakamarić[4]

[1] Computer Science Division, University of California, Berkeley
[2] Microsoft Research and University College London
[3] Department of Computer Science, University of British Columbia
[4] School of Computing, University of Utah

Abstract. We describe a simple and efficient algorithm for proving the termination of a class of loops with nonlinear assignments to variables. The method is based on divergence testing for each variable in the cone-of-influence of the loop's condition. The analysis allows us to automatically prove the termination of loops that cannot be handled using previous techniques. We also describe a method for integrating our nonlinear termination proving technique into a larger termination proving framework that depends on linear reasoning.

Keywords: program termination; nonlinear assignments; divergence testing; well-foundedness provers

1. Introduction

Throughout the history of program verification (e.g., [30, 16, 18]) the task of proving a program correct has been decomposed into the tasks of proving correctness *if* the program terminates, and separately proving termination. Proving termination is clearly undecidable, but thanks to considerable research progress, a variety of practical techniques can now automatically prove termination of many loops that occur in practice. These automatic techniques, however, are temperamental. While most provers can, for example, prove the termination of

```
if (x > 3) {
    while (x < y) {
        x = x + x;
    }
}
```

until now, no automatic tool could prove the termination of a nonlinear example like

Correspondence and offprint requests to: Domagoj Babić, Computer Science Division, University of California, Berkeley.
E-mail: babic.domagoj@gmail.com

```
if (x > 3) {
    while (x < y) {
        x = pow(x,3) - 2*pow(x,2) - x + 2;
    }
}
```

This paper outlines a proof procedure for cases of this sort. Our intention for this proposed procedure is for it to be combined with existing termination analysis techniques—thus leading to more robust termination provers.

The failure of previous automated tools to handle the above example is somewhat surprising, as a human can quickly see that for large enough x, repeated iteration of the loop will make x diverge to infinity, thus guaranteeing loop termination. Exactly how large an initial value of x is large enough, though, requires a bit of thought and some algebra. But if we have a method to quickly prove that loop variables diverge, we can harness that fact to help prove termination.

Our proposed technique is based on such divergence testing: the transition system of each program variable is independently examined for divergence to plus- or minus-infinity. The approach is limited to loops containing only polynomial update expressions with finite degree, allowing highly efficient computation of certain regions that guarantee divergence. Like any automated termination prover, our technique cannot handle all loops. However, it is fast, it is sound, and it can prove termination in cases that previously could not be handled or could be handled only by a much more expensive analysis. Our hope is that, in practice, this restricted analysis (and some extensions) will handle the termination of the majority of loops in which a nonlinear analysis is required. In our investigations, we have found that this simple type of nonlinear loop appears in industrial numerical computations and nonlinear digital filters. Such loops can also be found in the code that handles multi-dimensional matrices and nonlinear time-outs.

2. Termination by Divergence

This section starts with some basic definitions. Then, we proceed with the definition and examples of *regions of guaranteed divergence*, a key concept in the paper. Section 2.3 explains how we test for divergence. Section 2.4 presents how we use the divergence information to prove termination, and gives the overall algorithm.

2.1. Basic Definitions

Two types of variables are differentiated by our analysis — symbolic constants and induction variables. The symbolic constants are not modified in the loop body and the set of symbolic constants will be denoted as $\mathcal{S}$. Individual symbolic constants will be represented with capital letters X, Y, Z and are assumed to be bounded on both sides (e.g. $-32 \leq X \leq 32$). The set of variables $\mathcal{V}$ that are modified within the loop body will be called the set of induction variables. The induction variables will be denoted by lower-case letters x, y, z. Both types of variables are considered to be rationals. We will assume that the loop condition is a conjunction of a finite number of (in)equality relations of the form $\phi \bowtie 0$, where $\bowtie \in \{<, >, \leq, \geq, =\}$, and the constraints on ϕ are given below. Each induction variable x is updated within the loop. The update expression is of the form $x = f(x) + X$, where $f(x)$ is a univariate finite degree polynomial with constant rational coefficients and X is a symbolic constant that represents the cumulative effect of all the symbolic constants in the expression. The polynomial $f(x)$ will be called an update function. The initial value of x, before entering the loop, will be denoted as x_0. Let x^+ stands for the result (e.g., $\pm\infty$, some constant, a cycle, etc.) obtained by applying f infinitely many times to x (informally, $\cdots f \circ f \circ f(x)$). Let $f^2 = f \circ f$ be a composition of f with itself. The value obtained by applying f^2 to x infinitely many times will be denoted as x^{++}. The set of univariate polynomials with constant coefficients over $\mathbb{Q}$ (the set of rationals) will be denoted as $\mathbb{Q}[x]$, while the set of multivariate polynomials over $\mathbb{Q}$ as $\mathbb{Q}[\mathcal{V}]$. Let support of an expression, denoted $\mathsf{supp}\,(\psi)$, be the finite set of names in a logical expression ψ, e.g. $\mathsf{supp}\,(x \cdot y + z > 0) = \{x, y, z\}$.

The algorithm described in this paper is designed such that it can be used as a sub-procedure within a larger termination proving framework, such as Berdine et al. [2]'s or that of Cook et al. [11]. These termination provers convert the termination problem for a program with an arbitrary control-flow graph into a sequence of well-foundedness queries for relations composed as the conjunction of inequalities, or *simple loops*. These tools then find a termination argument for the whole program based on the arguments drawn from these simple loops. Numerous techniques can be used to prove the well-foundedness of these simple loops. Our goal in this paper is to provide support for nonlinear reasoning when trying to prove the well-foundedness of these simple loops. Our technique handles simple loops with the following properties:

1. A set of bound constraints $\mathcal{B} \subseteq \{v \bowtie q \mid v \in \mathcal{V} \cup \mathcal{S} \wedge q \in \mathbb{Q}\}$ over symbolic constants and induction variables represents the initial conditions. Symbolic constants must be from a closed finite interval. These restrictions make it easy to check for intersection between the initial conditions and a region that guarantees loop divergence, computed in Section 2.2. Relaxing these restrictions is possible if using a more expressive representation of the set of possible initial states, e.g., if we allowed $\neq$, the algorithm could track a finite disjunction of convex regions to represent the initial conditions.
2. There are no data or control dependencies between induction variables, *i.e.*, the dependency relation is an antichain. In other words, all update statements within the loop body must be executable in parallel. (Symbolic execution and other analysis/optimization techniques can, of course, be used to eliminate false dependencies between induction variables.) This restriction allows our technique to analyze the divergence each induction variable separately, greatly simplifying the analysis.
3. The loop condition is a relation of the form $\phi \bowtie 0$ such that each ϕ is a finite degree multivariate polynomial from $\mathbb{Q}[\mathcal{V} \cup \mathcal{S}]$. Only the constant term (i.e. the one associated with the exponent zero) is allowed to be a symbolic constant (or a linear function of symbolic constants). This restriction is for expository convenience. In Section 2.4, we present an extremely simple algorithm for safely determining termination under this restriction. If more sophisticated computer algebra software is available, then this restriction can be relaxed (as we will see in our experimental results). Also, in practice, loop conditions are often conjunctions of multiple relations; if any conjunct can be proven to be eventually false, the loop must terminate. (Disjunction is more difficult, which is why we disallow $\neq$ in loop tests.)
4. Update functions f_i are univariate polynomials of finite degree from $\mathbb{Q}[x_i]$. This restriction is needed because our technique uses algebraic properties of polynomials. If some numerical imprecision is acceptable, we could generalize our method to non-polynomial update functions by using numerical root-finding techniques in Section 2.2.
5. Given a linear function $g : \mathcal{S}^n \to \mathbb{Q}$ of symbolic and numeric constants, each assignment can have the form $x_i \leftarrow f_i(x_i) + g_i(Y_1, Y_2, \dots)$. The second term can be represented with a single symbolic constant X_i because symbolic constants do not change within the loop. It is assumed that the symbolic constant X_i can take any value between the worst case min and max values of the replaced symbolic constant expression $g_i(Y_1, Y_2, \dots)$. This restriction allows efficient computation of a safe termination test for all possible values of the symbolic constant.

If a loop in the sequence does not fit into this criteria, then other techniques must be used in order to establish well-foundedness.

A loop with such properties will be called a NAW (Nonlinear Antichain While) loop. The properties of the NAW loop enable us to analyze the limit behavior of each induction variable independently. Given the limit behavior of each induction variable $v \in \mathsf{supp}\,(\phi)$, we try to prove that the left-hand side value of the loop condition will eventually cross the given bound, terminating the loop. Obviously, it suffices to consider only the variables that are in the cone-of-influence of the loop condition. Standard slicing techniques [28] can be used to remove the unnecessary code. We will assume that the NAW loops have been preprocessed so as to eliminate assignments to variables that are not in the cone-of-influence of the loop condition.

Several examples are given in Figure 1. The generic form of the NAW loop 1(a) starts with range constraints for induction variables and symbolic constants. The range constraints can be computed by range analysis, which is a simple forward dataflow analysis [13]. The left column (Figures 1(b),1(d),1(f)) contains examples of loops that are not NAW loops. The loop condition of 1(b) does not fit the NAW loop requirements because the symbolic constant Y is from a half-open interval. The other two loops in the same column are not NAW loops either, as the symbolic constant B is used in a non-constant term (Figure 1(d)) and the induction variable x (Figure 1(f)) depends on the induction variable n. The right column contains three examples of NAW loops. Examples 1(c) and 1(e) are adapted from previous work [12].

The overall divergence analysis algorithm takes a loop condition ϕ, a set of bound constraints $\mathcal{B}$, and a set of update functions f_i, and indicates either that it cannot prove termination, or that it can prove termination, with a mapping from induction variables to limits that demonstrates that the loop condition

```
Range constraints
while φ ⋈ 0, ⋈ ∈ {<, >, ≤, ≥, =} do
    x_1 ← f_1(x_1) + X_1;
    x_2 ← f_2(x_2) + X_2;
    x_3 ← f_3(x_3) + X_3;
    ...
    x_n ← f_n(x_n) + X_n;
end while
```

(a) Generic form of NAW loops

```
x < −17; −32 < Y;
while x ≠ Y do
    x ← 3x − 1
end while
```

(b) Non-NAW loop – Symbolic constant Y is only single-side bounded

```
e ← 0.00001; a ← 1 − e; x ← 0.5;
while e ≤ x ∧ x ≤ 1 do
    x ← −a · x^2 + a · x;
end while
```

(c) Terminating NAW loop

```
x ← 5; −10 < B < 10;
while x < 1000 do
    x ← x^2 − B · x − 7
end while
```

(d) Non-NAW loop – Symbolic constant B is used in a non-constant term

```
q ← 0; r ≥ 0; 1 ≤ Y ≤ 65535;
while y ≤ r do
    r ← r − Y
    q ← q + 1
end while
```

(e) Terminating NAW loop

```
n ← 0; x ← 1; 1 < N < 1000;
while x ≤ N do
    n ← n + 1
    x ← n · x
end while
```

(f) Non-NAW loop – Dependency between induction variables

```
x < −2;
while x < 10 do
    x ← x^3 − 2x^2 − x + 2;
end while
```

(g) Non-terminating NAW loop

Fig. 1. General NAW-form together with some examples

must eventually become false. The algorithm has three main steps. The first task is a fast analysis to find regions where iterated application of the update functions f_i can easily be proven to diverge. Once some "regions of guaranteed divergence" have been found, the next task, divergence analysis, tries to find the limit behavior of each induction variable. Finally, termination analysis conservatively uses the limit behaviors to prove that the loop condition must be violated eventually, which proves termination.

2.2. Regions of Guaranteed Divergence

Definition 1 (Stable Point). A value x is a stable point if $x = f(x)$. If x_0 is a stable point, then $x = x_0$ is a loop invariant.

Definition 2 (Region of Guaranteed Divergence (RGD)). A region of guaranteed divergence of an update function f is a (possibly unbounded) interval such that iterated application of f to any element from the interval diverges.

A brief synopsis of our approach is as follows. We use a fast polynomial factoring algorithm [19, 21] to compute the stable points of the update function. Let r_{min} and r_{max} denote the minimal and maximal stable points. Then, depending on characteristics of the update function, we conclude that $(-\infty, r_{\mathsf{min}})$ and/or

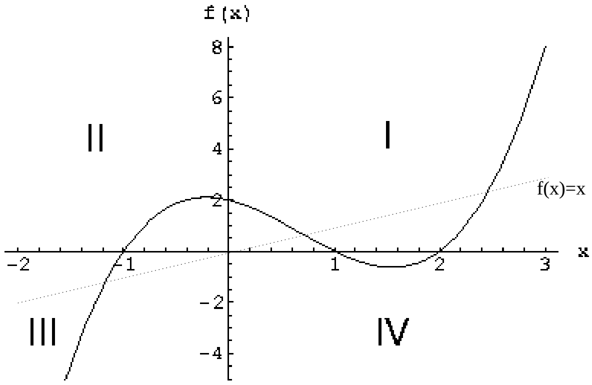

Fig. 2. $f(x) = x^3 - 2x^2 - x + 2$. Quadrants are denoted by the Roman numerals I,II,...

$(r_{\mathsf{max}}, \infty)$ (subject to some side conditions) are RGDs.[1] We can strengthen the analysis by applying the update function a finite number of times (currently exactly once in certain cases) before checking to see if the induction variable has entered an RGD. We now present the details.

The shape of the update function determines how the RGD is computed. Update functions are univariate polynomials that can be drawn in the two-dimensional Cartesian coordinate system, which has four quadrants, numbered counter-clockwise from the top right (Figure 2). In the limit, finite degree polynomials diverge in two out of four quadrants. The pair of quadrants in which the function diverges determines the shape of the curve. Our analysis case splits on the four possible shapes:

Case 1: Divergence in Quadrants I and III The update function must have odd degree and positive leading coefficient. This is the most straightforward case. If the degree is greater than 1, then $(-\infty, r_{\mathsf{min}})$ and $(r_{\mathsf{max}}, \infty)$ are both RGDs, because the update function must have a slope greater than 1 in those regions (Figure 2). If the degree of the update function is 1 (linear), then the key is the leading coefficient (the slope) of the update function. If the slope is strictly greater than 1, we have the same RGDs as above (with $r_{\mathsf{min}} = r_{\mathsf{max}}$). If the slope is strictly less than 1 (and it must be greater or equal to 0), then there is no RGD — the iteration converges to the stable point. If the slope is exactly 1, then all of $(-\infty, \infty)$ is an RGD (except in the degenerate case when the update function is the identity function).

Case 2: Divergence in Quadrants I and II The update function must have even degree and positive leading coefficient. If there are no stable points, all of $(-\infty, \infty)$ is an RGD. Otherwise, as in Case 1, $(r_{\mathsf{max}}, \infty)$ is an RGD. Unlike Case 1, we cannot say anything about $(-\infty, r_{\mathsf{min}})$. However, because f diverges in quadrant II, if we know there exist sufficiently small values x_0, such that $f(x_0) > r_{\mathsf{max}}$, and hence in an RGD. In particular, let r'_{min} be the smallest solution of $f(x) = r_{\mathsf{max}}$. Then for all $x < r'_{\mathsf{min}}$, we know that $f(x) > r_{\mathsf{max}}$, so $(-\infty, r'_{\mathsf{min}})$ is also an RGD.

Case 3: Divergence in Quadrants III and IV The update function must have even degree and negative leading coefficient. This is the dual of Case 2. If there are no stable points, $(-\infty, \infty)$ is an RGD. Otherwise, the region $(-\infty, r_{\mathsf{min}})$ is an RGD, as is the region $(r'_{\mathsf{max}}, \infty)$, where r'_{max} is the largest solution of $f(x) = r_{\mathsf{min}}$.

Case 4: Divergence in Quadrants II and IV The update function must have odd degree and negative leading coefficient. This is the most complex case because when iterated application of the update function diverges, it will alternate sign on each iteration. Happily, in this case, $f^2 = f \circ f$ is an update function that follows Case 1. So, we compute the stable points of $x = f^2(x)$ instead, and analyze f^2 as in Case 1.

In this paper, we prove termination by divergence of induction variables. In general, the termination analysis can be much smarter. For example, if an induction variable always cycles or converges to a constant, the termination analysis might exploit that knowledge for the termination proof.

[1] In floating-point arithmetic, the RGDs must be safely under-approximated. This is done by rounding the computed stable points to reduce the size of the RGD, i.e., rounding r_{max} toward ∞, and r_{min} toward $-\infty$.

Equations	Roots	Quadrants
$x = x^3 - 2x^2 - x + 2$	$-1.170086\ldots$, $0.688892\ldots$, $2.481194\ldots$	I,III

Table 1. Real roots of the update function $f(x) = x$

Example: The polynomial plotted in Figure 2 will serve as our working example. Since the update function has degree 3 and positive leading coefficient, it diverges in the quadrants I and III. Therefore, we compute the stable points from $x = f(x)$. By solving $x = x^3 - 2x^2 - x + 2$, we get $r_{\mathsf{min}} = -1.1700086\ldots$ and $r_{\mathsf{max}} = 2.481194\ldots$, which we can round conservatively (for brevity here, to two decimal places) to get RGDs of $(-\infty, -1.18)$, where iterated function application diverges towards $-\infty$, and $(2.49, \infty)$, where iteration diverges towards ∞. If range analysis tells us that $x_0 > 2.49$ in the loop below:

```
while (x < 10) {
    x = pow(x,3) - 2*pow(x,2) - x + 2;
}
```

then the induction variable x diverges towards $+\infty$ and therefore will eventually overshoot the given bound in the loop condition. Alternatively, if $x_0 < -1.18$, then x diverges towards $-\infty$ and the loop certainly doesn't terminate. For $x_0 \in (-1.18, 2.49)$, our analysis makes no attempt to determine the outcome. Indeed, if $x_0 = 0$, x will cycle through the set $\{0, 2\}$ taking values $\{0, 2, 0, 2, 0, 2, \cdots\}$.

An important question is what happens when the constant term[2] in the polynomial $f(x)$ is a symbolic constant. Let's assume a symbolic constant $0 \leq Z \leq 30$, the previously given loop condition, and the assignment `x = pow(x,3) - 2*pow(x,2) - x + 2 + Z`. The entire curve in Figure 2 can shift upwards by Z. The divergence analysis is the same, but the shift caused by symbolic constants must be taken into account.

Definition 3 (Safe Region of Guaranteed Divergence (SRGD)). A safe region of guaranteed divergence of an update function f with symbolic constant Z from a closed finite interval is a (possibly unbounded) interval such that iterated application of $f + Z$ to any element from the interval diverges for all possible values of Z.

Because this definition quantifies over all possible values of Z, it takes into account the effect of worst-case shifts of the polynomial caused by the symbolic constant. Fortunately, because we restrict symbolic constants to closed finite intervals, it is easy to compute SRGDs:

Lemma 1. Let f be any legal update function, except that f must not be of the form $f(x) = x + c$ for some constant c. Let Z be a symbolic constant from a closed finite interval $[Z_{\min}, Z_{\max}]$. The intersection of the corresponding RGDs of $f + Z_{\min}$ and of $f + Z_{\max}$ are SRGDs.

Proof. The result follows immediately from the fact that the values of largest and smallest roots change monotonically with Z, so we need compute RGDs for only the endpoint values of the interval bounding Z. For example, if f falls under Case 1 (odd degree with positive leading coefficient), then increasing Z will decrease the value of r_{max}, the largest root of $f(x) + Z = x$. The other cases are similar. (The only exception to this reasoning is when $f(x) = x + c$ for some constant c. In this case, if Z can be equal to $-c$, then all values are roots and there is no RGD or SRGD; otherwise, there are no roots, and the entire interval $(-\infty, +\infty)$ is an SRGD.) $\square$

Going back to our working example, Z can shift the entire curve by $Z \in [0, 30]$. If the shift is equal to 30, by solving $x^3 - 2x^2 - 2x + 32 = 0$, we get a single real root $-2.7990\ldots$, yielding RGDs $(-\infty, -2.80)$ and $(-2.79, \infty)$. From the previous analysis, we know that $(-\infty, -1.18)$ and $(2.49, \infty)$ were RGDs when $Z = 0$. Thus, the safe RGDs are the intersection of these two shifts, yielding $(-\infty, -2.80)$ and $(2.49, \infty)$ as safe RGDs.

Computing safe RGDs doubles the amount of work. However, due to the efficiency of our algorithm, the computational overhead is negligible compared to the benefits of being able to handle some degree of non-determinism. In order to keep the exposition simple, in the rest of the paper, we will not explicitly discuss the safe regions. The algorithm and the proofs can be trivially extended to handle them.

[2] By definition of NAW loops, only the constant term of the update function is allowed to be a symbolic constant.

2.3. Divergence Analysis

Once we have RGDs, if any, for all update functions, we can test whether we can easily prove divergence for each induction variable. The basic computation is whether the range bound on the initial value of some induction variable x_0 is wholly contained in an RGD. If so, the induction variable must diverge eventually.

In a general, more elaborate computation of RGDs, in which we apply the update function a bounded number of times in order to compute more RGDs, we would tag each RGD with whether it leads to divergence to $+\infty$, $-\infty$, or alternating $\pm\infty$. In our current, fast-and-lightweight implementation, however, we determine the direction of divergence easily by checking, for an arbitrarily chosen point x in the RGD, whether $f(x) > x$, which gives divergence to $+\infty$, or $f(x) < x$, which gives divergence to $-\infty$:

Case 1 (Quadrants I,III) We label an induction variable as diverging if the bound constraints of the initial value of the induction variable are contained in an RGD. We determine the direction of divergence by checking whether $f(x) < x$ or $f(x) > x$ for an arbitrarily chosen point x in the RGD.

Case 2 (Quadrants I,II) We label an induction variable as diverging if the bound constraints of the initial value of the induction variable are contained in an RGD. Divergence is always to $+\infty$.

Case 3 (Quadrants III,IV) We label an induction variable as diverging if the bound constraints of the initial value of the induction variable are contained in an RGD. Divergence is always to $-\infty$.

Case 4 (Quadrants II,IV) We label an induction variable as diverging if the bound constraints of the initial value of the induction variable are contained in an RGD. Divergence is always to alternately $\pm\infty$.

2.4. Termination Analysis

Once we have determined the divergence behavior of the induction variables, the remaining task is to compute a limit of the multivariate polynomial in the loop condition. Note that we cannot compute this limit by haphazardly applying the limits computed for each variable independently, or by focusing only on the highest-degree terms in the polynomial. For example, consider the loop in which one induction variable (y) changes much faster than the others:

```
while (y - pow(x,2) > 100) {
    y = pow(y,10) + 1000;
    x--;
}
```

Obviously, $x \to -\infty$ and $y \to +\infty$. Considering only the highest-degree term $-x^2$ in the loop condition would incorrectly compute that the left side of the condition diverges towards $-\infty$ and that the loop terminates.

However, given the restrictions on the loop condition we imposed in Section 2.1, it is easy to compute a safe, conservative evaluation of the loop condition given the information we have. Essentially, the computation is a trivial abstract interpretation [13] of the loop condition, where the induction variables have values in the abstract domain $\{-\infty, +\infty, \pm\infty, \perp\}$, depending on the result of the divergence analysis (where the abstract value $\perp$ denotes no information about the divergence behavior; and $\pm\infty$ denotes that the induction variable diverges, with alternating sign on each iteration). For example, for the preceding loop, we would assign $x = -\infty$ and $y = +\infty$ based on the divergence analysis in Section 2.3. Then, we would evaluate $x^2 = +\infty$, and $y - x^2 = +\infty - (+\infty) = \perp$. Since $\perp > 100$ is not provably false, we do not claim termination. By expanding the abstract domain (e.g., to indicate different growth rates, convergence to constants, etc.), we could improve the accuracy of the termination analysis.

Conveniently, in our prototype implementation, none of this was needed. Symbolic algebra packages are readily available, and computing the limit behavior of a multivariate polynomial is a commonly supported operation. Our prototype implementation used Maple [24], and we simply fed the result of our divergence analysis to the package and asked it to try to evaluate the limit behavior of the loop condition. The additional power of Maple also allowed us to handle an example (Figure 3(c)) whose termination condition did not obey the restrictions in Section 2.1. (This example could also have been handled correctly by the abstract interpretation described above. It is difficult to characterize precisely the limitations of Maple or other computer algebra packages, so in this paper, we have kept our presentation to a more restricted formalization.)

2.5. Algorithm Summary, Soundness, and Efficiency

The outline of the overall algorithm is given in Algorithm 1.

Algorithm 1 ZigZag

Find the RGDs for each induction variable ▷ Section 2.2
For each induction variable determine its divergence behavior ▷ Section 2.3
if the loop condition is

$\phi < 0$ and ϕ evaluates to $+\infty$, or ▷ "evaluates" as in Section 2.4
$\phi \leq 0$ and ϕ evaluates to $+\infty$, or
$\phi \geq 0$ and ϕ evaluates to $-\infty$, or
$\phi > 0$ and ϕ evaluates to $-\infty$, or
$\phi = 0$ and ϕ evaluates to $-\infty$ or $+\infty$, or
$\phi \bowtie 0$ and ϕ evaluates to $\pm\infty$

then
  return TRUE ▷ The loop terminates
else
  return FALSE ▷ Can't prove termination
end if

Note that the algorithm does not execute the loop, but soundly abstracts the limit behavior of the induction variables, substitutes the computed limits in the loop condition expressions, and determines whether the loop condition is TRUE or FALSE. The TRUE outcome implies that the loop terminates.

Lemma 2. If the initial value of an induction variable x_0 is in an RGD, and $f(x_0) > x_0$, then x^+ diverges to $+\infty$. (For a Quadrant II,IV update function f, the lemma checks $f^2(x_0) > x_0$ instead, and concludes x^{++} diverges to $+\infty$.)

Proof. As usual, we case-split on the shape of the update function f:

Case 1 (Quadrants I,III) When there are two RGDs, they are $(-\infty, r_{\mathsf{min}})$ and $(r_{\mathsf{max}}, +\infty)$. Only in the latter region is $f(x) > x$, and the inequality holds throughout the region. Since r_{max} is the largest stable point, and $f(x)$ is always greater than x, we have x^+ diverging to $+\infty$. When all of $(-\infty, \infty)$ is the RGD, then $f(x)$ is linear with leading coefficient 1 and a non-zero additive constant, so if $f(x) > x$, then divergence to $+\infty$ is obvious.

Case 2 (Quadrants I,II) When there are two RGDs, they are $(-\infty, r'_{\mathsf{min}})$ and $(r_{\mathsf{max}}, +\infty)$. For the RGD $(r_{\mathsf{max}}, +\infty)$, the same argument as in Case 1 applies. For the RGD $(-\infty, r'_{\mathsf{min}})$, we recall that r'_{min} was the smallest solution of $f(x) = r_{\mathsf{max}}$. Since in this case, $\lim_{x\to-\infty} f(x) = +\infty$, we know that $\forall x \in (-\infty, r'_{\mathsf{min}}) \,.\, f(x) > r_{\mathsf{max}}$. Therefore, after one iteration, the induction variable will map to the $(r_{\mathsf{max}}, +\infty)$ RGD, and hence diverge to $+\infty$ as well.
When there is only the single RGD $(-\infty, \infty)$, then $f(x) > x$ for all x. Furthermore, since f is a finite-degree polynomial, $f(x) - x \geq \epsilon$ for some $\epsilon > 0$. So, on each iteration, the induction variable must increase by at least ϵ, and hence diverges to $+\infty$.

Case 3 (Quadrants III,IV) This is analogous to Case 2.

Case 4 (Quadrants II,IV) In this case, f is an odd-degree polynomial with a negative leading coefficient. Therefore, $f^2 = f \circ f$ will have odd degree as well, but with a positive leading coefficient, which means f^2 falls under Case 1.

□

Lemma 3. If the initial value of an induction variable x_0 is in an RGD, and $f(x_0) < x_0$, then x^+ diverges to $-\infty$. (For a Quadrant II,IV update function f, the lemma checks $f^2(x_0) < x_0$ instead, and concludes x^{++} diverges to $-\infty$.)

Proof. The proof is the same (mutatis mutandis) as for Lemma 2. □

Loop	a	b	c	d	e	f
Time (s)	0.03	0.03	0.02	0.03	0.03	0.03
Result	✓	✓	✓	✓	✓	⊘

Table 2. Results of experiments using ZigZag on both hand-crafted and Terminator-produced NAW-loops. The symbol ✓ indicates that ZigZag was able to prove the loop terminates. The symbol ⊘ means that ZigZag hasn't been able to prove termination.

Lemma 4. If the abstract evaluation of the loop condition, performed as described in Section 2.4, evaluates to FALSE, the loop must terminate.

Proof. The loop condition is a conjunction of comparisons of the form $\phi \bowtie 0$, where $\bowtie \in \{<, >, \leq, \geq, =\}$. Only if some conjunct provably must become false does the overall condition abstractly evaluate to FALSE. In that case, during program execution, that conjunct in the loop condition must eventually become false at some iteration, making the loop terminate.

We assume standard, conservative evaluation with the abstract domain. Therefore, if we conclude ϕ evaluates to $+\infty$, $-\infty$, or $\pm\infty$, we know that ϕ must diverge in the positive direction, negative direction, or alternating sign on each iteration, respectively. We evaluate the loop condition to be FALSE only when the condition is of the form:

$\phi < 0$ and ϕ evaluates to $+\infty$, or
$\phi \leq 0$ and ϕ evaluates to $+\infty$, or
$\phi \geq 0$ and ϕ evaluates to $-\infty$, or
$\phi > 0$ and ϕ evaluates to $-\infty$, or
$\phi = 0$ and ϕ evaluates to $-\infty$ or $+\infty$, or
$\phi \bowtie 0$ and ϕ evaluates to $\pm\infty$.

In each case, the divergence of ϕ forces the condition to eventually be false at some iteration. □

Theorem 1. From Lemmas 2, 3 and 4, it follows that the proposed algorithm is sound, i.e., it will never falsely report that a loop terminates if it doesn't.

Complexity. Univariate (and multivariate) polynomials with rational coefficients can be factored in polynomial time $\mathcal{O}(n^{12} + n^9(\log|f|^3))$, where n is the degree of the polynomial f and $\left|\sum_i a_i X^i\right| = \sqrt{(\sum_i a_i^2)}$ using a well-known LLL algorithm [19, 21]. Each additional test can be performed by evaluating the update function at specific points, hence these tests are performed in time linear with the size of the update polynomial. The simple, abstract evaluation of a multivariate polynomial $f \in \mathbb{Q}[\mathcal{V}]$ over our abstract domain can also be computed in linear time. Therefore, the overall complexity of the analysis is polynomial. The number of induction variables is in general small and the experiments confirm that the computational cost of the approach is negligible.

Experiments. To test our approach, we have prototyped our analysis in Maple [24]. Our prototype, called ZigZag, is designed to accept inputs of the form described in Figure 1(a). We ran ZigZag on both hand-crafted and real-life examples of linear and nonlinear NAW loops given in Figure 3. Because of nonlinearity, tools which are restricted to linear cases, such as RankFinder, can't handle most of these loops.

Our initial results are encouraging (see Table 2). Note that performance is currently not an issue— all examples are handled in less than 0.03 seconds. ZigZag was able to prove that all the loops, except for the last one, terminate. The last loop does not terminate because the induction variable x diverges towards $-\infty$, and therefore the loop condition never evaluates to FALSE.

3. Progress Measure Abstractions

Our goal is to integrate the procedure proposed in the previous section, ZigZag, within a larger termination proving framework, such as Terminator [11]. The difficulty here is that tools such as Terminator use explicit ranking functions (*i.e.* maps from the program state space into a well-ordered set) found from analyzing simple loops when constructing termination arguments for the whole program. This whole-program

$x \leftarrow 1;$
$-32768 \leq LIMIT \leq 32767;$
while $x > 0 \wedge x < LIMIT$ **do**
$\quad \ldots$
$\quad x \leftarrow 5x;$
end while

(a) Linear walking of an array

$i \leftarrow 0; j \leftarrow 0;$
$-32768 \leq MAX \leq 32767;$
while $i \cdot j \leq MAX$ **do**
$\quad \ldots$
$\quad i \leftarrow i + 1;$
$\quad \ldots$
$\quad j \leftarrow j + 1;$
end while

(b) Walking the diagonal of a 2D array

$place \leftarrow 1;$
$base \leftarrow 10;$
$-32768 \leq SRC \leq 32767;$
while $-base^{place} + SRC \geq 0$ **do**
$\quad \ldots$
$\quad place \leftarrow place + 1;$
$\quad \ldots$
end while

(c) Finding maximal exponent

$60 \leq TOUT \leq 300;$
$rto \leftarrow 3;$
while $rto < TOUT$ **do**
$\quad \ldots$
$\quad rto \leftarrow 2rto;$
end while

(d) Model of TCP's retransmission timer

$x \leftarrow 4; 2 \leq Y \leq 1024$
while $x \geq -2^{64}$ **do**
$\quad x \leftarrow -2x^5 + 4x^4 - 8x + Y;$
end while

(e) Hand-crafted example

$x < -2;$
while $x < 10$ **do**
$\quad x \leftarrow x^3 - 2x^2 - x + 2;$
end while

(f) Hand-crafted example

Fig. 3. Examples of NAW loops used for testing our approach. The symbol "$\cdots$" indicates where program slicing has been performed. For the examples 3(a), 3(b), 3(c), 3(d), and 3(e) our algorithm reported TRUE (i.e., the loop provably terminates), whereas for the example 3(f) the algorithm reported FALSE(i.e., could not prove termination).

termination argument is then checked using other static analysis techniques. But ZigZag does not produce an explicit ranking function. Furthermore, tools like Terminator do not support nonlinear program commands and thus convert them into coarse linear abstractions. Thus, even if we find a ranking function, the whole-program check will likely still fail due to the coarse abstractions used for nonlinear commands.

In this section, we describe a method for producing additional linear measures based on our termination analysis that can be used as annotations in the original program. The idea is to annotate the program with progress measures proved to exist. These progress measures are updated using only linear commands. Our procedure is called α_{ZigZag}. The procedure, when passed a path π and a program P attempts to prove the path π to be terminating, and then produces a new program that is an abstraction of P, which can be used to prove properties such as safety and termination. This new program contains fresh variables used as progress measures. The control flow graphs of the two programs are identical. The resulting abstraction can then be passed directly to a tool like Terminator.

Consider any finite path segment through the control-flow graph G of a program, which — when unrolled, interpreted in G, and converted into static single assignment form [14] — can be represented as a conjunction of inequalities. Tools such as RankFinder [26], PolyRank [4, 5, 6], and ZigZag can be used to prove that relations like these are well-founded, meaning that a progress measure can be tracked on each execution of the path segment.

We assume a set Cmd of syntactic commands (assignments, assumptions, etc.), which are interpreted as relations between program states before and after execution of a command. Hence, a command $c \in \mathsf{Cmd}$ denotes a relation from $2^{\mathsf{St} \times \mathsf{St}}$, where St is the set of states.

Programs are viewed as control-flow graphs (CFG, e.g., [25]) where the vertices are program locations labeled with commands. Our formulation of a CFG slightly differs from the standard representation to simplify the presentation: all control-flow is achieved by nondeterministic jumps, and branch guards are represented with assumptions. Thus, control-flow statements (if, while, ...) are represented with non-deterministic edges and assume statements, instead of branch statements and deterministic edges used in the standard definition of CFGs.

We will assume that a *program* G is a rooted vertex-labeled directed graph $(\mathbb{L}, \mathcal{L}_G, \mathbb{E})$, where $\mathbb{L}$ is the set of program locations, $\mathcal{L}_G$ is the labelling function of type $\mathbb{L} \to \mathsf{Cmd}$, and $\mathbb{E} \in 2^{\mathbb{L}\times\mathbb{L}}$ is the set of nondeterministic jump edges between program locations. We assume that $1 \in \mathbb{L}$ is the start location of the CFG. A *path* π is a possibly infinite sequence of program locations $\pi = \ell_0 \to \ell_1 \to \cdots$, such that $(\ell_i, \ell_{i+1}) \in \mathbb{E}$. For such a path π, we write π_i for ℓ_i. Path *segments* are contiguous subsequences of paths. Let π be a finite path segment of G that forms the cycle in a lasso candidate counterexample to termination, and let ρ_π be the binary relation on states which is the composition of the statements at vertex: $\rho_\pi = \mathcal{L}_G(\pi_0)\,;\mathcal{L}_G(\pi_1)\,;\cdots;\mathcal{L}_G(\pi_n)\,.$ Suppose that ZigZag has proved the well-foundedness of ρ_π. In this case we can construct an abstraction of G in which a progress measure decreases on each pass over π, and in which the progress measure is nondeterministically assigned to a new value whenever the program executes a statement not in π that modifies a variable of the ranking function found by our well-foundedness checker.

Let $\mathsf{fv}(c)$ be the set of free (program) variables occurring in the command c. Then, let W be the variables the ranking function depends on — if we only know of the existence of a ranking function then it is sound to assume that W depends on all the variables in the path: $W = \mathsf{fv}(\pi) = \bigcup_i \mathsf{fv}(\mathcal{L}_G(\pi_i))$. We will model the ranking function with a fresh variable m. Given the path $\pi = \pi_0 \longrightarrow \cdots \pi_n$ we can construct a new linear abstraction in which the progress measure m has been instrumented in along that path. This abstraction uses a fresh variable a to track where the execution has been. Values of a are pairs of a location and an index into the path where $a = (-1, -1)$ indicates that execution has strayed from the path. Since paths may visit a particular location more than once, the index part of a is necessary to uniquely identify points on the path. The construction of the abstraction is found in Algorithm 2. In essence, the abstraction that is constructed represents an automaton that accepts contiguous sub-executions that follow π. Each time π is followed, the progress measure m is decremented. If the execution leaves the path π and enters a program location that could affect the value of the ranking function, the progress measure's value is lost. The procedure constructs a new control-flow graph with labeling function $\mathcal{L}_A$ which annotated at locations ℓ with update relations where variables with a `_s` suffix are used to represent the previous state. We assume fresh variables i and m. The variable Ψ is used to construct the new relation which is eventually stored in $\mathcal{L}_A(\ell)$.

Proposition 1 (Soundness). If π is a finite path segment of G then $\alpha_{\text{ZigZag}}(G, \pi)$ is a sound abstraction with respect to termination of G.

Proof. (Sketch) The abstraction produces a program with an identical control-flow graph in which only new variables are read/written. The only relation introduced that could potentially remove executions allowed by the original program is introduced by this case (due to the inequality $m > 0$):

$\Psi \leftarrow \Psi \vee (a_s = (\pi_n, n) \wedge a = (-1, -1) \wedge m > 0 \wedge m = m_s - 1)$

This clause could only remove transitions in the case where π is executed infinitely often and in which the paths followed between passes through π have no effect on the ranking function proving π's well-foundedness. Thus these executions would not exist in G (due to the fact that ρ_π is well-founded). □

Example. Consider the example in Figure 4, and the path through this loop that goes repeatedly into the `if` branch with the nonlinear assignment (ignoring the line numbers that do not have assignment statements or assume statements): $\pi = 3 \to 5$. In this case, ρ_π is the binary relation on states (where y is used as the mathematical variable representing the value stored in `n` and y' represents the new value of `n` after the assignment): $\rho_\pi = (y' = y^2 \wedge 1 < y' \wedge y' < b)$. ZigZag can prove the well-foundedness of ρ_π, meaning that we can construct a transition relation which monitors G's execution and decrements a progress measure m when the path is executed. Figure 5 illustrates how this is done. Note that a standard termination prover—such as Terminator—can then prove the termination of the resulting program.

4. Related Work

A number of techniques are available for proving termination of programs (e.g., [12, 26, 8, 29, 4, 9, 17, 7, 20, 22, 27, 11, 10]). With the exception of a few cases [1, 4, 12], these tools are all limited to linear arithmetic.

Algorithm 2 α_{ZigZag}: ZigZag-based path abstraction procedure. The procedure constructs a control-flow graph annotated with update relations where variables with a _s suffix are used to represent the previous state. We assume fresh variables i and m.

input: a program G
input: finite path segment $\pi = \pi_0 \longrightarrow \cdots \pi_n$

$A := G$ ▷ Initialize A

if $\neg$ZigZag(π) **then** ▷ Try to prove ρ_π well-founded using nonlinear well-foundedness prover
  return A
end if

W := support of ranking function or $\mathsf{fv}(\pi)$ if the support is not known

for all $\ell \in \mathbb{L}$ **do**
  if $\exists i.\ \ell = \pi_i \wedge 0 \leq i \leq n$ **then**
    for all i s.t. $\ell = \pi_i \wedge 0 \leq i \leq n$ **do**
      $\Psi \leftarrow FALSE$
      if $\ell = \pi_i \wedge i > 0$ **then**
        $\Psi \leftarrow a_s \neq (\pi_i, i) \wedge a = (-1, -1)$
      end if
      if $\ell = \pi_0$ **then**
        $\Psi \leftarrow \Psi \vee (a = (\pi_1, 1) \wedge m = m_s)$
      end if
      if $\ell = \pi_i \wedge 0 < i < n$ **then**
        $\Psi \leftarrow \Psi \vee (a_s = (\pi_i, i) \wedge a = (\pi_{i+1}, i+1) \wedge m = m_s)$
      end if
      if $\ell = \pi_n$ **then**
        $\Psi \leftarrow \Psi \vee (a_s = (\pi_n, n) \wedge a = (-1, -1) \wedge m > 0 \wedge m = m_s - 1)$
      end if
    end for
  else
    $\Psi \leftarrow a = (-1, -1)$
    if $\mathcal{L}_G(\ell)$ does not modify a variable in W **then**
      $\Psi \leftarrow \Psi \wedge m = m_s$
    end if
  end if
  $\mathcal{L}_A(\ell) \leftarrow \mathcal{L}_A(\ell) \wedge \Psi$
end for
return A

```
1  j = 0;
2  while (nondet()) {
3    assume(1 < n && n < b);
4    if (nondet()) {
5      n = n * n;
6    } else {
7      j = j + 1;
8      n = j;
9    }
10 }
```

Fig. 4. An example `while` loop. As explained earlier, the assumption on line 3 represents the loop guard. Each iteration of the loop either squares the contents of `n` or increments the counter `j`. The termination condition of the loop obviously involves reasoning about non-linearities. The function nondet() is assumed to be nondeterministic value introduction (*i.e.* a proof of correctness will have to consider all possible cases of nondet()'s result).

```
1  j = 0;
1  a_s=a; m_s=m;
1  a=nondet(); m=nondet();
1  assume(a==(-1,-1) && m==m_s);
2  while(nondet()) {
3    assume(1 < n && n < b);
3    a_s=a; m_s=m;
3    a=nondet(); m=nondet();
3    assume(a==(5,1) && m==m_s);
4    if(nondet()) {
5      n = n * n;
5      a_s=a; m_s=m;
5      a=nondet(); m=nondet();
5      assume((a_s!=(5,1) && a==(-1,-1)) || (a_s==(5,1) && a==(-1,-1) && m>0 && m==m_s-1));
6    } else {
7      j = j+1;
7      a_s=a; m_s=m;
7      a=nondet(); m=nondet();
7      assume(a==(-1,-1) && m==m_s);
8      n = j;
8      a_s=a; m_s=m;
8      a=nondet(); m=nondet();
8      assume(a==(-1,-1));
9    }
10 }
```

Fig. 5. Output of $\alpha_{\text{ZigZag}}\,\pi\,G$ (Algorithm 2) where the program G is from Figure 4 and $\pi = 3 \rightarrow 5$. Variables with the `_s` suffix store the previous state.

This paper extends our previous work [1] with a more in-depth algorithmic description, together with new details on how ZigZag can be combined within a larger termination proving framework.

Bradley et al. [4]'s analysis supports nonlinear multipath polynomial programs. Their algorithm is based on building finite difference trees for expressions. In some cases that we can handle (see Figures 3(a), 3(c), 3(d), and 3(e) for examples) this can't be done as the trees are infinite.

Cousot [12]'s general framework for termination proving supports the synthesis of ranking functions from nonlinear loops. First, the standard Floyd-style verification conditions are abstracted into a (user-chosen) parametric form. If it is still possible to prove termination within the parametric abstraction, then verification reduces to solving for the parameters. The constraints on the parameters are then further abstracted to a conjunction of inequalities by Lagrangian relaxation, which can in turn be solved using semidefinite programming, if the constraints are quadratic. The framework is general and can handle interactions between variables that our analysis cannot. On the other hand, semi-definite programming is limited to quadratic functions, so the framework can handle polynomial updates of higher degree only if they can be expressed (or further conservatively approximated) by a sum of squares. Furthermore, the framework depends on extensive numerical computation, making it vulnerable to numerical errors and complicating its use in practice. (For example, a synthesized rank function in Cousot's paper [12] is actually non-monotonic when n is large, apparently due to numerical errors in the solver, or perhaps a transcription error somewhere in the process.[3])

In Section 3, we proposed a method of adding new variables which track facts proved by ZigZag. The

---

[3] The error occurs in Example 9, where the synthesized rank function is reported as $r(n,i,j) = (7.024176 \times 10^{-4})n^2 + (4.394909 \times 10^{-5})ni - (2.809222 \times 10^{-3})nj + (1.533829 \times 10^{-2})n + (1.569773 \times 10^{-3})i^2 + (7.077127 \times 10^{-5})ij + (3.093629 \times 10^{1})i - (7.021870 \times 10^{-4})j^2 + (9.940151 \times 10^{-1})j + (4.237694)$. This is for a nested loop:

```
for (i=n; i>=0; i--) for (j=n; j>=0; j--) /* skip */;
```

and the rank function should decrease with every iteration. The paper shows a well-behaved plot when $n = 10$. However, an examination of the purported ranking function suggests that when n is somewhat larger and the inner loop terminates, the effect of i decrementing by 1 will not overcome j increasing back to n. For example, using arbitrary precision rationals, we can compute $r(100, 1, 0) = 43.737953682$, whereas $r(100, 0, 100) = 77.083119$. As n grows larger, other non-monotonicities emerge. For example, $r(1000, 0, 1000) = -1795.400316$, but $r(1000, 0, 999) = -1792.181437287$.

idea of tracking additional information with new variables for the purpose of proving termination is not new (e.g. [3, 23]). The difference here is in what is tracked: we are using new variables to track if specific paths are taken, whereas previous techniques have been used to track artifacts such as the sizes of lists.

5. Conclusion

We have described an automatic tool, called ZigZag, that can be used to prove the termination of loops with nonlinear assignments to variables. ZigZag uses divergence testing on variables that are relevant for proving loop's termination. ZigZag can automatically prove the termination of loops that are not supported with previously reported techniques.

Future work. ZigZag could be extended in several directions. In addition to divergence, x^+ might also converge to a stable point. The analysis can be extended to handle such cases, for example:

```
while (x > 0.5) {
    x = 0.8 * x;
}
```

This is just an application of the Banach fixed point theorem [15]. Since this is a well-known result, we do not consider it explicitly in our approach.

For low-order polynomials (which have only a small number of real roots) with only constant coefficients, the analysis could search for RGDs (or regions of guaranteed convergence) in an iterated application of the update function $f^k = f \circ \cdots \circ f$, which is simply a higher-order polynomial. This extension would make the analysis much more powerful, as currently our analysis only finds the most apparent RGDs. Since most of the nonlinear functions that appear in practice are low-order polynomials, this might be an interesting direction for the future research.

We are currently investigating methods for supporting interdependent induction variables. Another possible extension is to allow nonlinear functions of symbolic constants in update functions. Finding minimums and maximums of such functions is $\mathcal{NP}$-complete in general.

Acknowledgments

We would like to thank Richard Fateman, Robert Israel, and Daniel Lichtblau for useful discussions about the complexity of polynomial factorization, divergence of the update functions, and multivariate limits, and Josh Berdine and Jacopo Mantovani for their comments on Section 3. We would also like to thank the anonymous reviewers for many helpful comments and for pointing out the special case in the computation of safe RGDs.

References

[1] D. Babić, B. Cook, A. J. Hu, and Z. Rakamarić. Proving termination by divergence. In *IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 93–102, 2007.

[2] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 211–224, 2007.

[3] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *International Conference on Computer Aided Verification (CAV)*, pages 517–531, 2006.

[4] A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 113–129, 2005.

[5] A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1349–1361, 2005.

[6] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination analysis of integer linear loops. In *International Conference on Concurrency Theory (CONCUR)*, pages 488–502, 2005.

[7] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

[8] M. Colón and H. Sipma. Practical methods for proving program termination. In *International Conference on Computer Aided Verification (CAV)*, pages 442–454, 2002.

[9] E. Contejean, C. Marché, B. Monate, and X. Urbain. Proving termination of rewriting with CiME. In *Extended Abstracts of the 6th International Workshop on Termination (WST)*, pages 71–73, 2003.

[10] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *International Static Analysis Symposium (SAS)*, pages 87–101, 2005.

[11] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426, 2006.

[12] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 1–24, 2005.

[13] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[15] J. Dugundji and A. Granas. *Fixed Point Theory.* Springer-Verlag, New York, NY, USA, 1st edition, 2003.

[16] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[17] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *International Conference on Rewriting Techniques and Applications (RTA)*, pages 210–220, 2004.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10): 576–583, 1969.

[19] E. Kaltofen. *On the complexity of factoring polynomials with integer coefficients.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 1982.

[20] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 81–92, 2001.

[21] A. K. Lenstra, J. Hendrik W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[22] N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. TermiLog: A system for checking termination of queries to logic programs. In *International Conference on Computer Aided Verification (CAV)*, pages 444–447, 1997.

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

[24] Maplesoft. Maple, version 9.5, 2004.

[25] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[26] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 239–251, 2004.

[27] C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In *International Static Analysis Symposium (SAS)*, pages 160–171, 1997.

[28] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[29] A. Tiwari. Termination of linear programs. In *International Conference on Computer Aided Verification (CAV)*, pages 70–82, 2004.

[30] A. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating machines*, pages 67–69, 1949.