Proving Termination by Divergence*

Domagoj Babić, Alan J. Hu, Zvonimir Rakamarić
Department of Computer Science, University of British Columbia
{babic,ajh,zrakamar}@cs.ubc.ca

Byron Cook
Microsoft Research
bycook@microsoft.com

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 termination condition. The analysis allows us to automatically prove the termination of loops that cannot be handled using previous techniques. The paper closes with experimental results using short examples drawn from industrial code.

1 Introduction

From the very beginnings of the formal analysis of software [12, 14], the task of formally verifying the correctness of a program has been decomposed into the tasks of proving correctness *if* the program terminates, and separately proving termination. Deciding termination, in general, is obviously undecidable, but thanks to considerable research progress over the years (e.g., [9, 20, 5, 23, 3, 6, 13, 4, 16, 18, 21, 8, 7]), a variety of techniques and heuristics can now automatically prove termination of many loops that occur in practice.

Today's automatic program termination provers, 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

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

This paper outlines a new proof procedure for cases of this sort. Using combination techniques described in [1] and [2], our intention for this proposed procedure is to be combined with the existing termination analysis techniques—making future termination provers a little less temperamental.

The proposed technique is based on 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 all automated termination provers, the technique can't handle all loops. However, it is very 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 sym-

*This work was supported by an NSERC Discovery Grant, a Microsoft Graduate Fellowship, and a graduate fellowship from the University of British Columbia.

bolic 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 lowercase letters x, y, z. Both types of variables are considered to be ideal rationals We will assume that the loop test is a conjunction of a finite number of (in)equality relations of the form[1] $\phi \bowtie 0$, where $\bowtie \in \{<, >, \leq, \geq, =\}$. 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 $\hat{f} = f \circ f$ be a composition of f with itself. The value obtained by applying $\hat{f}$ 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 $\mathsf{supp}\,(\psi)$ 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 back-end within the frameworks described in [2] or [1]. Both [2] and [1] 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*. If each simple loop produced is well-founded, then the original program is guaranteed to terminate. A multitude of techniques can be used to prove the well-foundedness of these simple loops.

Our goal in this paper is to provide support for non-linear 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 $\mathcal{B} = \{v \bowtie q \in \mathbb{Q}\}$ over symbolic constants and induction variables $v \in \mathcal{V} \cup \mathcal{S}$ represents the initial conditions. Symbolic constants must be from a closed finite interval.

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.)

3. The loop termination condition is a conjunction of relations $\phi(\mathcal{V} \cup \mathcal{S}) \bowtie 0$ such that each $\phi()$ 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). For simplicity, this paper will focus on a single conjunct. With multiple conjuncts, if any conjunct can be proven to be eventually false, the loop must terminate.

4. Update functions $f_i\,()$ are univariate polynomials of finite degree from $\mathbb{Q}[x_i]$.

5. Given a linear function $g : \mathcal{S}^n \rightarrow \mathbb{Q}$ 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 $g_i(Y_1, Y_2, \dots)$.

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 (Non-linear 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 test 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 termination condition. Standard slicing techniques [22] 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 test.

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 [10]. The left column (Figures 1(b),1(d),1(f)) contains examples of loops that are not NAW loops. The loop test 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 not a constant term (Figure 1(d)) and the induction variable x (Figure 1(f)) depends on the induction variable n. The right

[1]The constraints on $\phi()$ will be given soon.

Range constraints
while $\phi(\mathcal{V} \cup \mathcal{S}) \bowtie 0, \bowtie \in \{<, >, \leq, \geq, =\}$ **do**
$x_1 \leftarrow f_1(x_1) + X_1;$
$x_2 \leftarrow f_2(x_2) + X_2;$
$x_3 \leftarrow f_3(x_3) + X_3;$
$\ldots$
$x_n \leftarrow f_n(x_n) + X_n;$
end while

(a) Generic form of NAW loops

$x < -17; -32 < Y;$
while $x \neq Y$ **do**
$x \leftarrow 3x - 1$
end while

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

$e \leftarrow 0.00001; a \leftarrow 1 - e; x \leftarrow 0.5;$
while $e \leq x \wedge x \leq 1$ **do**
$x \leftarrow -a \cdot x^2 + a \cdot x;$
end while

(c) Terminating NAW loop

$x \leftarrow 5; -10 < B < 10;$
while $x < 1000$ **do**
$x \leftarrow x^2 - B \cdot x - 7$
end while

(d) Non-NAW loop – Symbolic constant B is not a constant term

$q \leftarrow 0; r \geq 0; 1 \leq Y \leq 65535;$
while $y \leq r$ **do**
$r \leftarrow r - Y$
$q \leftarrow q + 1$
end while

(e) Terminating NAW loop

$n \leftarrow 0; x \leftarrow 1; 1 < N < 1000;$
while $x \leq N$ **do**
$n \leftarrow n + 1$
$x \leftarrow n \cdot x$
end while

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

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

(g) Non-terminating NAW loop

Figure 1. General NAW-form together with some examples

column contains three examples of NAW loops. Examples 1(c) and 1(e) are adapted from Cousot's paper [9].

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 termination condition 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 [15, 17] 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 $(r_{\mathsf{max}}, \infty)$ (subject to some side conditions) are RGDs.[2] 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:

[2] 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 $-\infty$.

Case 1: Divergence in Quadrants I and III This is the most straightforward case. The update function f must be a polynomial of odd degree, with a positive leading coefficient. 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 In this case, the update function must be a polynomial of even degree, with a 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. We cannot say anything about $(-\infty, r_{\mathsf{min}})$, but for sufficiently small initial value x_0, we know that $f(x_0)$ must be positive and in the 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 This is the dual of Case 2; the update function has even degree, but a negative leading coefficient. 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 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, $\hat{f} = f \circ f$ is an update function that follows Case 1. So, we compute the stable points of $x = \hat{f}(x)$ instead, and analyse $\hat{f}$ 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.

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

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 $f(x) = x$

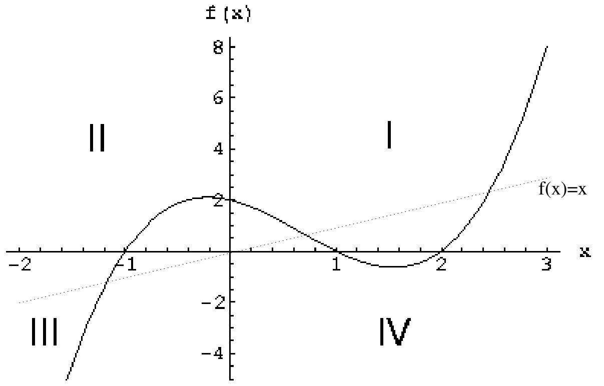

Figure 2. $f(x) = x^3 - 2x^2 - x + 2$**. Quadrants are denoted by the Roman numerals I,II,...**

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 test. 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[3] in the polynomial $f(x)$ is a symbolic constant. Let's assume a symbolic constant $0 \leq Z \leq 30$, the previously given loop test, 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) *An RGD of a polynomial assignment containing symbolic constant terms is said to be safe if the effect of worst-case shifts of the polynomial caused by symbolic constants are taken into account.*

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.[4]

Computing safe RGDs doubles the amount of work, as it requires two factorizations of the polynomial. 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, further on we will not explicitly discuss the safe regions. The algorithm and the proofs can be trivially extended to handle them.

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

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

[4] The only exception to this procedure occurs in the special case that the update function is linear and has leading coefficient 1, and the symbolic constant shift allows the update to be the identity function. In this degenerate case, the safe RGD is empty, since there is no RGD when the induction variable keeps the same value on each iteration. This special case can be easily identified and handled separately.

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 $\pm\infty$.

Notice that at most two quadrants are relevant in each case. In Cases 2 and 3, executing the update assignment once will move all the action to a single quadrant, rendering the divergence analysis easy.

2.4 Termination Analysis

The termination analysis boils down to computing a limit of the multivariate polynomial in the loop test 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 in the test condition $-x^2$ would incorrectly compute that the left side of the loop test diverges towards $-\infty$ and that the loop terminates.

Instead, we make a safe, conservative evaluation of the test condition given the information we have. Essentially, the computation is the straightforward abstract interpretation [10] of the termination condition, where the induction variables have values in the abstract domain $\{-\infty, +\infty, \pm\infty, \perp\}$, depending on the result of the divergence analysis. (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 $y = +\infty$. Then, we would evaluate $x^2 = +\infty$, and $y - x^2 = +\infty - (+\infty) = \perp$. The test $\perp > 100$ is not provably false, so we do not claim termination. By expanding the abstract domain (e.g., to indicate different growth rates, convergence to constants, etc.), we can straightforwardly improve the accuracy of the termination analysis, but this was not needed for the examples we studied.

The outline of the overall algorithm we implemented is given below:

1. If each $v \in \mathsf{supp}\,(\phi())$ has a unique value and $\phi() \bowtie 0$ evaluates to FALSE, report "THE LOOP TERMINATES".

2. Find the RGDs for each induction variable.

3. For each induction variable determine its divergence behavior.

4. Evaluate the loop test condition using the divergence information conservatively. If the loop test is

   $\phi() < 0$ and $\phi()$ evaluates to $+\infty$, or

   $\phi() \leq 0$ and $\phi()$ evaluates to $+\infty$, or

   $\phi() \geq 0$ and $\phi()$ evaluates to $-\infty$, or

   $\phi() > 0$ and $\phi()$ evaluates to $-\infty$, or

   $\phi() = 0$ and $\phi()$ evaluates to $-\infty$ or $+\infty$, or

   $\phi() \bowtie 0$ and $\phi()$ evaluates to $\pm\infty$,

   report "THE LOOP TERMINATES". Otherwise, report "CAN'T PROVE TERMINATION".

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

3 Correctness

In this section, we discuss the correctness of the algorithm for proving termination of NAW loops. The proof of correctness consists of two parts. First, we need to prove that the divergence analysis is correct. Then, we need to prove the correctness of the termination analysis.

Lemma 1 *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 $\hat{f}(x_0) > x_0$ instead, and concludes x^{++} diverges to $+\infty$.)*

Proof 1 *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,* $\hat{f} = f \circ f$ *will have odd degree as well, but with a positive leading coefficient, which means* $\hat{f}$ *falls under Case 1.*

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* $\hat{f}(x_0) < x_0$ *instead, and concludes* x^{++} *diverges to* $-\infty$.*)*

Proof 2 *The proof is the same (mutatis mutandis) as for Lemma 1.*

Lemma 3 *If the abstract evaluation of the termination condition described in Section 2.4 evaluates to* FALSE, *the loop must terminate.*

Proof 3 *The termination 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 termination condition abstractly evaluate to* FALSE. *In that case, during program execution, that conjunct in the loop termination 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 test to be* FALSE *only when the test 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 1, 2 and 3, it follows that the proposed algorithm is sound, i.e., it will never falsely report that a loop terminates if it doesn't.*

4 Complexity

Univariate (and multivariate) polynomials 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 [15, 17]. 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.

5 Experimental Results

To test our approach, we have prototyped our analysis in Maple [19]. 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 Figure 4). 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 test never evaluates to FALSE.

$x \leftarrow 1;$
$-32768 \leq LIMIT \leq 32767;$
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

Figure 3. Examples of NAW loops used for testing our approach. The symbol "$\ldots$" indicates where program slicing has been performed. For the examples 3(a), 3(b), 3(c), 3(d), and 3(e) our algorithm reported "THE LOOP TERMINATES", while for the example 3(f) the algorithm reported "CAN'T PROVE TERMINATION".

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

Figure 4. 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.**

6 Related Work

A number of techniques are available for proving termination of programs (*e.g.* [9, 20, 5, 23, 3, 6, 13, 4, 16, 18, 21, 8, 7]). With the exception of [3] and [9] these tools are all limited to linear arithmetic.

The analysis described in [3] supports nonlinear multipath polynomial programs. The algorithm is based on building finite difference trees for expressions. In some cases which 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 [9] presents a general framework for proving termination (and invariants) and synthesizing polynomial ranking functions of 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 very 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 the paper [9, Example 9] is actually non-monotonic when n is large, apparently due to numerical errors in the solver.)

7 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 to the loop's termination condition. 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 [11]. 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) between each pair of adjacent stable points. 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. We would also like to thank the anonymous reviewers for several helpful comments and for pointing out the special case in the computation of safe RGDs.

References

[1] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *POPL'2007: Principles of Programming Languages*, pages 211–224. ACM Press, 2007.

[2] J. Berdine, B. Cook, and J. Mantovani. Combining linear abstractions for termination. In preparation.

[3] A. Bradley, Z. Manna, and H. Sipma. Termination of Polynomial Programs. In *VMCAI'2005: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*. Springer, 2005.

[4] 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.

[5] M. Colón and H. Sipma. Practical Methods for Proving Program Termination. In *CAV'2002: Computer Aided Verification*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002.

[6] E. Contejean, C. Marché, B. Monate, and X. Urbain. Proving Termination of Rewriting with C*i*ME. In *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, pages 71–73, June 2003.

[7] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *SAS'2005: Static Analysis Symposium*, volume 3672 of *LNCS*, pages 87–101. Springer, 2005.

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

[9] P. Cousot. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *VMCAI'2005: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*. Springer, 2005.

[10] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'1977: Principles of Programming Languages*, pages 238–252, 1977.

[11] J. Dugundji and A. Granas. *Fixed Point Theory, 1ed*. Springer-Verlag, New York, NY, 2003.

[12] R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

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

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

[15] E. Kaltofen. *On the complexity of factoring polynomials with integer coefficients*. PhD thesis, RPI, Troy, N. Y., dec 1982.

[16] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL'2001: Principles of Programming Languages*, volume 36, 3 of *ACM SIGPLAN Notices*, pages 81–92. ACM Press, 2001.

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

[18] N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. TermiLog: A System for Checking Termination of Queries to Logic Programs. In *CAV'1997: Computer-Aided Verification*, LNCS, pages 444–447. Springer, 1997.

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

[20] A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI'2004: Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.

[21] C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 160–171, London, UK, 1997. Springer-Verlag.

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

[23] A. Tiwari. Termination of linear programs. In *CAV'2004: Computer Aided Verification*, volume 3114 of *LNCS*, pages 70–82. Springer, 2004.