Termination of Polynomial Programs

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma *

Computer Science Department
Stanford University
Stanford, CA 94305-9045
{arbrad,zm,sipma}@theory.stanford.edu

Abstract. We present a technique to prove termination of multipath polynomial programs, an expressive class of loops that enables practical code abstraction and analysis. The technique is based on finite differences of expressions over transition systems. Although no complete method exists for determining termination for this class of loops, we show that our technique is useful in practice. We demonstrate that our prototype implementation for C source code readily scales to large software projects, proving termination for a high percentage of targeted loops.

1 Introduction

Guaranteed termination of program loops is necessary for many applications, especially those for which unexpected behavior can be catastrophic. Even for applications that are not considered "safety critical," applying automatic methods for proving loop termination would certainly do no harm. Additionally, proving general temporal properties of infinite state programs requires termination proofs, for which automatic methods are welcome [4, 7, 10].

We present a method of *nonlinear* termination analysis for imperative loops with multiple paths, polynomial guards, and polynomial assignments. The method is nonlinear, first, because the guards and assignments need not be linear and, second, because it can prove the termination of terminating loops that do not have linear ranking functions. The method is sound, but not complete. Indeed, we show that no complete method for this class of programs exists. In practical programs, however, our method proves termination of a high percentage of the targeted loops at low computation cost, and hence is useful.

Recent work on automatic proofs of termination for linear imperative loops has mostly focused on the synthesis of *linear ranking functions*. A ranking function for a loop maps the values of the loop variables to a well-founded domain; further, it decreases value on each iteration. A linear ranking function is a ranking function that is a linear combination of the loop variables and a constant. Colón

* This research was supported in part by NSF grants CCR-01-21403, CCR-02-20134 and CCR-02-09237, by ARO grant DAAD19-01-1-0723, by ARPA/AF contracts F33615-00-C-1693 and F33615-99-C-3014, by NAVY/ONR contract N00014-03-1-0939. The first author was additionally supported by a Sang Samuel Wang Stanford Graduate Fellowship.

and Sipma first address the synthesis of linear ranking functions in a deductive manner [2]. They present a method based on the manipulation of *polyhedral cones*, extending these results to loops with multiple paths and nested loops in [3]. In [11], Podelski and Rybalchenko specialize the analysis to a less expressive class of single-path imperative loops, providing an efficient and complete synthesis method based on linear programming. Departing from linear ranking function synthesis, Tiwari proves that the termination of a class of single-path loops with linear guards and assignments is decidable, providing a decision procedure via constructive proofs [13].

In the functional programming community, the *size-change principle* has recently been proposed for termination analysis of functional programs [6]. This effort is largely orthogonal to efforts for imperative loops. The principle focuses on structural properties of functional programs, given that particular expressions decrease or do not increase. While imperative loops, and in particular our abstractions of such loops, may be translated to tail-recursive functional programs, nothing is gained. Finding the proper *size measure* to show termination based on the size-change principle is equivalent to proving termination in the imperative setting. However, it is possible that our work may be applied as a size measure for the termination analysis of some functional programs or recursive functions in imperative programs, thus combining the strengths of each approach.

Our method extends termination analysis to *multipath polynomial programs* (MPPs). We show that this class is sufficiently expressive to serve as a sound abstraction for a large class of loops appearing in ordinary C code. We implemented our method and, via CIL [8], applied it to several large open-source C programs, with size up to 75K lines of code. The timing results clearly demonstrate the practicality of the analysis.

Unlike other recent work, we analyze loops via finite differences. Finite differences have a long history in program analysis (*e.g.*, [14, 5, 1]). These methods construct and solve difference equations and inequations, producing loop invariants, running times, and termination proofs. While the equations and inequations are often difficult to solve, we observe that, for termination analysis anyway, explicit solutions are unnecessary. Rather, our method analyzes loops for qualitative behavior — specifically, that certain expressions *eventually* only decrease by at least some positive amount, yet are bounded from below. We address the challenge of characterizing such behavior in loops with multiple paths and nonlinear assignments and guards.

The rest of the paper is ordered as follows. Section 2 introduces MPPs, while Section 3 develops the mathematical foundations for our analysis. Section 4 then formalizes the termination analysis of MPPs, additionally suggesting an alternate abstraction and analysis based on sets of guarded commands. Section 5 describes our prototype implementation and empirical results, and Section 6 concludes.

2 Preliminaries

Definition 1 (Multipath Polynomial Program) For real variables $\mathbf{x} = (x_1, \ldots, x_n)$, a *multipath polynomial program* (MPP) with m paths has the form shown in Figure 1**(a)**, where $\mathbf{P}_i$ and P_{ij} are a vector and a matrix, respectively, of polynomials in $\mathbf{x}$. θ expresses the initial condition on $\mathbf{x}$.

This abstraction of loops is convenient. Multiple paths and arbitrary Boolean combinations of guard expressions are essential for a straightforward abstraction of real code. Moreover, the initial condition, θ, is useful for expressing invariants of variables unaffected by the loop. Such variables may appear in constant expressions in our analysis.

initially θ
while $\bigvee_i \bigwedge_j P_{ij}(\mathbf{x}) \ \{\geq, >\} \ 0$ **do**
  $\tau_1 : \ \mathbf{x} := \mathbf{P}_1(\mathbf{x})$
    or
    $\vdots$
    or
  $\tau_m : \ \mathbf{x} := \mathbf{P}_m(\mathbf{x})$
od

(a)

while $x \geq y$ **do**
  $\tau_1 : \ (x, \ y) := (x+1, \ y+x)$
    or
  $\tau_2 : \ (x, \ y, \ z) := (x-z, \ y+z^2, \ z-1)$
od

(b)

Fig. 1. (a) Form of multipath polynomial programs. **(b)** Multipath polynomial program CHASE.

Example 1. Consider the MPP CHASE in Figure 1**(b)**. x and y may each increase or decrease, depending on current values. Further, while they both *eventually* increase, termination relies on y increasing more rapidly than x.

Theorem 1. (No Complete Method) *Termination of MPPs is not semi-decidable; that is, there is no complete method for determining termination of MPPs.*

Proof. We construct a reduction from Hilbert's 10^{th} problem, the existence of a nonnegative integer root of an arbitrary Diophantine equation, which is undecidable. First, we note that the existence of such a root is semi-decidable, via a proper enumeration of vectors of integers: if a root exists, the enumeration terminates with an affirmative answer. Thus, the nonexistence of nonnegative integer roots is not semi-decidable. Now, we reduce from the question of *nonexistence* of roots.

Instance: Given Diophantine equation $P(\mathbf{x}) = 0$ in variables $\mathbf{x} = (x_1, \ldots, x_n)$, determine if there does not exist a nonnegative integer solution.

Reduction: Construct the multipath polynomial program with the following variables: (1) one variable corresponding to each x_i, called x_i; (2) counter variable

c; and (3) upper limit variable N. The program has the following form:

$$
\begin{array}{l}
\textbf{initially } c = 1 \wedge \bigwedge_{i=1}^{n} x_i = 0 \\
\textbf{while } c \le N \textbf{ do} \\
\quad \{x_1, c\} := \{x_1 + 1,\ 2 \cdot P(\mathbf{x})^2 \cdot c\} \\
\qquad \textbf{or} \\
\qquad \vdots \\
\qquad \textbf{or} \\
\quad \{x_n, c\} := \{x_n + 1,\ 2 \cdot P(\mathbf{x})^2 \cdot c\} \\
\textbf{od}
\end{array}
$$

The program is a multipath polynomial program, as the loop condition and assignment statements involve only polynomials. Computations in which always $P(\mathbf{x}) \neq 0$ are terminating, as c is initially 1 and at least doubles on each iteration, while N remains constant. If $P(\mathbf{x}) = 0$ does occur in a computation, then c is assigned 0 on the subsequent iteration, and thus for every future iteration. When such a computation is possible, there exist values for N as the upper bound on c so that the computation does not terminate before $P(\mathbf{x}) = 0$; afterward, c remains 0, so the computation does not terminate. Since the program always terminates if and only if there is no solution to the Diophantine equation, we conclude that termination of multipath polynomial programs is neither decidable nor even semi-decidable.

Given this fundamental negative result, this paper focuses on a sound and computationally inexpensive method for concluding termination of multipath polynomial programs. The approach essentially looks for expressions that evolve with polynomial behavior, independently of the order in which transitions are taken. A polynomially behaved expression must eventually only increase, only decrease, or — in a degenerate case — remain unchanged, even if its initial behavior varies. The method that we present soundly classifies expressions that eventually only decrease (or eventually only increase). An expression that eventually only decreases, yet is bounded from below within the loop, indicates termination.

3 Finite Difference Trees

To classify polynomial expressions as eventually only decreasing with respect to a transition system, we use finite differences over transitions. We first recall the definition of a finite difference, placing finite differences in the context of transition systems.

Definition 2 (Finite Difference) The *finite difference* of an expression $E(\mathbf{x})$ in $\mathbf{x}$ over assignment transition τ is

$$\Delta_\tau E(\mathbf{x}) \stackrel{\text{def}}{=} E(\mathbf{x}') - E(\mathbf{x}),$$

where τ provides the value of $\mathbf{x}'$ in terms of $\mathbf{x}$. Thus, $\Delta_\tau E(\mathbf{x})$ is also an expression in $\mathbf{x}$. For convenience, we denote a chain of finite differences $\Delta_{\tau_{i_n}} \cdots \Delta_{\tau_{i_1}} E(\mathbf{x})$

by $\Delta_{\tau_{i_1},\ldots,\tau_{i_n}} E(\mathbf{x})$ or more simply by $\Delta_{i_1,\ldots,i_n} E(\mathbf{x})$ (note the reversal of the list). If $\tau_{i_n} = \cdots = \tau_{i_1}$, we denote the chain by $\Delta^n_{\tau_{i_1}} E(\mathbf{x})$ or more simply by $\Delta^n_{i_1} E(\mathbf{x})$. For list of transitions T with length n, we say that $\Delta_T E(\mathbf{x})$ is an n^{th} *order* finite difference.

Example 2. For program CHASE, the first, second, and third order finite differences of $x - y$ over transition τ_1 are the following:

$$\begin{aligned}
&\Delta_1(x-y) = (x+1) - (y+x) - (x-y) = 1 - x \\
&\Delta_1^2(x-y) = \Delta_1(\Delta_1(x-y)) = \Delta_1(1-x) = 1 - (x+1) - (1-x) = -1 \\
&\Delta_1^3(x-y) = \Delta_1(\Delta_1^2(x-y)) = \Delta_1(-1) = (-1) - (-1) = 0.
\end{aligned}$$

Consider also the first and second order finite differences

$$\begin{aligned}
\Delta_2(x-y) &= (x-z) - (y+z^2) - (x-y) = -(z^2+z) \\
\Delta_{2,1}(x-y) &= \Delta_1(\Delta_2(x-y)) = -(z^2+z) + (z^2+z) = 0.
\end{aligned}$$

Finite differences with respect to transitions in different orders can be represented in a *finite difference tree.*

Definition 3 (Finite Difference Tree) The *finite difference tree* (FDT) of an expression $E(\mathbf{x})$ with respect to transitions $\mathcal{T} = \{\tau_1, \ldots, \tau_m\}$ has root $E(\mathbf{x})$ and branching factor m. Each node, indexed by its position with respect to the root, represents an expression over $\mathbf{x}$; specifically, the node indexed I represents finite difference $\Delta_I E(\mathbf{x})$. The *leaves* of an FDT are nodes with only 0-children (child nodes with value 0). Thus, each leaf is a constant expression with respect to $\mathcal{T}$. The *height* of an FDT is the longest path to a leaf. A *finite FDT* is a finite difference tree with finite height.

For notational convenience, we sometimes refer to FDT nodes by their finite difference expressions; *i.e.*, $\Delta_T E(\mathbf{x})$ is the node indexed by T, where T is the list of transitions that lead from the root to the node.

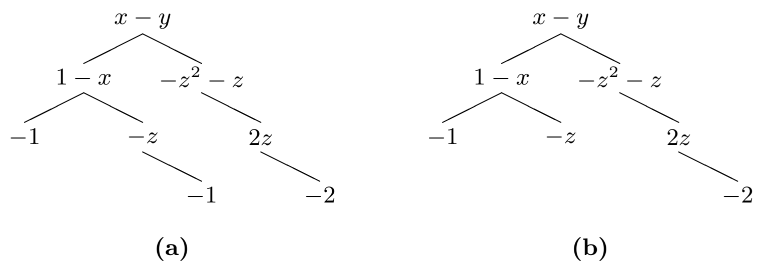


Fig. 2. (a) Finite difference tree for CHASE of $x-y$ with respect to $\{\tau_1, \tau_2\}$. **(b)** Taylored finite difference tree.

Example 3. The FDT of $x-y$ with respect to $\{\tau_1, \tau_2\}$ is shown in Figure 2**(a)**. 0-nodes are not shown. The left node labeled with -1 is indexed (τ_1, τ_1), reflecting that it is the result of twice taking the finite difference with respect to τ_1.

The finite FDT t of an expression $E(\mathbf{x})$ succinctly describes the evolution of $E(\mathbf{x})$. Given computation $\pi = \tau_{i_0}\tau_{i_1}\tau_{i_2}\ldots$ with initial values $\mathbf{x}_0$, t has initial value t_0. Its subsequent values are found by applying each transition in turn, where an application of a transition τ_i to t increases each node by the value of its τ_i child (simultaneously, or starting from the root). The value of $E(\mathbf{x})$ depends not only on the number of times each transition is taken, but also on the *order* that transitions are taken.

Example 4. Suppose x, y, and z of CHASE have initial values x_0, y_0, and z_0, respectively. After taking transition τ_1, the root node of Figure 2**(a)** has value $x_0 - y_0 + 1 - x_0 = 1 - y_0$, node (τ_1) has value $-x_0$, and the other nodes remain unchanged. After then taking transition τ_2, the root and (τ_1) nodes have values $1-y_0-z_0^2-z_0$ and $-x_0-z_0$, respectively. The other nodes are similarly updated.

Note that an expression may not have a finite FDT with respect to some sets of transitions. Such cases arise when the transitions have exponential behavior (*e.g.*, $x := 2x$); conversely, finite cases arise when transitions have qualitatively polynomial behavior. Intuitively, the height of a finite FDT parallels the degree (*i.e.*, linear, quadratic, *etc.*) of the polynomial behavior. In this paper, we address only the finite case — expressions in loops that evolve with qualitatively polynomial behavior.

To facilitate the analysis we define *Taylor FDTs* and *partial Taylor FDTs*, which eliminate the dependence on the order in which the transitions are taken. We then show how every finite FDT can be conservatively approximated by a partial Taylor FDT.

Definition 4 (Critical Leaves) The set of *critical leaves* $\Delta_T E(\mathbf{x})$ of a finite FDT are those nodes such that for all permutations σ, $\Delta_{\sigma(T)}E(\mathbf{x})$ has value 0 or is a leaf, and for at least one permutation σ, $\Delta_{\sigma(T)}E(\mathbf{x})$ is a leaf.

Definition 5 (Taylor FDT and Partial Taylor FDT) A finite FDT of $E(\mathbf{x})$ is a *Taylor FDT* if for each sequence of transitions T and every permutation σ of T, $\Delta_T E(\mathbf{x}) = \Delta_{\sigma(T)}E(\mathbf{x})$. That is, all n^{th} order finite differences sharing the same multiset of transitions have the same value. A finite FDT of $E(\mathbf{x})$ is a *partial Taylor FDT* if for each critical leaf $\Delta_T E(\mathbf{x})$ and permutation σ, $\Delta_T E(\mathbf{x}) = \Delta_{\sigma(T)}E(\mathbf{x})$.

Even if an FDT is not a Taylor or partial Taylor FDT, it is associated with a partial Taylor FDT.

Definition 6 (Taylored FDT) Given finite FDT t of $E(\mathbf{x})$, the *positive Taylored FDT* t^+ is a partial Taylor FDT. Each critical leaf $\Delta_T E(\mathbf{x})$ of t is given value $\max_\sigma \Delta_{\sigma(T)}E(\mathbf{x})$ in t^+; the rest of t^+ is identical to t. The *negative*

Taylored FDT t^- is similar, except that each critical leaf's value is given by $\min_\sigma \Delta_{\sigma(T)} E(\mathbf{x})$.

The definition of a positive Taylored FDT t^+ implies that the value of a node in t^+ is at least that of its counterpart in t. The opposite relation holds between t^- and t. Consequently, given a computation $\pi = \tau_{i_0}\tau_{i_1}\tau_{i_2}\ldots$, $t^- \leq t \leq t^+$ always holds, where $\leq$ expresses nodewise comparison, and thus $E(\mathbf{x})^- \leq E(\mathbf{x}) \leq E(\mathbf{x})^+$.

Example 5. The Taylored FDT of x is shown in Figure 2**(b)**. Node (τ_1, τ_2, τ_2) becomes 0 because

$$\max\{\Delta_{1,2,2}(x-y), \Delta_{2,1,2}(x-y), \Delta_{2,2,1}(x-y)\} = \max\{-1, 0, 0\} = 0.$$

For conceptual clarity, we extend the definition of a Taylored FDT so that the result is a Taylor FDT; however, the extension can only be computed with respect to the initial values of a computation.

Definition 7 (Fully Taylored FDT) Given finite FDT t of $E(\mathbf{x})$ and initial value $\mathbf{x}_0$, the *positive fully Taylored FDT* t_f^+ is a Taylor FDT. Each node n at index T in t_f^+ has value $\max_\sigma(\Delta_{\sigma(T)} E(\mathbf{x})[\mathbf{x} \mapsto \mathbf{x}_0])$. The *negative Taylored FDT* t_f^- is similar, except that each leaf's value is given by $\min_\sigma(\Delta_{\sigma(T)} E(\mathbf{x})[\mathbf{x} \mapsto \mathbf{x}_0])$.

We note that for a given initial state and computation π, always $t_f^- \leq t^- \leq t \leq t^+ \leq t_f^+$. Because the negative (fully) Taylored FDT of an expression is equivalent to the positive (fully) Taylored FDT of the negated expression, we will only consider the positive form henceforth and drop the qualifier "positive."

A fully Taylored FDT has the property that for any multiset of transitions T, all finite difference nodes $\Delta_{\sigma(T)} E(\mathbf{x})$ have the same value. Consequently, the FDT may be analyzed in a way parallel to the analysis of polynomials of multiple variables that vary continuously with time. Specifically, we look at the Taylor expansion around "time" 0 — the beginning of the computation. Since the behavior is polynomial, the Taylor expansion is exact.

Consider, for a moment, a fully Taylored FDT as expressing derivatives of $E(\mathbf{x})$ with respect to time. Then given the initial value of the computation, the Taylor series expansion is simply given by the FDT itself; *i.e.*,

$$\sum_{\Delta_T E(\mathbf{x}) \in t} \frac{\prod_{\tau \in T} x_\tau}{|T|!} \Delta_T E(\mathbf{x})[\mathbf{x} \mapsto \mathbf{x}_0],$$

viewing the Ts as lists or multisets. In the discrete context, the expansion is slightly different; however, the dominant terms are the same for the continuous and discrete expansions. Moreover, the coefficients of the dominant terms are those of the critical leaves, which are either constants or constant expressions. In some cases, constant expressions may be soundly approximated by constants, taking care that if a constant expression can possibly be 0, other terms in the expansion dominate. Then for a partial Taylor FDT, the dominant terms comprise the *dominant Taylor expression*.

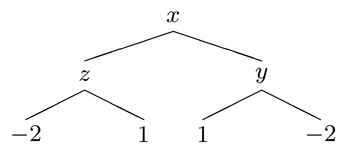
Definition 8 (Dominant Taylor Expression) Given finite partial Taylor FDT t, its *dominant Taylor expression* is

$$\sum_{\Delta_T E(\mathbf{x}) \in critical\_leaves(t)} \frac{\prod_{\tau \in T} x_\tau}{|T|!} \Delta_T E(\mathbf{x}),$$

where one variable x_τ is introduced per transition τ, representing the number of times the transition has been taken.

while $x \geq 0$ **do**
  $\tau_1 :\ (x,\ y,\ z) := (x + z,\ y + 1,\ z - 2)$
    or
  $\tau_2 :\ (x,\ y) := (x + y,\ y - 2)$
od

(a)



(b)

Fig. 3. **(a)** MPP INTERACTION. **(b)** Taylored FDT.

Example 6. For CHASE, the dominant Taylor expression of the Taylored FDT for the expression $x - y$ is

$$-1\frac{x_1^2}{2!} - 2\frac{x_2^3}{3!} = \frac{-x_1^2}{2} - \frac{x_2^3}{3},$$

where x_1 and x_2 express the number of times that transitions τ_1 and τ_2 are taken, respectively. Conceptually, we may consider a new MPP in which the nonnegativity of the dominant Taylor expression is the guard, x_i are the variables, and each transition τ_i in the original MPP corresponds to a transition $x_i := x_i + 1$ in the new MPP. Clearly, the value of the new guard at any point in a computation depends only on the number of times each transition has been taken.

Example 7. Consider MPP INTERACTION in Figure 3**(a)** and the Taylored FDT for x with respect to $\{\tau_1, \tau_2\}$ in Figure 3**(b)**. The dominant Taylor expression of the Taylored FDT for the expression x is

$$-2 \cdot \frac{1}{2!}x_1x_1 + \frac{1}{2!}x_1x_2 + \frac{1}{2!}x_2x_1 - 2 \cdot \frac{1}{2!}x_2x_2 = -x_1^2 + x_1x_2 - x_2^2.$$

Note the nonnegative term x_1x_2, indicating the adverse interaction of τ_1 and τ_2.

Combining the result $t_f^- \leq t^- \leq t \leq t^+ \leq t_f^+$ with the dominant Taylor expression admits analysis of the evolution of $E(\mathbf{x})$. In the next section, we show how to use the dominant Taylor expression of t^+ to discover if $E(\mathbf{x})$ eventually decreases beyond any bound on all computations, which leads naturally into proofs of termination.

4 FDTs and Termination

In the last section, we developed a theory of finite differences for transition systems involving polynomial expressions and assignments. The conclusion hinted at the intuition for a termination analysis. If the dominant Taylor expression of the Taylored FDT of $E(\mathbf{x})$ with respect to the transitions $\mathcal{T}$ decreases without bound on all computations then, first, $E(\mathbf{x})^+_f$ from the fully Taylored FDT must decrease without bound so, second, as $E(\mathbf{x}) \le E(\mathbf{x})^+ \le E(\mathbf{x})^+_f$, $E(\mathbf{x})$ must also decrease without bound. If continuation of the loop depends on $E(\mathbf{x})\ \{\ge, >\}\ 0$, then the loop must terminate on all input. In this section, we formalize this description and analyze several conditions on a MPP guard's FDTs that ensure termination.

4.1 Single Loop Condition

For the case of one loop condition $P(\mathbf{x})\ \{\ge, >\}\ 0$, we consider the FDT of $P(\mathbf{x})$ with respect to the loop's assignment transitions $\mathcal{T}$.

Proposition 1. (Taylor Condition) *Suppose that for each nonempty $\mathcal{T}' \subseteq \mathcal{T}$,*

1. *the FDT t of $P(\mathbf{x})$ with respect to $\mathcal{T}'$ is finite;*
2. *the dominant Taylor expression of the Taylored FDT t^+ decreases without bound as the length of the computation increases.*

Then the loop terminates on all input.

Proof. Each subset $\mathcal{T}'$ represents a possible set of transitions that are taken infinitely often. Consider one such set. Suppose the dominant Taylor expression of t^+ with respect to $\mathcal{T}'$ decreases without bound as the length of the computation increases. For all initial values, the dominant Taylor expression dominates the Taylor expansion of the root of the fully Taylored FDT; therefore, $P(\mathbf{x})^+_f$ decreases without bound. But $P(\mathbf{x})^+_f \ge P(\mathbf{x})^+ \ge P(\mathbf{x})$, so $P(\mathbf{x})$ also decreases without bound. Since this conclusion holds for all $\mathcal{T}'$, the loop must terminate.

Since the FDTs we consider have finite depth, if a dominant Taylor expression eventually only decreases, then it eventually decreases without bound. We refer to polynomials that satisfy the assumption of the proposition as *decreasing* with respect to $\mathcal{T}$.

Example 8. For CHASE, the dominant Taylor expressions for x with respect to $\{\tau_1\}$, $\{\tau_2\}$, and $\{\tau_1, \tau_2\}$ are the following, respectively:

$$\frac{-x_1^2}{2}, \quad \frac{-x_2^3}{3}, \quad \text{and} \quad \frac{-x_1^2}{2} - \frac{x_2^3}{3}.$$

The last expression was calculated in Example 6. All three expressions clearly decrease without bound as the length of the computation increases. Thus, CHASE terminates on all input.

The following example introduces a technique for showing that a more complicated dominant Taylor expression is decreasing. Changing to polar coordinates allows the length of a computation to appear explicitly in the dominant Taylor expression.

Example 9. Recall that the dominant Taylor expression for x with respect to $\{\tau_1, \tau_2\}$ in INTERACTION is $-x_1^2 + x_1 x_2 - x_2^2$. Call the expression $\sqrt{x_1^2 + x_2^2}$ the *absolute length* of a computation (in which both transitions are taken infinitely often). Since x_1 and x_2 express the number of times τ_1 and τ_2 have been taken, the absolute length is initially 0 and grows with each iteration. If the dominant Taylor expression decreases without bound as the absolute length of the computation increases, then the assumption of the Taylor condition is satisfied for each of τ_1 and τ_2 occurring infinitely often.

Let $x_1 = r\cos\theta$ and $x_2 = r\sin\theta$. r corresponds to the absolute length, while $\theta \in [0, \frac{\pi}{2}]$ expresses the ratio of x_2 to x_1. Then after a change of variables,

$$-x_1^2 + x_1 x_2 - x_2^2 = -r^2\cos^2\theta + r^2\cos\theta\sin\theta - r^2\sin^2\theta = r^2(\cos\theta\sin\theta - 1).$$

Call this expression $Q(r, \theta)$. Differentiating, we find

$$\frac{\partial Q}{\partial r} = 2r(\cos\theta\sin\theta - 1) \quad \text{and} \quad \frac{\partial^2 Q}{\partial r^2} = 2(\cos\theta\sin\theta - 1).$$

The relevant domain of θ is $[0, \frac{\pi}{2}]$, over which the maximum of $\frac{\partial^2 Q}{\partial r^2}$ is -1, occurring at $\theta = \frac{\pi}{4}$. Therefore, independent of θ, as r increases, $Q(r, \theta)$ eventually decreases without bound; therefore, the dominant Taylor expression also eventually decreases without bound.

Finally, considering the case where only τ_1 (τ_2) is taken after a certain point, we note that the dominant Taylor expression is $-x_1^2$ ($-x_2^2$), which decreases without bound. Thus, INTERACTION terminates on all input.

We can apply the trick of using the absolute length and changing to polar coordinates in general, via the usual extension of polar coordinates to higher dimensions. For m transitions and expression $Q(r, \theta_1, \ldots, \theta_{m-1})$, we check if $\frac{\partial^n Q}{\partial r^n}$ is everywhere at most some negative constant over $\theta_i \in [0, \frac{\pi}{2}]$, $i \in [1..m-1]$, where $\frac{\partial^n Q}{\partial r^n}$ is the first derivative with respect to r that is constant with respect to r.

In many cases, a weaker condition on the structure of the single FDT with respect to all transitions $\mathcal{T}$ is sufficient for proving termination, precluding an expensive analysis of the dominant Taylor expression for each subset of transitions. The condition follows from the Taylor condition, although it is intuitive by itself.

Proposition 2. (Standard Condition) *If every leaf of the FDT t of $P(\mathbf{x})$ with respect to $\mathcal{T}$ is negative, and the root of t has $|\mathcal{T}|$ children, then the loop terminates on all input.*

Example 10. The Taylored FDT for CHASE in Figure 2**(b)** meets this condition, proving termination, while the Taylored FDT for INTERACTION in Figure 3**(b)** does not.

4.2 General Loop Condition

Consider now the loop condition $\bigwedge_j P_j(\mathbf{x}) \ \{\geq, >\}\ 0$. Clearly, one simple condition is that at least one conjunct's FDT decreases without bounds; however, a stronger condition based on a lexical ordering is possible. The following definition will be useful for specifying the condition.

Definition 9 (Neutral Transitions) A set of transitions $\mathcal{T}_n \subseteq \mathcal{T}$ is *neutral* toward an expression $E(\mathbf{x})$ decreasing with respect to transitions $\mathcal{T} \backslash \mathcal{T}_n$ if $E(\mathbf{x})$ is decreasing with respect to $\mathcal{T}$, except possibly when only transitions in $\mathcal{T}_n$ are taken infinitely often.

Checking if a set of transitions is neutral merely requires excluding certain subsets of transitions (those that contain only transitions that need only be neutral) when analyzing termination. For the standard condition, if transition τ_i is neutral, the root need not have a τ_i child.

$$\begin{array}{l}
\textbf{while } x \geq 0 \wedge z^3 \geq y \textbf{ do} \\
\quad \tau_1 : \ (x,\ y) := (x-1,\ y-1) \\
\qquad\quad \textbf{or} \\
\quad \tau_2 : \ (y,\ z) := (y-1,\ z+y) \\
\textbf{od}
\end{array}$$

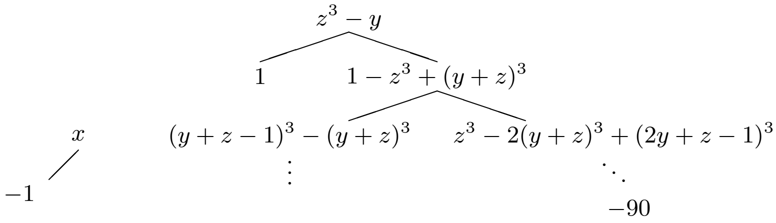
Fig. 4. Program CONJUNCT.

Example 11. Consider program CONJUNCT in Figure 4. The FDT of x with respect to $\{\tau_1, \tau_2\}$ is shown in Figure 5**(a)**. Transition $\{\tau_2\}$ is neutral toward x, which decreases with respect to $\{\tau_1\}$. Its dominant Taylor expression is $-x_1$, which decreases without bound unless τ_1 is not taken after a certain iteration, regardless of how frequently τ_2 is taken.

Proposition 3. (Conjunction Condition) *Consider the loop with transitions $\mathcal{T}$, a conjunction of n loop conditions $P_j(\mathbf{x}) \ \{\geq, >\}\ 0$, and a map $\mu : \mathcal{T} \mapsto [1..n]$ mapping transitions to conjuncts. Then the loop terminates on all input if for each j, the set $\{\tau \mid \mu(\tau) > j\}$ is neutral toward $P_j(\mathbf{x})$, which decreases with respect to $\{\tau \mid \mu(\tau) = j\}$.*

Proof. Suppose the assumption holds, yet the computation σ is nonterminating. Let $\mathcal{T}_\infty$ be the set of transitions occurring infinitely often and $\mathcal{T}_{\min} = \{\tau \in \mathcal{T}_\infty \mid \mu(\tau) = \min_{\tau' \in \mathcal{T}_\infty} \mu(\tau')\}$ be the set of $\mathcal{T}_\infty$-transitions mapping to the loop condition with the lowest index, j. Following the assumption, $P_j(\mathbf{x})$ is decreasing with respect to $\mathcal{T}_{\min}$, while $\mathcal{T}_\infty \backslash \mathcal{T}_{\min}$ is neutral toward $P_j(\mathbf{x})$. Thus, $P_j(\mathbf{x})$ is decreasing with respect to $\mathcal{T}_\infty$, and $P_j(\mathbf{x}) \ \{\geq, >\}\ 0$ is violated in a finite number of steps, a contradiction.

For the most general loop condition $\bigvee_i \bigwedge_j P_{ij}(\mathbf{x}) \ \{\geq, >\}\ 0$, each disjunct must satisfy the conjunction condition. Of course, either the Taylor condition or the standard condition may be used to determine whether an expression is decreasing with respect to a certain set of transitions or if a set of transitions is neutral toward an expression.



(a) **(b)**

Fig. 5. FDTs of **(a)** x and **(b)** $z^3 - y$ with respect to the transitions of program CONJUNCT.

Example 12. Consider program CONJUNCT with index order $P_1(x, y, z) = x$, $P_2(x, y, z) = z^3 - y$, and map μ such that $\mu(\tau_1) = 1$, $\mu(\tau_2) = 2$. τ_2 is neutral toward x, while τ_1 is not neutral toward $z^3 - y$, as suggested by the FDT for $z^3 - y$ in Figure 5**(b)**. Nonetheless, the conjunction condition holds, so CONJUNCT terminates on all input.

Specifically, for $P_1(x, y, z) = x$, we need only consider the subsets $\{\tau_1, \tau_2\}$ and $\{\tau_1\}$, each of which result in the decreasing dominant Taylor expression $-x_1$. For $P_2(x, y, z) = z^3 - y$, only the subset $\{\tau_2\}$ must be considered, for which the dominant Taylor expression $-\frac{90}{6!}x_2^6$ is decreasing.

Using the standard condition, we merely note that the FDT for x with respect to $\{\tau_1, \tau_2\}$ shows that $\{\tau_2\}$ is neutral toward x (no τ_2 child of the root), while x decreases with respect to $\{\tau_1\}$ (-1 leaf). Further, the rightmost branch in Figure 5**(b)**, which is the FDT of $z^3 - y$ with respect to $\{\tau_2\}$, terminates with -90, also satisfying the standard condition.

4.3 Guarded Commands

Instead of an MPP, consider a set of *polynomial guarded commands* $\mathcal{C} = \{G_1 \to S_1, \ldots, G_n \to S_n\}$, for which the G_i are conjunctions of polynomial inequations and the S_i are polynomial assignments. An initial condition θ may be associated with the set of guarded commands.

Proposition 4. (Guarded Commands Condition) *Consider the set of guarded commands $\mathcal{C} = \{G_1 \to S_1, \ldots, G_n \to S_n\}$. $\mathcal{C}$ always terminates if there exists a permutation σ such that for each i, there exists a conjunct (e $\{\geq, >\}$ 0) of G_i such that e is decreasing with respect to $\{S_i\}$ and to which $\{S_j \mid \sigma(j) > \sigma(i)\}$ is neutral.*

Briefly, the first guarded command given by σ can only be executed a finite number of times before its guard is violated; as the remaining commands are neutral toward G_1, it is henceforth violated. The same reasoning applies to the second command once the first command is disabled. The disabling of the remaining commands follows by induction.

The language of sets of polynomial guarded commands is more expressive than the language of MPPs — indeed, our practical experience (see Section 5) supports the guarded command abstraction as the more useful. However, the relationship between the general loop condition of Section 4.2 and Proposition 4 is incomparable. Given an MPP and its natural translation to a set of polynomial guarded commands, if Proposition 4 proves termination, then applying Proposition 3 to each disjunct of the MPP's guard also proves termination (extract the lexicographic orders for the latter from the single lexicographic order of the former). However, if, for example, the MPP has two disjuncts requiring opposite orders for Proposition 3, then no interpolation produces a suitable order for Proposition 4. Of course, Proposition 4 is more applicable, in some sense, because of the extra expressiveness of the guarded command abstraction. Allowing disjunction in the guards of the guarded commands makes the resulting guarded command abstraction and the natural termination condition strictly more powerful, but we have not found this additional power useful.

5 Experimental Results

To test the applicability of our termination analysis, we implemented a C loop abstracter in CIL [8] and the termination analysis in Mathematica [15]. The purpose of the loop abstracter is to extract a set of polynomial guarded commands from a C loop with arbitrary control flow, including embedded loops. The analysis then applies to the extracted guarded commands a version of Proposition 4 that exploits the standard condition. The implementation is weaker than Proposition 4, in that it requires for each i that all other assignments $S_j \neq S_i$ are neutral toward the expression e from G_i, rather than allowing a lexicographic ordering. The analysis is sound up to alias analysis, modification of variables by called functions, and unsigned casts. We chose to ignore these factors in our experimentation, as they have no bearing on the scalability of the actual termination analysis. Handling unsigned casts, for example, would require proving invariants about signed integers; a complete program analysis package would contain this functionality.

Given a loop, the abstraction first creates a *number abstraction* by slicing on the number typed variables that receive values within the loop from polynomial expressions. Division is allowed for floats, but not for integers; further, an integer cast of an expression with a floating point value excludes the expression from consideration. Nondeterministic choice replaces disallowed expressions. Next, the abstraction constructs all possible top-level guarded paths; variables that are modified by embedded loops are set nondeterministically (a heavy-handed version of summarizing embedded loops). The construction of a guarded path proceeds by the usual composition of assignments, so that the final guarded path consists of a conjunction of guard expressions and a single concurrent update to a set of variables. The result is a set of guarded commands, as described in Section 4.3. Our Mathematica implementation of the standard condition then analyzes this set, failing if an FDT reaches a predetermined maximum depth during construction.

```
while(i < a.n || j < b.n) {
  if (i >= a.n)
    c.e[c.n++] = b.e[j++];
  else if (j >= b.n)
    c.e[c.n++] = a.e[i++];
  else if (a.e[i] <= b.e[j])
    c.e[c.n++] = a.e[i++];
  else
    c.e[c.n++] = b.e[j++];
}
```

(a)

$$
\begin{array}{l}
j < b.n \wedge i < a.n \;\rightarrow\; (c.n, j) := (c.n+1, j+1) \\
j < b.n \wedge i < a.n \;\rightarrow\; (c.n, i) := (c.n+1, i+1) \\
j \geq b.n \wedge i < a.n \;\rightarrow\; (c.n, i) := (c.n+1, i+1) \\
j < b.n \wedge i \geq a.n \;\rightarrow\; (c.n, j) := (c.n+1, j+1)
\end{array}
$$

(b)

Fig. 6. (a) Imperative loop in C and **(b)** the corresponding set of polynomial guarded commands.

Example 13. The loop in Figure 6**(a)** merges two lists. The abstracted set of guarded commands is shown in Figure 6**(b)**; four guarded commands with false guards were pruned. The analysis proves that the loop terminates.

5.1 Empirical Results

We applied the analysis to several open-source projects from Netlib [9] and Sourceforge [12]. The results of the analyses are summarized in Table 1. These programs span a range of applications: for example, `f2c` converts FORTRAN source to C source; `spin` is a model checker; and `meschach` is a package of numerical algorithms.

5.2 Analysis

A glance over the loops in the programs suggests that when the number abstraction of a C loop is proved terminating, the reason is probably because of a counter. In some sense, this observation is disappointing: of what value is our analysis when the reasons are trivial? Three points come to mind. First, applying any analysis at all is useful. Programmers regularly write loops with complicated control structure that span several editor pages. Verifying manually that all paths increment a counter (and the right counter) is thus tedious and ineffective. An automated analysis filters out correct cases, while remaining loops warrant a second look.

Name	LOC	# L	# A	# P	% P/A	% P/L	Time (s)
small1	310	8	6	4	66	50	4
vector	361	13	13	12	92	92	3
serv	457	9	6	5	83	55	4
dcg	1K	55	53	53	100	96	4
bcc	4K	70	18	18	100	25	6
sarg	7K	122	26	25	96	20	102
spin	19K	652	132	119	90	18	29
meschach	28K	896	803	770	95	85	40
f2c	30K	434	114	96	84	22	41
ffmpeg/libavformat	33K	453	270	214	79	47	45
gnuplot	50K	825	329	298	90	36	106
gaim	57K	605	60	52	86	8	97
ffmpeg/libavcodec	75K	2216	1945	1856	95	83	112

Table 1. Results of analysis. Legend: **LOC**: lines of code *of files successfully parsed and containing loops*, as measured by `wc`; **# L**: total number of analyzed loops; **# A**: number of loops successfully abstracted; **# P**: number of (abstracted) loops proved terminating; **% P/A**: percentage of abstracted loops proved terminating; **% P/L**: percentage of total loops proved terminating; **Time**: total time in seconds required to analyze the program. `small1` requires a maximum FDT height of 4; data for all others are for a maximum height of 1.

Second, our analysis scales well to triviality: the FDTs are shallow (of depth one for the counter case), thus requiring an insignificant amount of time. Compare the minimal computation required for the standard condition on a shallow FDT to the manipulation of polyhedra [2, 3], the solving of linear programs [11], or the analysis of matrix-like transitions [13] (assuming that the latter two approaches can scale to multiple paths). However, the extra power of the nonlinear analysis is available when needed.

Finally, compared to a naive syntactic analysis, our approach has two advantages. First, a naive syntactic analysis would be sensitive to the presentation of the loop. For example, a syntactic analysis may well stumble on a `while` loop that terminates using `break` or `goto` statements. Our abstraction and analysis approach not only is insensitive to such presentations of loops, it may also identify other loop guards than the one explicitly provided by the `for` or `while` statement. Second, even trivial termination behavior is not always completely trivial. For example, the `meschach` source contains loops with terminating behavior similar to that in Figure 6. Our analysis easily handles such cases. Additionally, despite the prototype-related overheads of our implementation, the timing results indicate acceptable performance.

Reasons for failed proofs are numerous. A failure to abstract a nontrivial guarded set may indicate nontermination (especially if, say, all transitions but one increment a counter), but usually arises because the termination behavior is not number-related. Even "successful" abstractions may present only incidental information; termination may rest on other criteria. In several cases, we noted that the lack of an initial condition weakens the abstraction. Other cases played on the weaknesses of the analysis, including the following: (1) expressions evolve with exponential behavior, resulting in infinite FDTs; (2) variables are modified by inner loops, often in a way that trivially suggests an inequality relation.

The number abstraction may be extended beyond pure numbers. Many loops are based on iterating through collection data structures, such as linked lists and heaps. A sophisticated analysis tool would allow the user to input information about such data structures, allowing a number abstraction of iteration. The resulting termination analysis would be sound relative to the correctness of the data structure. Widely used implementations of data structures, such as those provided by the STL, are candidates for automatic analysis.

6 Conclusion

Multipath polynomial programs and polynomial guarded commands provide an expressive language for abstracting real code. Although termination for this class of loops is not even semi-decidable, we provide a sound analysis that is effective in practice. This analysis is notable for two reasons. First, it is applicable to polynomial, rather than just linear, expressions and assignments. Second, our analysis naturally scales to the difficulty of the problem, which enables our prototype implementation to analyze tens of thousands of lines of C in seconds.

The analysis can be strengthened in several ways. First, *head* and *tail loops*, or embedded loops that precede or follow, respectively, all assignments in the top-level loop, may be abstracted to form a set of paths to include as *neutral* top level paths. Second, analysis of the code preceding loop entry, or even invariant generation, can supply initial conditions. Third, embedded loops that modify some variables may sometimes be abstracted as transition relations with inequations. Extending both the abstraction and the analysis to handle such inequations would increase our method's applicability. Fourth, the analysis may be extended to handle FDTs with infinite or large finite height by arbitrarily curtailing FDT construction and using invariant analysis to provide useful bounds on the resulting leaves. Fifth, the abstraction may be extended to iterating over data structures. Finally, we plan to employ the analysis within a larger C analysis that exploits alias information, thus providing a path toward a sound implementation.

Acknowledgments We thank the reviewers for their insightful comments and suggestions. Additionally, we gratefully acknowledge the contribution of George Necula and the other developers of CIL to the software analysis community.

References

1. Cohen, J. Computer-assisted microanalysis of programs. *Comm. ACM 25*, 10 (1982).
2. Colón, M., and Sipma, H. Synthesis of linear ranking functions. In *TACAS* (2001).

3. Colón, M., and Sipma, H. Practical methods for proving program termination. In *CAV* (2002).
4. H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. In *CAV* (1996).
5. Katz, S., and Manna, Z. Logical analysis of programs. *Comm. ACM 19*, 4 (1976).
6. Lee, C. S., Jones, N. D., and Ben-Amram, A. M. The size-change principle for program termination. In *POPL* (2001).
7. Manna, Z., Browne, A., Sipma, H., and Uribe, T. E. Visual abstractions for temporal verification. In *Algebraic Methodology and Software Technology* (1998).
8. Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Conf. on Compiler Construction* (2002).
9. *Netlib Repository*, 2004. (`http://www.netlib.org`).
10. Podelski, A., and Rybalchenko, A. Software model checking of liveness properties via transition invariants. Technical Report, MPI für Informatik, 2003.
11. Podelski, A., and Rybalchenko, A. A complete method for the synthesis of linear ranking functions. In *VMCAI* (2004).
12. *SourceForge*, 2004. (`http://sourceforge.net`).
13. Tiwari, A. Termination of linear programs. In *CAV* (2004).
14. Wegbreit, B. Mechanical program analysis. *Comm. ACM 18*, 9 (1975).
15. Wolfram Research, Inc. *Mathematica, Version 5.0.* Champaign, IL, 2004.