Termination Analysis without the Tears

Shaowei Zhu
shaoweiz@cs.princeton.edu
Princeton University
Princeton, NJ, USA

Zachary Kincaid
zkincaid@cs.princeton.edu
Princeton University
Princeton, NJ, USA

Abstract

Determining whether a given program terminates is the quintessential undecidable problem. Algorithms for termination analysis may be classified into two groups: (1) algorithms with strong behavioral guarantees that work in limited circumstances (e.g., complete synthesis of linear ranking functions for polyhedral loops), and (2) algorithms that are widely applicable, but have weak behavioral guarantees (e.g., Terminator). This paper investigates the space in between: *how can we design practical termination analyzers with useful behavioral guarantees?*

This paper presents a termination analysis that is both *compositional* (the result of analyzing a composite program is a function of the analysis results of its components) and *monotone* ("more information into the analysis yields more information out"). The paper has two key contributions. The first is an extension of Tarjan's method for solving path problems in graphs to solve *infinite* path problems. This provides a foundation upon which to build compositional termination analyses. The second is a collection of monotone conditional termination analyses based on this framework. We demonstrate that our tool ComPACT (Compositional and Predictable Analysis for Conditional Termination) is competitive with state-of-the-art termination tools while providing stronger behavioral guarantees.

CCS Concepts: • **Theory of computation** → **Program analysis**; **Regular languages**; • **Software and its engineering** → **Automated static analysis**.

Keywords: Algebraic program analysis, termination analysis, loop summarization, algebraic path problems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8391-2/21/06...$15.00
https://doi.org/10.1145/3453483.3454110

ACM Reference Format:
Shaowei Zhu and Zachary Kincaid. 2021. Termination Analysis without the Tears. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454110

1 Introduction

Termination is an important correctness property in itself, and is a sub-problem of proving total correctness, liveness properties [12, 15, 17–19], and bounds on resource usage [1, 13, 32, 33, 43]. Determining whether a program terminates is undecidable, and so progress on automated tools for termination analysis is driven by heuristic reasoning techniques. While these heuristics are often effective in practice, they can be brittle and unpredictable. For example, termination analyzers may themselves fail to terminate on some input programs, or report false alarms, or return different results for the same input, or suffer from "butterfly effects", in which a small changes to the program's source code drastically changes the analysis.

This paper is motivated by the principle that *changes to a program should have a predictable impact on its analysis*. We develop a style of termination analysis that achieves two particular desiderata:

- *Compositionality*: composite programs are analyzed by analyzing their sub-components and then combining the results. Compositionality implies that changing part of a program only changes the analysis of that part. It enables prompt user interaction, since an analysis need not reanalyze the whole program to respond to a program change.
- *Monotonicity*: more information into the analysis yields more information out. Monotonicity implies that certain actions, e.g., a user annotating a procedure with additional pre-conditions, or an abstract interpreter instrumenting loop invariants into a program, *cannot* degrade analysis results.

Our approach is based on the paradigm of *algebraic program analysis* [26, 45, 46]. An algebraic program analysis is described by an algebraic structure in which the elements represent properties of finite program executions and the operations compose those properties via sequencing, choice, and iteration (mirroring the structure of regular expressions). To verify a safety property, an algebraic program analyzer computes a regular expression recognizing all paths through

a program to a point of interest, interprets the regular expression within the given structure, and checks whether the resulting property entails the property of interest.

In this paper, we extend the algebraic approach to reason about *infinite* program paths, and thereby provide a conceptual and algorithmic basis for compositional analysis of liveness properties such as termination. Conceptually, our method proves that a program terminates by computing a transition formula for each loop that over-approximates the behavior of its body, and then proving that the corresponding relation admits no infinite sequences. (Our approach is not unique in this regard (see Section 8)—we provide a unifying foundation for such analyses).

A drawback of using summaries to prove termination is that the loop body summary *over-approximates* its behavior, and so the summary may not terminate even if the original loop does. The advantage is that we can reason about the summary effectively, whereas any non-trivial question about behavior of the original loop is undecidable. This is the key idea that enables the design of *monotone* termination analyses.

A particular challenge of compositional termination analysis is that termination arguments must be synthesized independently of the surrounding context of the loop (that is, without supporting invariants). We meet this challenge with a set of methods that exploit loop summarization to generate monotone *conditional* termination arguments. These methods synthesize both a termination argument and an initial condition under which that argument holds, with the latter acting as a surrogate for a supporting invariant.

Contributions. The contributions of this paper are:

- A framework for designing compositional analyses of infinite program paths. This framework extends Tarjan's method for solving path problems [45, 46] from finite to infinite paths.
- An efficient algorithm for computing an ω-regular expression that recognizes the infinite paths through a control flow graph, which forms the algorithmic foundation of our program analysis framework.
- The first termination analysis that is compositional, monotone, and applies to a general program model. We present a set of combinators for constructing a family of such (conditional) termination analyses based on our framework. In particular, we introduce *phase analysis*, which improves the precision of a given conditional termination analysis by partitioning the space of transitions in a loop into phases.

2 Overview

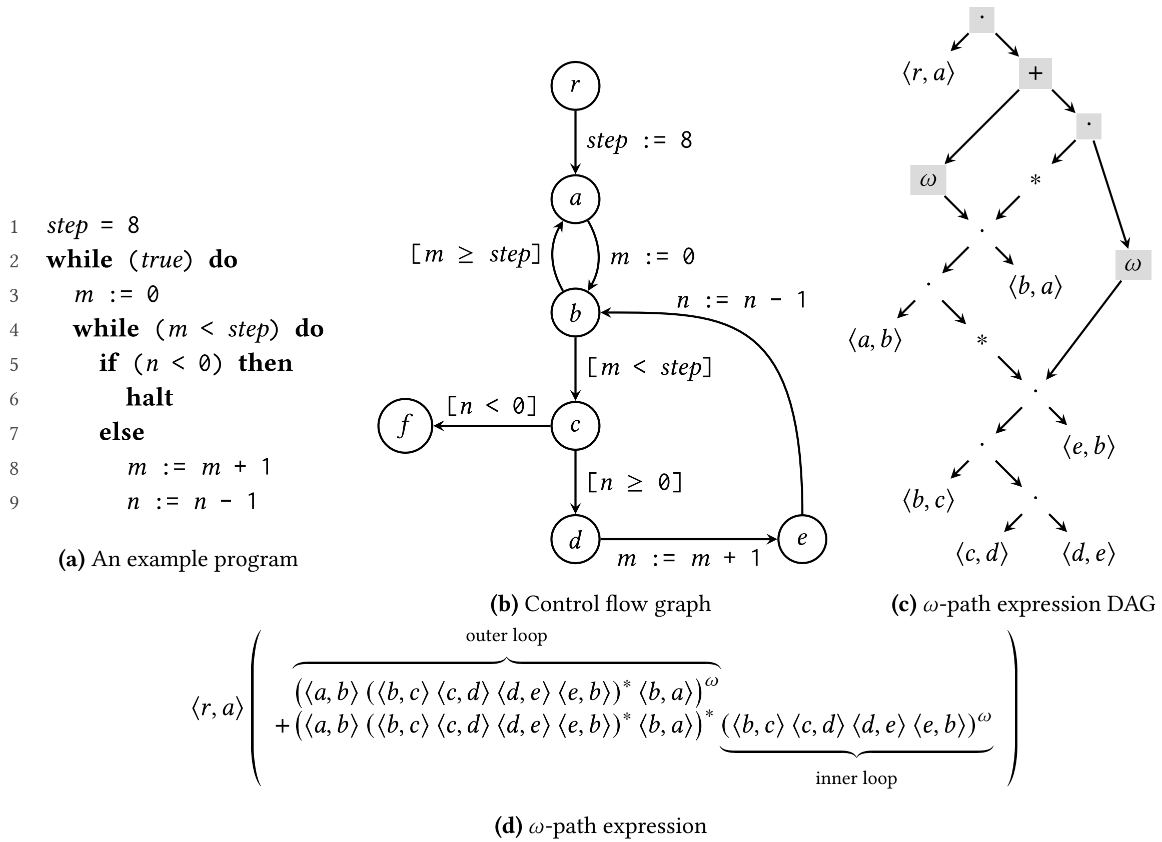
In Section 5, we define an algebraic framework for analyzing liveness properties of programs. An analysis proceeds in two steps: (1) compute an ω-regular expression that recognizes the paths through a program, and (2) interpret that ω-regular expression with an algebraic structure corresponding to a program analysis of interest.

We illustrate this process in Figure 1. Consider the example program given by its control flow graph (CFG) in Figure 1b (concrete syntax for the CFG is given in Figure 1a). Note that conditional control flow is encoded as *assumptions*, which do not change the program variables but can only be executed if the assumed condition holds (e.g., if the program is in a state where m is less than *step* then it may execute the assumption [m < *step*], otherwise it is blocked).

Step 1: Compute an ω-path expression. Using the algorithm described in Section 4, we can compute an ω-regular expression that represents all infinite paths in the CFG that begin at the entry vertex r (Figure 1d). The expression can be represented efficiently as a directed acyclic graph (DAG), where each leaf is labeled by a control flow edge, each internal node with an operator (one of: choice (+), concatenation ($\cdot$), iteration ($*$), or infinite repetition (ω)—see Section 3.1), and edges are drawn from operators to operands (Figure 1c). Observe that each node in the DAG corresponds to either a regular expression (white nodes) or an ω-regular expression (gray nodes).

Step 2: Interpretation. The result of a particular analysis is computed by interpreting a path expression for a program within some abstract domain. The domain consists of (1) a *regular algebra*, which is equipped with choice, concatenation, and iteration operators and which can be used to interpret regular expressions, and (2) an *ω-regular algebra*, which is equipped with choice, concatenation, and ω-iteration operators, and which can be used to interpret ω-regular expressions.

Our main interest in this paper is in a family of termination analyses. In this family, the regular algebra is the algebra of *transition formulas*, which we denote by **TF**. A *transition formula* is a logical formula over the variables of the program (in Figure 1: m, n, *step*) along with primed copies (m', n', *step*$'$) representing the program variables before and after executing a computation, respectively. The choice operation for **TF** is disjunction, concatenation is relational composition, and iteration over-approximates reflexive transitive closure (a particular iteration operator is defined in Section 3). The ω-regular algebra is an algebra of *mortal preconditions*, which we denote by **MP**; in fact, we will define several such algebras in this paper, but they share a common structure. A mortal precondition is a state formula (over the program variables (m, n, *step*)) that is satisfied only by *mortal states*, from which the program must terminate. The choice operation for **MP** is conjunction (a mortal state must be mortal on *all* paths), concatenation is weakest precondition (a state is mortal only if it can reach only mortal states), and ω-iteration computes a mortal precondition for a transition formula (we will define several mortal precondition operators in Section 6). We compute a mortal precondition for a program by traversing



(a) An example program

(b) Control flow graph

(c) ω-path expression DAG

$$\langle r,a\rangle\left(\overbrace{\big(\langle a,b\rangle\,(\langle b,c\rangle\,\langle c,d\rangle\,\langle d,e\rangle\,\langle e,b\rangle)^*\,\langle b,a\rangle\big)^{\omega}}^{\text{outer loop}} + \big(\langle a,b\rangle\,(\langle b,c\rangle\,\langle c,d\rangle\,\langle d,e\rangle\,\langle e,b\rangle)^*\,\langle b,a\rangle\big)^{*}\underbrace{(\langle b,c\rangle\,\langle c,d\rangle\,\langle d,e\rangle\,\langle e,b\rangle)^{\omega}}_{\text{inner loop}}\right)$$

(d) ω-path expression

Figure 1. An example program, its control flow graph, and a corresponding ω-path expression

its ω-path expression DAG from the bottom up, using **TF** to interpret regular expression operators and **MP** to interpret ω-regular expression operators.

We illustrate a selection of the interpretation steps. We use $\mathcal{T}[\![-]\!]$ and $\mathcal{T}^{\omega}[\![-]\!]$ to denote the interpretation of a regular and ω-regular expression, respectively. For the leaves of the path expression DAG, we may simply encode the meaning of the corresponding program command into logic; e.g., the transition formulas for the edges $\langle c,d\rangle$ and $\langle d,e\rangle$ (corresponding to the commands `[n ≥ 0]` and `m := m + 1`, resp.) are:

$$\mathcal{T}[\![\langle c,d\rangle]\!] = n \geq 0 \land m' = m \land n' = n \land step' = step$$
$$\mathcal{T}[\![\langle d,e\rangle]\!] = m' = m+1 \land n' = n \land step' = step$$

Proceeding up the DAG, we compute a transition formula for the regular expression $\langle c,d\rangle\,\langle d,e\rangle$ by taking the relational composition of $\mathcal{T}[\![\langle c,d\rangle]\!]$ and $\mathcal{T}[\![\langle d,e\rangle]\!]$

$$\begin{aligned}\mathcal{T}[\![\langle c,d\rangle\,\langle d,e\rangle]\!] &= \mathcal{T}[\![\langle c,d\rangle]\!] \circ \mathcal{T}[\![\langle d,e\rangle]\!] \\ &\equiv \begin{array}{c} n \geq 0 \\ \land m' = m+1 \land n' = n \land step' = step\end{array}\end{aligned}$$

Similarly, we sequence $\mathcal{T}[\![\langle c,d\rangle\,\langle d,e\rangle]\!]$ with $\mathcal{T}[\![\langle b,c\rangle]\!]$ on the left and $\mathcal{T}[\![\langle e,b\rangle]\!]$ on the right to get a summary for the body of the inner loop $inner \triangleq \langle b,c\rangle\,\langle c,d\rangle\,\langle d,e\rangle\,\langle e,b\rangle$:

$$\mathcal{T}[\![inner]\!] \equiv \begin{array}{c} m < step \land n \geq 0 \\ \land m' = m+1 \land n' = n-1 \land step' = step\end{array}$$

The *inner* node has two parents, corresponding to $inner^*$ and $inner^{\omega}$. For the first, we over-approximate the transitive closure of the formula $\mathcal{T}[\![inner]\!]$:

$$\mathcal{T}[\![inner^*]\!] \equiv \exists k. \left(\begin{array}{c}\left(\begin{array}{c} k = 0 \\ \lor\left(\begin{array}{c} k \geq 1 \land m < step \land n \geq 0 \\ \land m' \leq step \land n' \geq -1\end{array}\right)\end{array}\right) \\ \land m' = m+k \land n' = n-k \land step' = step\end{array}\right)$$

(In the above formula, the existentially quantified variable k represents the number of times the loop is taken. The first conjunct encodes that if the loop is taken at least once, then its guard must hold in the initial state, and the post-image of its guard must hold in the final state. The second conjunct encodes that m increases by 1 at each iteration, n decreases by 1, and *step* is constant. See Section 3.3 for details on how we compute the transitive closure of any transition formula.)

For the second parent, $inner^{\omega}$, we compute a mortal precondition for the formula $\mathcal{T}[\![inner]\!]$. Observing that $(step - m)$

is a ranking function for this loop (i.e., the difference between *step* and m is non-negative and decreasing), we may simply take $\mathcal{T}^{\omega}[\![inner^{\omega}]\!] = true$: the inner loop terminates starting from any state.

Now consider the ω-node corresponding to the outer loop, $outer = \langle a, b\rangle\ inner^{*}\ \langle b, a\rangle$. This loop illustrates a trade-off of compositionality. On one hand, compositionality makes proving termination easier: by the time that we reach the ω-node, we have already built a transition formula that summarizes the body of the outer loop. Despite the fact that the body contains an inner loop, we can use a theorem prover to answer questions about its behavior (conservatively, since the summary is an over-approximation). On the other hand, compositionality makes termination proving more difficult: a compositional analysis cannot prove that n *decreases* at each iteration, since it does not have access to the surrounding context of the loop that initializes *step* to 8. In Section 6.2 we provide a method that (for this particular loop) effectively performs a case split on whether n increases, decreases, or remains constant, and generates a mortal precondition that is sufficient for all three cases: $\mathcal{T}^{\omega}[\![outer^{\omega}]\!] = step > 0$. Thus, we have a *conditional* termination argument: the outer loop terminates as long as it begins in a state where *step* is positive.

Continuing up the DAG, we combine the mortal preconditions of the inner and outer loops to get

$$\mathcal{T}^{\omega}[\![outer^{\omega} + outer^{*}inner^{\omega}]\!] \equiv step > 0\ .$$

Finally, we compute a mortal precondition for the root of the DAG (and thus the whole program) by taking the weakest precondition of $step > 0$ under the transition formula $\mathcal{T}[\![\langle r, a\rangle]\!] = step' = 8 \land m' = m \land n' = n$, yielding the formula *true*. Thus, by propagating the conditional termination argument for the outer loop backwards through its context, the analysis discharges the assumption of the conditional termination argument, and verifies that the program always terminates.

3 Background

3.1 Flow Graphs and Path Expressions

A **control flow graph** $G = \langle V, E, r\rangle$ consists of a set of vertices V, a set of directed edges $E \subseteq V \times V$, and a root vertex $r \in V$ with no incoming edges. A **path** in G is a finite sequence $e_1e_2 \dots e_n \in E^*$ such that for each i, the destination of e_i matches the source of e_{i+1}; an ω-**path** is an infinite sequence $e_1e_2 \cdots \in E^{\omega}$ such that any finite prefix is a path. For any vertices $u, v \in V$, we use $Paths_G(u, v)$ to denote the (regular) set of paths in G from u to v, and we use $Paths^{\omega}_G(u)$ to denote the (regular) set of ω-paths in G starting from u.

We say that a vertex u **dominates** a vertex v if every path from r to v includes u. Every vertex dominates itself; we say u **strictly dominates** v if u dominates v and $u \neq v$. We say that u is the **immediate dominator** of v if it is the unique vertex that strictly dominates v and is dominated by every vertex that strictly dominates v. The immediate dominance relation forms a tree structure with r as the root; we use $children(v)$ to denote the set of vertices whose immediate dominator is v. We say that G is **reducible** if every cycle contains an edge $\langle u, v\rangle$ such that v dominates u.

Taking the alphabet Σ to be the set of edges in a given control flow graph G, a regular set of (finite) paths in G can be represented by a regular expression, and a regular set of ω-paths in G can be recognized by an ω-regular expression; we call such regular expressions (ω-)**path expressions**. The syntax of regular ($\mathsf{RegExp}(\Sigma)$) and ω-regular (ω-$\mathsf{RegExp}(\Sigma)$) expressions over an alphabet Σ is given by (see e.g. [5], Ch. 4):

$$\begin{aligned} a &\in \Sigma \\ e \in \mathsf{RegExp}(\Sigma) &::= a \mid 0 \mid 1 \mid e_1 + e_2 \mid e_1e_2 \mid e^* \\ f \in \omega\text{-}\mathsf{RegExp}(\Sigma) &::= e^{\omega} \mid ef \mid f_1 + f_2 \end{aligned}$$

where 0 recognizes the empty language, 1 recognizes the empty word, + corresponds to union, juxtaposition (or $\cdot$) to concatenation, $*$ to unbounded repetition, and ω to infinite repetition.

3.2 Logic and Geometry

The syntax of linear integer arithmetic (LIA) is given as follows:

$$\begin{aligned} x &\in \mathsf{Variable} \\ n &\in \mathbb{Z} \\ t \in \mathsf{Term} &::= x \mid n \mid n \cdot t \mid t_1 + t_2 \\ F \in \mathsf{Formula} &::= t_1 \leq t_2 \mid t_1 = t_2 \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \neg F \\ &\quad\mid \exists x.F \mid \forall x.F \end{aligned}$$

Let $X \subseteq \mathsf{Variable}$ be a set of variables. A **valuation** over X is a map $v : X \to \mathbb{Z}$. If F is a formula whose free variables range over X and v is a valuation over X, then we say that v satisfies F (written $v \models F$) if the formula F is true when interpreted over the standard model of the integers, using v to interpret the free variables. We write $F \models G$ if every valuation that satisfies F also satisfies G.

For a formula F, we use $F[x \mapsto t]$ to denote the formula obtained by substituting each free occurrence of the variable x with the term t. We use the same notation to represent parallel substitution of multiple variables by multiple terms; e.g., if X is a set of variables and $X' = \{x' : x \in X\}$ is a set of "primed" versions of those variables, then $F[X \mapsto X']$ denotes the result of replacing each variable in x with its corresponding x'. Substitution binds more tightly than logical connectives, so e.g., in the formula $F \land G[x \mapsto y]$, x is replaced with y within G, but not within F.

Let F be an LIA formula with free variables $\mathbf{x} = x_1, \dots, x_n$. The **convex hull** of F, denoted $conv(F)$, is the strongest (unique up to equivalence) formula of the form $A\mathbf{x} \geq \mathbf{b}$ that

is entailed by F, where A is an integer matrix and $\mathbf{b}$ is an integer vector. Farzan and Kincaid [26] give an algorithm for computing $conv(F)$.

3.3 Transition Formulas

Fix a finite set Var of variables, and let $\text{Var}' = \{x' : x \in \text{Var}\}$ denote a set of "primed copies", presumed to be disjoint from Var. A **state formula** is an LIA formula whose free variables range over Var. A **transition formula** is an LIA formula whose free variables range over $\text{Var} \cup \text{Var}'$. We use **SF** and **TF** to denote sets of state and transition formulas, respectively. Define a *state* to be a valuation over Var (the set of which we denote State) and a *transition* to be a valuation over $\text{Var} \cup \text{Var}'$. Any pair of states s, s' defines a transition $[s, s']$ which interprets each $x \in \text{Var}$ as $s(x)$ and each $x' \in \text{Var}'$ as $s'(x)$. A transition formula F defines a relation $\to_F$ on states, with $s \to_F s' \iff [s, s'] \models F$.

Define the *relational composition* of two transition formulas to be the formula

$$F_1 \circ F_2 \triangleq \exists \text{Var}''.F_1[\text{Var}' \mapsto \text{Var}''] \wedge F_2[\text{Var} \mapsto \text{Var}''] \ .$$

For any $k \in \mathbb{N}$, we use F^k to denote the k-fold relational composition of F with itself. For a transition formula F and a state formula S, define the weakest precondition of S under F to be the formula

$$wp(F, S) \triangleq \forall \text{Var}'.F \Rightarrow S[\text{Var} \mapsto \text{Var}'] \ .$$

We suppose the existence of an operation $(-)^\star$ that over-approximates the reflexive transitive closure of a transition formula (i.e., for any transition formula F, we have $\to_F^* \subseteq \to_{F^\star}$). Several such operators exist [23, 26, 34, 35, 42]; here we will describe one such method, based on techniques from [4, 26].

Let $\mathbf{x}'$ and $\mathbf{x}$ be vectors containing the variables Var′ and Var, respectively; let $n = |\text{Var}|$ be the dimension of these vectors. In general, the transitive closure of a transition formula cannot be expressed in first-order logic. Two special cases where the transitive closure can be expressed are:

1. If F takes the form $pre \wedge post$, where the free variables of pre range over Var and the free variables of $post$ range over Var′, then F is already transitively closed, so we need only to take its reflexive closure: $(pre \wedge post) \vee (\bigwedge_{x \in \text{Var}} x' = x)$
2. If F takes the form $A\mathbf{x}' \geq A\mathbf{x} + \mathbf{b}$, then for any $k \in \mathbb{N}$, we have that F^k is equivalent to $A\mathbf{x}' \geq A\mathbf{x} + k\mathbf{b}$, and so the formula $\exists k.k \geq 0 \wedge A\mathbf{x}' \geq A\mathbf{x} + k\mathbf{b}$ represents the reflexive transitive closure of F.

Let F be a transition formula. We cannot expect F to take one of the above forms, but we *can* always over-approximate F by a formula that does:

1. Let $Pre(F) \triangleq \exists \text{Var}'.F$ and let $Post(F) \triangleq \exists \text{Var}.F$. We have that $F \models Pre(F) \wedge Post(F)$, and $Pre(F) \wedge Post(F)$ takes form (1) above.
2. For each variable x, let δ_x denote a fresh variable which we use to represent the difference between x' and x; we use δ to denote a vector containing the δ_x variables. The convex hull
$$conv\left(\exists \text{Var}, \text{Var}'.F \wedge \bigwedge_{x \in \text{Var}} \delta_x = x' - x\right)$$
takes the form $A\delta \geq \mathbf{b}$. Then we have $F \models A\mathbf{x}' \geq A\mathbf{x}+\mathbf{b}$, and $A\mathbf{x}' \geq A\mathbf{x} + \mathbf{b}$ takes form (2) above.

Combining (1) and (2), we define an operation exp by

$$\exp(F, k) \triangleq \left(\left(\bigwedge_{x \in \text{Var}} x' = x\right) \vee (Pre(F) \wedge Post(F))\right) \wedge A\mathbf{x}' \geq A\mathbf{x} + k\mathbf{b}$$

and observe that for any $k \in \mathbb{N}$, we have that $F^k \models \exp(F, k)$. Finally, we over-approximate transitive closure by existentially quantifying over the number of loop iterations:

$$F^\star \triangleq \exists k.k \geq 0 \wedge \exp(F, k) \ .$$

Lemma 3.1. *The* $(-)^\star$ *and* exp *operators are monotone in the sense that if* $F \models G$*, then* $F^\star \models G^\star$ *and* $\exp(F, k) \models \exp(G, k)$ *(where* k *is a variable symbol).*

3.4 Transition Systems

A **transition system** T is a pair $T = \langle S_T, R_T \rangle$ where S_T is a set of states and $R_T \subseteq S_T \times S_T$ is a transition relation. We write $s \to_T s'$ to denote that the pair $\langle s, s' \rangle$ belongs to R_T. We say that a state $s \in S_T$ is **mortal** if there exists no infinite sequence $s \to_T s_1 \to_T s_2 \to_T s_3 \dots$. A **mortal precondition** for T is a state formula such that any state that satisfies the formula is mortal.

Each transition formula F defines a transition system, where the state space is State, and where the transition relation is $\to_F$. Define a **mortal precondition operator** to be a function $mp : \textbf{TF} \to \textbf{SF}$, which given a transition formula F, computes a state formula $mp(F)$ that is a mortal precondition for F. We say that mp is **monotone** if for any transition formulas F_1, F_2 with $F_1 \models F_2$, we have $mp(F_2) \models mp(F_1)$ (Note that this definition is *antitone* with respect to the entailment ordering, but since *weaker* mortal preconditions are more desirable it is natural to order mortal preconditions by reverse entailment.)

Example 3.2 Gonnord et al. [30] give a complete method for synthesizing linear lexicographic ranking functions (LLRFs) for transition formulas. We may define a monotone mortal precondition operator mp_{LLRF} as follows:

$$mp_{LLRF}(F) \triangleq \begin{cases} true & \text{if there is an LLRF for } F \\ \neg Pre(F) & \text{otherwise} \end{cases}$$

The fact that mp_{LLRF} is monotone follows from the fact that if $F_1 \models F_2$ then $Pre(F_1) \models Pre(F_2)$ and any LLRF for F_2 is

also an LLRF for F_1, and the completeness of Gonnord et al. [30]'s LLRF synthesis procedure. ⌟

Within this paper, a program is represented as a **labeled control flow graph** $P = \langle G, L \rangle$, where $G = \langle V, E, r \rangle$ is a control flow graph, and $L : E \to \mathbf{TF}$ is a function that labels each edge with a transition formula. P defines a transition system $TS(P)$ where the state space is $V \times$ State, and where $\langle v_1, s_1 \rangle \to_P \langle v_2, s_2 \rangle$ iff $\langle v_1, v_2 \rangle \in E$ and $[s_1, s_2] \models L(v_1, v_2)$.

4 An Efficient ω-Path Expression Algorithm

This section describes an algorithm for computing an ω-regular expression that recognizes all infinite paths in a graph that start at a designated vertex. The algorithm is based on Tarjan's path expression algorithm, which computes path expressions that recognize *finite* paths that start at a designated vertex [45]. Our algorithm operates in $O(|E|\alpha(|E|) + t)$ time, where α is the inverse Ackermann function and t is technical parameter that is $O(|V|)$ for reducible flow graphs and is at most $O(|V|^3)$, matching the complexity of Tarjan's algorithm.

It is technically convenient to formulate our algorithms on *path graphs* rather than control flow graphs. A **path graph** for a flow graph $G = \langle V, E, r \rangle$ is a graph $H = \langle U, W \rangle$ where $U \subseteq V$ and $W \subseteq U \times \text{RegExp}(E) \times U$ is a set of directed edges labeled by regular expressions over E, and such that for every $\langle u, e, v \rangle \in W$, e recognizes a subset of $\mathit{Paths}_G(u, v)$. We say that H *represents* a path p from u to v (in G, with $u, v \in U$) if there is a path

$$(w_1, e_1, w_2)(w_2, e_2, w_3) \dots (w_n, e_n, w_{n+1})$$

in H with $w_1 = u$ and $w_{n+1} = v$ and p is recognized by the regular expression $e_1 e_2 \dots e_n$. Similarly, H represents an ω-path p if there is a decomposition $p = p_1 p_2 p_3 \dots$ and an ω-path $(w_1, e_1, w_2)(w_2, e_2, w_3) \dots$ in H with p_i recognized by e_i for all i. We use $\mathit{PathRep}_H(u, v)$ to denote the set of paths from u to v that are represented by H, and $\mathit{PathRep}^\omega_H(v)$ to denote the set of ω-paths starting at v that are represented by H. We say that H is **complete** for a set of edges $E' \subseteq E$ if

1. For each $u, v \in U$, $\mathit{PathRep}_H(u, v)$ is the set of paths from u to v in G consisting only of edges from E'.
2. For each $v \in U$, $\mathit{PathRep}^\omega_H(v)$ is the set of ω-paths from v in G consisting only of edges from E', and which visit some vertex of U infinitely often.

4.1 A Naïve Algorithm

Algorithm 1 is a naïve algorithm for computing path expressions, which is used as a sub-procedure in the main algorithm. It is a variation of the classic state elimination algorithm for converting finite automata to regular expressions. The input to Algorithm 1 is a path graph $H = \langle U, W \rangle$ (for some flow graph G) and a root vertex r (not necessarily the root of G); its output is a pair consisting of an ω-path expression that recognizes $\mathit{PathRep}^\omega_H(r)$ and a function that maps each vertex $v \in U$ to a path expression that recognizes $\mathit{PathRep}_H(r, v)$. The idea is to successively eliminate the outgoing edges of every vertex in the graph except the root, while preserving the set of paths (and ω-paths) emanating from vertices whose outgoing edges have not yet been removed. The algorithm operates in $O(|U|^3)$ time.

```
1  Subroutine solve-dense(H, r) begin
     Input  : Path graph H = ⟨U, W⟩, vertex r ∈ U
              with no incoming edges
     Output : Pair ⟨pe^ω, pe⟩ where
              pe^ω ∈ ω-RegExp(E) recognizes
              PathRep^ω_H(r) and pe : U → RegExp(E)
              maps each v ∈ U to a path expression
              that recognizes PathRep_H(r, v).
     /* pe(u, v) recognizes paths from u to v represented by H */
2    pe ← λ ⟨u, v⟩ .0;
3    foreach ⟨u, e, v⟩ ∈ W do
4      pe(u, v) ← pe(u, v) + e;
     /* Suppose V is ordered as V = {r = v_0, v_1, ..., v_n} */
5    for i = n downto 1 do
6      for j = i − 1 downto 0 do
7        e_ji ← pe(v_j, v_i) · pe(v_i, v_i)*;
8        for k = n downto 1, k ≠ i do
9          pe(v_j, v_k) ← pe(v_j, v_k) + e_ji · pe(v_i, v_k)
10   return ⟨ Σ_{i=1}^n pe(r, v_i) · pe(v_i, v_i)^ω,
              λv.pe(r, v) · pe(v, v)* ⟩
```

Algorithm 1: Naïve path expression algorithm

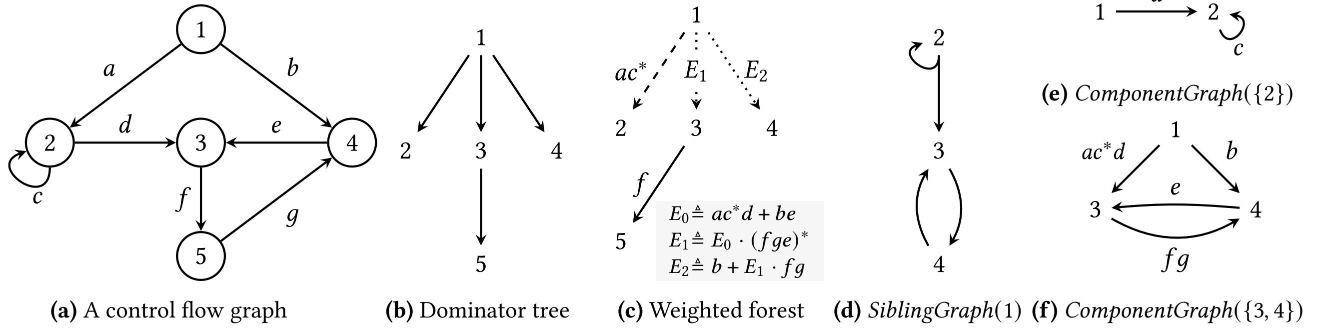
4.2 ω-Path Expressions in Nearly Linear Time

Algorithm 2 is an efficient ω-path expression algorithm that exploits sparsity of control flow graphs. Following Tarjan [45], the algorithm uses the dominator tree of the graph to break it into single-entry components, and uses a compressed weighted forest data structure to combine paths from different components.

A **compressed weighted forest** is a data structure that represents a forest of vertices with edges weighted by regular expressions. The data structure supports the following operations:

- $\mathsf{link}(u, e, v)$: set v to be the parent of u by adding an edge from v to u labeled e (u must be a root)
- $\mathsf{find}(v)$: return the (unique) vertex u such that $u \to^* v$ and u is a root
- $\mathsf{eval}(v)$: return the regular expression $e_1 \cdots e_n$, where $u_1 \xrightarrow{e_1} u_2 \xrightarrow{e_2} \cdots \xrightarrow{e_n} v$ is the path from a root to v.

This data structure can be implemented so that each operation takes $O(\alpha(n))$ amortized time, where n is the number of vertices in the forest [44].



(a) A control flow graph **(b)** Dominator tree **(c)** Weighted forest **(d)** *SiblingGraph*(1) **(f)** *ComponentGraph*({3, 4})

Figure 2. Operation of solve-sparse on an example control flow graph.

The subroutine solve-sparse(v) returns an ω-path expression that recognizes the set of ω-paths $Paths_G^{\omega}(v) \cap E|_v^{\omega}$, where

$$E|_v \triangleq \{\langle u_1, u_2\rangle \in E : u_2 \text{ is strictly dominated by } v\} \ .$$

Moreover, it maintains the invariant that after completing a call to solve-sparse(v), we have that for every vertex u that is dominated by v, find$(u) = v$ and eval(u) is a path expression that recognizes $Paths_G(v, u) \cap E|_v^*$.

The solve-sparse(v) subroutine is structured as a recursive traversal of the dominator tree (see Example 4.1). First, it calls solve-sparse(c) for each child c of v in the dominator tree. Next, it computes a directed graph $G_v = SiblingGraph(v)$ whose vertices are $children(v)$, and such that there is an edge $\langle c_1, c_2\rangle$ iff there is a path from c_1 to c_2 in G such that each edge (except the last) belongs to $E|_{c_1}$. The edges of $SiblingGraph(v)$ can be computed efficiently as

$$\{\langle \text{find}(u), c\rangle : c, \text{find}(u) \in children(v), \langle u, c\rangle \in E\} \ .$$

The correctness argument for the edge computation is as follows. If there is a path from c_1 to c_2 consisting of $E|_{c_1}$ edges, it takes the form $\pi \langle u, c_2\rangle$ for some $\langle u, c_2\rangle$ in E, with $\pi \subseteq E|_{c_1}^*$ a path from c_1 to u. Since π ends at u and consists only of $E|_{c_1}$-edges, u is dominated by c_1. Since solve-sparse(v) calls solve-sparse(c_1) before constructing *SiblingGraph*, we must have find$(u) = c_1$ by the invariants of solve-sparse.

Next, solve-sparse computes the strongly connected components of G_v and processes them in topological order. The loop (lines 9-16) maintains the invariant that when processing a component C, for every sibling node u that is topologically ordered before C, we have that find$(u) = v$ and that eval(u) recognizes $Paths_G(v, u) \cap E|_v^*$. To process a component C, we form a path graph $G_C = ComponentGraph(C)$ whose vertices are $C \cup \{v\}$ and that is complete for $E|_v$, computing a path expression $C\text{-}pe(u)$ for each $u \in C$ that recognizes $Paths_G(v, u) \cap E|_v^*$ using solve-dense, and then linking u to v with the path expression $C\text{-}pe(u)$ in the compressed weighted forest. The edges of $ComponentGraph(C)$ are obtained by collecting all weighted edges of the form $\langle \text{find}(w), \text{eval}(w), u\rangle$ such that $u \in C$ and $\langle w, u\rangle \in E$; the fact that G_C is complete for $E|_v$ follows from the loop invariant, using an argument analogous to the correctness argument for the *SiblingGraph* construction above. Finally, we return an ω-path expression which is the sum of (line 13) the ω-path expressions for each component and (line 16) an ω-path expression for each child c, pre-concatenated with a path expression recognizing $Paths_G(v, c) \cap E|_v^*$.

Example 4.1 Figure 2 illustrates the solve-sparse procedure. Figure 2a depicts a control flow graph, whose dominator tree appears in Figure 2b (for legibility, we refer to edges by label rather than by their endpoints). Consider the operation of solve-sparse(1). The compressed weighted forest after calling solve-sparse on 1's children 2, 3, 4 is depicted in Figure 2c (the single solid link from 3 to 4 labeled f; the other links are added later). The sibling graph for 1 is given in Figure 2d – observe that it has two strongly connected components: $\{2\}$ and $\{3, 4\}$, with $\{2\}$ ordered topologically before $\{3, 4\}$.

The loop (lines 9-16) processes $\{2\}$ first, producing the component graph in Figure 2e. Then 2 is linked to 1 in the compressed weighted forest (dashed edge of Figure 2c) with the regular expression ac^* (the paths from 1 to 2 represented by $ComponentGraph(\{2\})$).

Next, the loop processes the $\{3, 4\}$ component, producing the component graph in Figure 2f; note that the edge from 2 to 3 in G produces the edge from 1 to 3 (since find$(2) = 1$) and the edge from 5 to 4 produces the edge from 3 to 4 (since find$(5) = 3$). Then 3 and 4 are both linked to 1 in the compressed weighted forest (dotted edges of Figure 2c).

Finally, solve-sparse returns the sum solve-dense$(G_{\{2\}}, 1)$ and solve-dense$(G_{\{3,4\}}, 1)$, which is the ω-path expression

$$(ac^*d + be)(fge)^{\omega} + ac^{\omega}$$ ⌟

Algorithm 2 operates in $O(|E|\alpha(|E|) + t)$ time, where t is the time taken by the calls to solve-dense. For reducible flow graphs, each sibling graph is a singleton (see [45]), so the complexity simplifies to $O(|E|\alpha(|E|))$.

```
1  Algorithm PathExp^ω_G(r) begin
     Input   : Graph G = ⟨V, E⟩ and root vertex r
     Output: An ω-path expression recognizing
             Paths^ω_G(r)
2    children ← dominator tree for G;
3    Init compressed weighted forest with vertices V;
4    return solve-sparse(r)
5    Subroutine solve-sparse(v) begin
       Input   : Vertex v ∈ V
       Output: An ω-path expression recognizing
               Paths^ω_G(v) ∩ E|^ω_v
6      foreach child c ∈ children(v) do
7        child-pe_ω(c) ← solve-sparse(c);
8      G_v ← SiblingGraph(v);
9      pe_ω ← 0 /* accumulating ω-path expression   */
10     foreach s.c.c. C of G_v in topological order do
11       G_C ← ComponentGraph(C);
12       ⟨C-pe_ω, C-pe⟩ ← solve-dense(G_C, v);
13       pe_ω ← pe_ω + C-pe_ω;
14       foreach u ∈ C do
15         link(u, C-pe(u), v);
16         pe_ω ← pe_ω + C-pe(u) · child-pe_ω(u);
17     return pe_ω
```

Algorithm 2: An ω-path expression algorithm

5 Algebraic Termination Analysis

This section describes the process of interpreting an (ω-)regular expression within a suitable algebraic structure. As a particular case of interest, we show how to apply the algebraic framework to termination analysis.

An **interpretation** over an alphabet Σ consists of a triple $\mathcal{I} = \langle \mathbf{A}, \mathbf{B}, L \rangle$, where $\mathbf{A}$ is a *regular algebra*, $\mathbf{B}$ is a *ω-regular algebra over* $\mathbf{A}$, and $L : \Sigma \to \mathbf{A}$ is a *semantic function.* A **regular algebra** $\mathbf{A} = \left\langle A, 0^A, 1^A, +^A, \cdot^A, {}^{*^A} \right\rangle$ is an algebraic structure equipped with two distinguished elements $0^A, 1^A \in A$, two binary operations $+^A$ and $\cdot^A$, and a unary operation $(-)^{*^A}$. An ω-**algebra** over $\mathbf{A}$ is 4-tuple $\mathbf{B} = \left\langle B, \cdot^B, +^B, {}^{\omega^B} \right\rangle$ consisting of a universe B, an operation $\cdot^B : A \times B \to B$, an operation $+^B : B \times B \to B$, and an operation $(-)^{\omega^B} : A \to B$. A **semantic function** $L : \Sigma \to A$ maps the letters of Σ into the regular algebra $\mathbf{A}$.

Given an interpretation $\mathcal{I} = \langle \mathbf{A}, \mathbf{B}, L \rangle$ over an alphabet Σ, we can evaluate any regular expression e over Σ to an element $\mathcal{I}[\![e]\!]$ of $\mathbf{A}$ and any ω-regular expression f over Σ to an element $\mathcal{I}^\omega[\![f]\!]$ of $\mathbf{B}$ by interpreting each letter according to the semantic function and each (ω-)regular operator using its corresponding operator in $\mathbf{A}$ or $\mathbf{B}$:

$$\mathcal{I}[\![a]\!] \triangleq L(a) \quad \text{for } a \in \Sigma$$

$$\begin{aligned} \mathcal{I}[\![0]\!] &\triangleq 0^A \\ \mathcal{I}[\![1]\!] &\triangleq 1^A & \mathcal{I}^\omega[\![e^\omega]\!] &\triangleq \mathcal{I}[\![e]\!]^{\omega^B} \\ \mathcal{I}[\![e_1 e_2]\!] &\triangleq \mathcal{I}[\![e_1]\!] \cdot^A \mathcal{I}[\![e_2]\!] & \mathcal{I}^\omega[\![ef]\!] &\triangleq \mathcal{I}[\![e]\!] \cdot^B \mathcal{I}^\omega[\![f]\!] \\ \mathcal{I}[\![e_1 + e_2]\!] &\triangleq \mathcal{I}[\![e_1]\!] +^A \mathcal{I}[\![e_2]\!] & \mathcal{I}^\omega[\![f_1 + f_2]\!] &\triangleq \mathcal{I}^\omega[\![f_1]\!] +^B \mathcal{I}^\omega[\![f_2]\!] \\ \mathcal{I}[\![e^*]\!] &\triangleq \mathcal{I}[\![e]\!]^{*^A} \end{aligned}$$

If an ω-path expression f is represented by a DAG with n nodes, we can process the DAG bottom-up (as in Section 2) to compute $\mathcal{I}^\omega[\![f]\!]$ in $O(n)$ operations of $\mathbf{A}$ and $\mathbf{B}$.

5.1 Termination Analysis

This paper is primarily concerned with applying the above algebraic framework to termination analysis. The fundamental operation of interest is this setting the ω-iteration operator.

Fix a mortal precondition operator $mp : \mathbf{TF} \to \mathbf{SF}$. We define a regular algebra of transition formulas, $\mathbf{TF}$, and an ω-regular algebra of mortal preconditions, $\mathbf{MP}$. The universe of $\mathbf{TF}$ is the set of transition formulas, and the universe of $\mathbf{MP}$ is the set of state formulas. The operations are given below:

$$\begin{aligned} 0^{\mathbf{TF}} &\triangleq \mathit{false} \\ 1^{\mathbf{TF}} &\triangleq \bigwedge_{x \in \mathsf{Var}} x' = x & F^{\omega^{\mathbf{MP}}} &\triangleq mp(F) \\ F_1 +^{\mathbf{TF}} F_2 &\triangleq F_1 \lor F_2 & F \cdot^{\mathbf{MP}} S &\triangleq wp(F, S) \\ F_1 \cdot^{\mathbf{TF}} F_2 &\triangleq F_1 \circ F_2 & S_1 +^{\mathbf{MP}} S_2 &\triangleq S_1 \land S_2 \\ F^{*^{\mathbf{TF}}} &\triangleq F^\star \end{aligned}$$

Let $P = \langle G, L \rangle$ be a labeled control flow graph, which defines a transition system $TS(P)$. Let r be the root of G. Using the algorithm in Section 4, we can compute an ω-regular expression that recognizes all ω-paths in G beginning at r. By interpreting this regular expression (as above) under the interpretation $\mathcal{T} \triangleq \langle \mathbf{TF}, \mathbf{MP}, L \rangle$, we can under-approximate the mortal initial states of $TS(P)$. The correctness of this strategy is formalized below.

Proposition 5.1 (Soundness)**.** *Let $P = \langle G, L \rangle$ be a labeled CFG, let r be the root of G, and let $PathExp^\omega_G(r)$ be an ω-path expression recognizing $Paths^\omega_G(r)$. Then $\mathcal{T}^\omega \left[\!\left[PathExp^\omega_G(r)\right]\!\right]$ is a mortal precondition for $TS(P)$, in the sense that for any $s \models \mathcal{T}^\omega \left[\!\left[PathExp^\omega_G(r)\right]\!\right]$, we have that $\langle r, s \rangle$ is a mortal state of $TS(P)$. In particular, if $\mathcal{T}^\omega \left[\!\left[PathExp^\omega_G(r)\right]\!\right]$ is valid, then the program P has no infinite executions.*

Proposition 5.2 (Monotonicity)**.** *Suppose that mp is a monotone mortal precondition operator, and **MP** and **TF** are defined as above. Let $f \in \omega\text{-}RegExp(E)$, and let $L_1, L_2 : E \to \mathbf{TF}$ be semantic functions such that for all $e \in E$, $L_1(e) \models L_2(e)$. Define $\mathcal{T}_1 \triangleq \langle \mathbf{TF}, \mathbf{MP}, L_1 \rangle$ and $\mathcal{T}_2 \triangleq \langle \mathbf{TF}, \mathbf{MP}, L_2 \rangle$. Then $\mathcal{T}_2^\omega[\![f]\!] \models \mathcal{T}_1^\omega[\![f]\!]$.*

5.2 Inter-procedural Analysis

Our algebraic framework extends to the inter-procedural case using the method of Cook et al. [21]. The essential idea is to merge the control flow graphs of all procedures of a program into an *inter-procedural control flow graph* (ICFG) so that infinite paths through the program—including paths that are infinite due to the presence of recursion—correspond to infinite paths its ICFG. We may then compute ω-path expressions for the ICFG and interpret them, just as in the intra-procedural case. That is, the same analysis that is used to prove conditional termination for loops also can be applied to recursive functions. In the following we sketch the inter-procedural extension; see [51] for details.

We represent a multi-procedure program as a tuple

$$P = \langle V, E, \mathit{Proc}, \Lambda, \mathit{entry}, \mathit{exit} \rangle \ ,$$

where $\langle V, E \rangle$ is a finite directed graph, *Proc* is a finite set of procedure names, $\Lambda : E \to (\mathbf{TF} \cup \mathit{Proc})$ labels each edge by either a transition formula or a procedure call, and $\mathit{entry}, \mathit{exit} : \mathit{Proc} \to V$ are functions associating each procedure name with an entry and an exit vertex. We presume that the set of variables Var is divided into a set of local variables LVar and a set of global variables GVar. Note that procedures do not have parameters or return values, but these can be modeled using global variables (see Figure 3 for an example)

Fix a program P. Define its *inter-procedural* control flow graph $\mathit{ICFG} \triangleq (V, E_{\mathit{ICFG}})$. as follows. The vertices V are the same as the vertices of P. The edges $E_{\mathit{ICFG}} \triangleq E \cup \mathit{Interproc}$ are the edges of P plus an additional set of *inter-procedural* edges, which represent transfer of control between procedures by connecting the source of a call to the entry of the called procedure:

$$\mathit{Interproc} \triangleq \{\langle u, \mathit{entry}(p)\rangle : \exists \langle u, v\rangle \in E.\Lambda(u, v) = p\} \ .$$

An example *ICFG* appears in Figure 3; dashed edges correspond to inter-procedural edges.

Finally, we define a semantic function that can be used to interpret the edges of *ICFG*. A *summary assignment* is a function $S : \mathit{Proc} \to \mathbf{TF}$ that maps each procedure to a transition formula that over-approximates its behavior. For example, one possible summary assignment for Figure 3 is $S(\mathit{fib}) = g \leq r'$, indicating that the output of *fib* is no less than its input. Summary assignments can be computed using standard iterative techniques (some care needs to be taken to ensure monotonicity; see [51] for details). With a summary assignment S in hand, we can define a semantic function $L_S : E_{\mathit{ICFG}} \to \mathbf{TF}$ by

$$L_S(u, v) \triangleq \begin{cases} \Lambda(u, v) & \text{if } \langle u, v\rangle \in E \text{ and } \Lambda(u, v) \in \mathbf{TF} \\ S(p) & \text{if } \langle u, v\rangle \in E \text{ and } \Lambda(u, v) = p \\ \bigwedge_{x \in \mathrm{GVar}} x' = x & \text{if } \langle u, v\rangle \in \mathit{Interproc} \end{cases}$$

Theorem 5.3 (Inter-Procedural Soundness)**.** *Let P be a program. For any procedure $p \in P$, $\mathcal{T}^{\omega} [\![\mathit{PathExp}^{\omega}_{\mathit{ICFG}}(\mathit{entry}(p))]\!]$ is a mortal precondition for the procedure p, in the sense that for any state s such that $s \models \mathcal{T}^{\omega} [\![\mathit{PathExp}^{\omega}_{\mathit{ICFG}}(\mathit{entry}(p))]\!]$, we have that $\langle \mathit{entry}(p), s\rangle$ is a mortal state of P.*

```
1 fib(n):
2   if (n ≤ 1):
3     return 1
4   else
5     return fib(n - 1) + fib(n - 2)
```

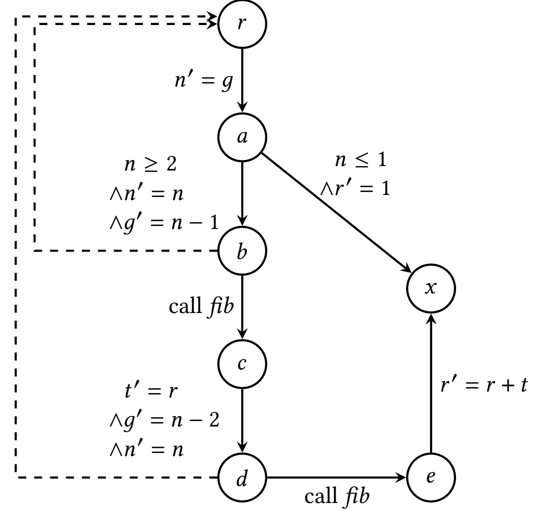


Figure 3. The recursive Fibonacci function (top), and representation as an inter-procedural control flow graph (bottom). The parameter and return are represented by the global variables g and r (respectively); t is a local temporary variable used to store the return value of the first recursive call. Dashed edges are inter-procedural.

Example 5.4 Consider the recursive Fibonacci function and its inter-procedural control flow graph pictured in Figure 3. We have

$$\begin{aligned} &\mathit{PathExp}^{\omega}_{\mathit{ICFG}}(r) = \mathit{body}^{\omega}, \text{ where} \\ &\qquad \mathit{body} = \langle r, a\rangle \langle a, b\rangle (\langle b, r\rangle + \langle b, c\rangle \langle c, d\rangle \langle d, r\rangle) \end{aligned}$$

Observe that any infinite execution of *fib* corresponds to a path in its *ICFG*, and therefore $\mathit{PathExp}^{\omega}_{\mathit{ICFG}}(r)$. We can compute a precondition under which Fibonacci terminates by evaluating $\mathit{PathExp}^{\omega}_{\mathit{ICFG}}(r)$, using $\mathit{mp}_{\mathit{LLRF}}$ as the mortal precondition operator:

$$\begin{aligned} \mathcal{T}[\![\mathit{body}]\!] &\equiv g \geq 2 \land (g' = g - 1 \lor g' = g - 2) \\ \mathcal{T}^{\omega} [\![\mathit{PathExp}^{\omega}_{\mathit{ICFG}}(r)]\!] &= \mathit{true} \end{aligned}$$

⌟

6 Modular Design of Mortal Precondition Operators

The interface provided by algebraic termination analysis is that the analysis designer provides a mortal precondition operator for transition formulas, and the framework "lifts" it to compute mortal preconditions for whole programs. Example 3.2 gives one instantiation of the mortal precondition operator using linear lexicographic ranking functions. This section demonstrates the applicability of the framework, by describing several combinators that can be used to construct monotone mortal precondition operators. A common theme to all is to take advantage of the properties of the algebraic framework (compositionality and monotonicity, in particular).

6.1 Termination Analysis for Free

Summarizing loops using an over-approximate transitive closure operator is an integral component of our algebraic framework. This section demonstrates that loop summarization can be also be exploited to construct a mortal precondition operator; i.e., an algebraic analyses for safety can be extended to prove termination analysis without any burden on the analysis designer.

Let F be a transition formula. A sufficient (but not necessary) condition for a state s of F to be mortal is that there is a bound on the length of any execution starting from s; that is there is some k such that for all s' with $[s, s'] \models F^k$, s' has no F-successors. This condition is not decidable, but it can be under-approximated using the procedure exp described in Section 3.3 (or any other method for over-approximating the iterated behavior of a transition formula); this yields the following mortal precondition operator:

$$mp_{\text{exp}}(F) \triangleq \exists k. \forall \mathsf{Var}', \mathsf{Var}''. k \geq 0 \land (\text{exp}(F, k) \Rightarrow \neg G)$$

where $G \triangleq F[\mathsf{Var} \mapsto \mathsf{Var}', \mathsf{Var}' \mapsto \mathsf{Var}'']$.

The fact that mp_{exp} is monotone follows from the monotonicity of quantification, conjunction, and the exp operator, and the fact that F and $\text{exp}(F, k)$ appear in negative positions in the formula.

Example 6.1 Consider the loop

```
while (x ≠ 0): x := x - 2,
```

with corresponding transition formula $F \triangleq x \neq 0 \land x' = x - 2$. In this case, we have $\text{exp}(F, k) \models x' = x - 2k$, and mp_{exp} computes the exact precondition for termination of the loop:

$$\begin{aligned} mp_{\text{exp}}(F) \equiv\ & \exists k. \forall x', x''. k \geq 0 \\ & \land (x' = x - 2k \Rightarrow \neg(x' \neq 0 \land x'' = x' - 2)) \\ \equiv\ & \exists k. k \geq 0 \land x - 2k = 0 \end{aligned}$$

i.e., the loop terminates provided that it begins in a state where x is a non-negative even number. ⌟

6.2 Phase Analysis

This section describes a *phase analysis* combinator that improves the precision of a given mortal precondition operator. The idea is to extract a *phase transition graph* from a transition formula, in which each vertex represents a phase of a loop, and each edge represents a phase transition. Using the algebraic framework from Section 5 and a given mortal precondition operator mp, we compute a mortal precondition for the phase transition graph, which (under mild assumptions) is guaranteed to be weaker than applying mp to the original transition formula (see Theorem 6.3). An important feature of phase analysis is that it can address the challenge of generating conditional termination arguments: even if some phases do not terminate, we can still use phase analysis to synthesize non-trivial mortal preconditions.

Let F be a transition formula. We say that a transition formula p is F-**invariant** if, should some transition of F satisfy p, then so too must any subsequent transition; that is, the formula $(F \land p) \circ (F \land \neg p)$ is inconsistent. Let P be a fixed set of transition formulas (e.g., in our implementation, we take P to be the set of all direction predicates, $P = \{x \bowtie x' : x \in \mathsf{Var}, \bowtie \in \{<, =, >\}\}$). Let $I(F, P)$ denote the F-invariant subset of P; $I(F, P)$ can be computed by checking the invariance condition for each formula in P using an SMT solver. The set of predicates $I(F, P)$ defines a partition $\mathcal{P}(F, P)$ of the set of transitions of F, where each cell corresponds to a valuation of the predicates in P (i.e., each cell has the form

$$F \land \left(\bigwedge_{p \in X} p \right) \land \left(\bigwedge_{p \in I(F,P) \setminus X} \neg p \right),$$

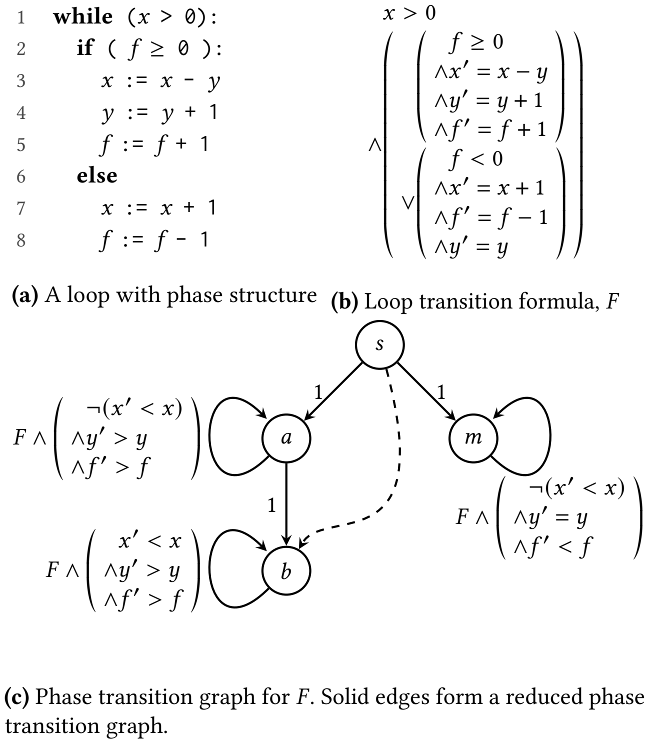
where X is a subset of $I(F, P)$). Since the predicates in $I(F, P)$ are F-invariant, this partition has the property that any infinite computation of F must eventually lie within a single cell of the partition.

Define the **phase transition graph** $Phase(F, P)$ to be a labeled control flow graph where the vertices are the cells of the partition $\mathcal{P}(F, P)$ plus a root vertex s, and which has the following properties: (1) each cell has a self-loop, labeled by the cell (2) if cell F_j can immediately follow F_i (i.e., $F_i \circ F_j$ is satisfiable), there is an edge from F_i to F_j with label $1^{\mathbf{TF}}$ (3) there is an edge from s to every cell with label $1^{\mathbf{TF}}$. The idea is that any infinite sequence $s_0 \to_F s_1 \to_F \cdots$ corresponds to an ω-path starting from s in G. Observe that this property is maintained if we relax conditions (2) and (3) so that we require only $1^{\mathbf{TF}}$-labeled *paths* rather than edges; call a phase transition graph *reduced* if it satisfies the relaxed conditions, and the number of edges is minimal. An algorithm that constructs a reduced phase transition graph is given in Algorithm 3.

We now define the phase analysis combinator. Suppose that mp is a mortal precondition operator; define the mortal precondition operator $mp_{Phase(P,mp)}$ as follows. Let F be a

1 **Subroutine** *phase-transition-graph*(F, P) **begin**
  Input : Formula F, set of transition predicates P
  Output: Reduced phase transition graph for F and P
  /* S is the set of literals for F-invariant predicates in P */
2   $S \leftarrow I(F, P) \cup \{\neg p : p \in I(F, P)\}$;
  /* Compute the cells of $\mathcal{P}(F, P)$ */
3   $n \leftarrow 0$;
4   **while** $F \wedge \bigwedge_{i=1}^{n} \neg F_i$ *is SAT* **do**
5   Select a model t with $t \models F \wedge \bigwedge_{i=1}^{n} \neg F_i$;
6     $n \leftarrow n + 1$;
7     $F_n \leftarrow F \wedge \bigwedge\{p \in S : t \models p\}$;
  /* Compute phase transitions */
8   Sort $F_1, \dots, F_n$ by # of positive literals;
9   $E \leftarrow \{\}$;
10   **for** $i = 2$ *to* n **do**
11   **for** $j = i - 1$ *downto* 1 **do**
12   **if** $\langle F_j, F_i \rangle \notin E^*$ *and* $F_j \circ F_i$ *is SAT* **then**
13         $E \leftarrow E \cup \{\langle F_j, F_i \rangle\}$;
  /* Connect virtual start node s to unreachable vertices */
14   $E \leftarrow E \cup \{\langle s, F_i \rangle : \nexists j. \langle F_j, F_i \rangle \in E\}$;
15   $E \leftarrow E \cup \{\langle F_i, F_i \rangle : 1 \leq i \leq n\}$; /* Add self-loops */
16   $L \leftarrow \lambda(F_i, F_j)$.**if** $i = j$ **then** F_i **else** $1^{\mathbf{TF}}$;
17   **return** $\langle \langle \{s, F_1, \dots, F_n\}, E, s \rangle, L \rangle$

Algorithm 3: Phase transition graph construction



(a) A loop with phase structure **(b)** Loop transition formula, F

(c) Phase transition graph for F. Solid edges form a reduced phase transition graph.

Figure 4. Analysis of a loop with a phase structure

transition formula. Construct the (reduced) phase transition graph $\langle G = \langle V, E, s \rangle, L \rangle$ using Algorithm 3. Compute an ω-path expression $PathExp_G^{\omega}(s)$ for G as in Section 4. Define an interpretation $\mathcal{T} \triangleq \langle \mathbf{TF}, \mathbf{MP}, L \rangle$, where the $(-)^{\omega^{\mathbf{MP}}}$ operator is taken to be mp. Finally, define

$$mp_{Phase(P,mp)}(F) \triangleq \mathcal{T}^{\omega} \left[\!\left[PathExp_G^{\omega}(s) \right]\!\right] .$$

Theorem 6.2 (Soundness)**.** *Let mp be a mortal precondition operator and let P be a set of transition predicates. Then $mp_{Phase(P,mp)}$ is a mortal precondition operator.*

Theorem 6.3 (Guaranteed improvement)**.** *Let mp be a monotone mortal precondition operator and let P be a set of transition predicates. Suppose that for any transition formula F, we have $wp(F^{\star}, mp(F)) = mp(F)$. Then $mp(F) \models mp_{Phase(P,mp)}(F)$.*

Theorem 6.4 (Monotonicity)**.** *Let mp be a monotone mortal precondition operator and let P be a set of transition predicates. Suppose that for any transition formula F, we have $wp(F^{\star}, mp(F)) = mp(F)$. Then the mortal precondition operator $mp_{Phase(P,mp)}$ is monotone.*

Example 6.5 Consider the loop in Figure 4. The loop does not always terminate, so mp_{LLRF} (Example 3.2) computes a trivial mortal precondition ($x \leq 0$). However, Algorithm 3 discovers a phase structure for this loop: once it execute the **then** branch, it cannot ever execute the **else** branch; the opposite is also true. Within the **then** branch, the variable x may increase (or remain constant) for some transient period, but then must ultimately decrease. This structure is depicted in the phase transition graph in Figure 4c.

Although the original loop has no linear lexicographic ranking function, the two phases in the **then** branch do: $-y$ is a ranking function for phase a and x is ranking function for phase b. The **else** branch does not, and so mp_{LLRF} generates a mortal precondition $x \leq 0 \vee f \geq 0$ (which is the trivial mortal precondition for phase m, but is a precise description of the mortal states of the original loop). Thus, by computing the mortal precondition of the loop using its phase graph rather than applying the mortal precondition operator to the loop itself, we get a weaker mortal precondition. ⌟

6.3 Combining Mortal Precondition Operators

State-of-the-art termination analyzers often use a portfolio of techniques to prove termination. Heuristics for selecting among appropriate techniques in a portfolio can be another source of unpredictable (non-monotone) behavior. A feature of our framework is that it makes it easy to combine the strengths of different termination analyses without such heuristics.

Suppose that mp_1 and mp_2 are mortal precondition operators. Then we can combine mp_1 and mp_2 into a single mortal precondition operator $mp_1 \otimes mp_2$ by defining

$$(mp_1 \otimes mp_2)(F) \triangleq mp_1(F) \vee mp_2(F) \ ;$$

if mp_1, mp_2 are monotone, then so too is $mp_1 \otimes mp_2$.

In fact, monotonicity allows us to do better. Define a second combinator by

$$(mp_1 \ltimes mp_2)(F) \triangleq mp_2(F \wedge \neg mp_1(F)) \ .$$

The intuition is that $mp_1 \ltimes mp_2$ is an *ordered* product, which asks mp_2 only to find a mortal precondition for the region of the state space that mp_1 cannot prove to be mortal. If we suppose that for all F we have $Pre(F) \models mp_2(F)$, then we have (for all F)

$$(mp_1 \otimes mp_2)(F) \models (mp_1 \ltimes mp_2)(F) \ .$$

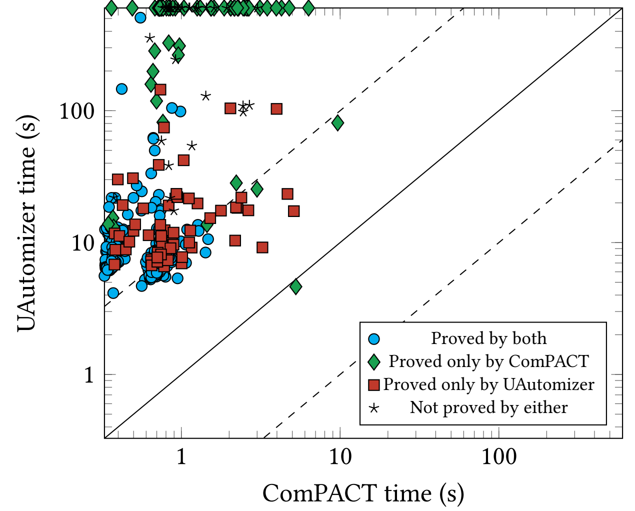


Figure 5. ComPACT vs. UAutomizer performance

7 Evaluation

Our tool ComPACT (**Com**positional and **P**redictable **A**nalysis for **C**onditional **T**ermination) implements the algebraic program analysis framework described in Sections 4 and 5), two mortal precondition operators mp_{LLRF} (Example 3.2) and mp_{exp} (Section 6.1), and the combinator mp_{Phase} (Section 6.2). ComPACT's default mortal precondition operator is $mp_{Phase(P, mp_{LLRF} \ltimes mp_{\text{exp}})}$ (where P is a set of direction predicates, $P \triangleq \{x \bowtie x' : x \in \text{Var}, \bowtie \in \{<, =, >\}\}$). We compare ComPACT against Ultimate Automizer [24], 2LS [14], and CPAchecker [38], the top three placing competitors in the termination category of the Competition on Software Verification (SV-COMP) 2020[1]. We also compare with Termite [30], which implements a complete procedure for linear lexicographic ranking function (LLRF) synthesis, to evaluate the effectiveness of our algebraic framework. With the exception of 2LS, all tools treat variables as unbounded integers.

Environment. We ran all experiments in a virtual machine with Ubuntu 18.04 and kernel version 5.3.0 − 62, with a single-core Intel Core i7-10710U CPU @ 1.10GHz and 8GB of RAM. All tools were run with a time limit of 10 minutes.

Benchmark design. We tested on a suite of 413 programs divided into 4 categories. The `termination`, `bitprecise`, and `recursive` suites contain small programs with challenging termination arguments, while the `polybench`[2] suite contains moderately sized kernels for numerical algorithms which have relatively simple termination arguments. The `termination` category consists of the *non-recursive, terminating* tasks in the `Termination-MainControlFlow` suite from SV-COMP. The `recursive` category consists of the *recursive, terminating* tasks from the `recursive` directory and `Termination-MainControlFlow`. The `bitprecise` category consists of the same tasks as the `termination` category, except that bounded integer semantics are encoded into unbounded integer semantics for a more accurate comparison with 2LS (minus one task, which Ultimate Automizer was able to prove to be non-terminating). Since signed overflow is undefined in C, proving termination necessitates proving absence of signed overflow. The encoding was performed by using `goto-instrument` [2, 3] to instrument the code with checks for signed overflow that enter an infinite loop on failure.

How does ComPACT compare with the state-of-the-art? A comparison of all tools across all test suites is shown in Table 1. Ultimate Automizer proves the most tasks in the `termination` and `bitprecise` suites, but uses significantly more time than ComPACT (Figure 5). ComPACT proves termination of the most tasks in the `recursive` and `polybench` suite (note that 2LS and Termite do not handle recursive programs, so we exclude them from the `recursive` suite). These results suggest that ComPACT is able to match or even exceed the capabilities of state-of-the-art termination provers while providing stronger behavioral guarantees.

How does each component contribute to ComPACT's capability? ComPACT implements two mortal precondition operators, LLRF-based mortal precondition operator mp_{LLRF} (LLRF) and transitive closure based mp_{exp} (exp), and the phase analysis (phase) combinator. We evaluate how each component contributes to ComPACT's ability to prove termination in Table 2. First we notice that there is a large overlap between the tasks solved by mp_{LLRF} and mp_{exp}. This can be attributed to the fact that both are sufficient to prove termination of loops with linear ranking functions, which

[1] https://sv-comp.sosy-lab.org/2020
[2] http://web.cs.ucla.edu/~pouchet/software/polybench

Table 1. Termination verification benchmarks; time in seconds.

benchmark	#tasks	ComPACT #correct	ComPACT time	2LS #correct	2LS time	UAutomizer #correct	UAutomizer time	CPAchecker #correct	CPAchecker time	Termite #correct	Termite time
termination	171	141	**81.7**	115	1925.8	**161**	4684.8	126	13434.6	78	937.5
bitprecise	169	115	**154.3**	111	1911.8	**122**	26596.5	92	32755.8	4	693.8
recursive	42	**31**	**49.6**	–	–	30	2073.8	23	710.1	–	–
polybench	30	**30**	93.8	0	7944.2	0	16285.8	0	4397.3	26	**36.7**
Total	412	**317**	**379.5**	226	11781.8	313	49640.8	241	51297.8	108	1668.0

Table 2. Contributions of different components implemented in ComPACT; time in seconds.

		ComPACT		Using mp_{LLRF} as base operator				Using mp_{exp} as base operator			
				LLRF only		LLRF + phase		exp only		exp + phase	
benchmark	#tasks	#correct	time	#correct	time	#correct	time	#correct	time	#correct	time
termination	171	**141**	81.7	122	**65.2**	138	70.3	112	72.2	130	92.4
bitprecise	169	**115**	154.3	105	**134.5**	**115**	142.6	103	175.7	113	240.6
recursive	42	**31**	49.6	15	**31.2**	22	38.7	24	44.7	**31**	78.9
polybench	30	**30**	93.8	**30**	**60.8**	**30**	93.4	**30**	86.5	**30**	565.0
Total	412	**317**	379.5	272	**291.7**	305	345.0	269	379.1	304	976.9

is the case for the majority of the tasks in our suite.[3] The relative strength of mp_{LLRF} is on loops with complex control structure (e.g., loops whose termination relies on precise reasoning about multiple paths through its body); the relative strength of mp_{exp} is on loops with non-convex guards (e.g., recursive functions where the recursive case is guarded by a disequality). Theorem 6.3 implies that the set of tasks that can be proved with phase analysis is a super-set of those that can be proved without; our experimental results show that the inclusion is strict for both configurations. These results above suggest that the algebraic framework can be successfully applied using a variety of different mortal precondition operators, and that different operators can be profitably combined.

Impact of compositionality and monotonicity. The algebraic framework "lifts" a termination analysis for transition formulas to whole programs. Comparing the LLRF column of Table 2 with the Termite results in Table 1 demonstrates the impact of this framework: both columns implement the same base analysis, but lift the analysis to whole programs in different ways. This comparison demonstrates the advantage of compositional summarization of nested loops, and also suggests that precision loss due to compositionality, i.e., synthesizing LLRFs without precise supporting invariants, is not substantial.

[3]We confirmed this fact by running the experiments with a modified mp_{LLRF} that finds only linear ranking functions: it succeeds on 258 tasks without phase analysis and 292 tasks with phase analysis.

A consequence of compositionality is that ComPACT has relatively stable running time across all tasks and scales to the larger tasks in the `polybench` suite. This suite contains program with loops that have complex control flow (e.g., nested loops) but simple termination arguments, in particular, **for** loops like

```
1   for(int i = 0; i < n; i++) { ... }
```

where the loop body does not contain instructions that decrease i. ComPACT is assured to prove termination of such loops as a consequence of compositionality and monotonicity. The other tools on our comparison, even those that employ complete procedures for linear ranking function synthesis, do not make such guarantees and may get stuck in the logic of the loop body. For example, ComPACT proves termination of the following loop in 0.3 seconds:

```
1   for(int i = 0; i < 4096; i++)
2     for(int j = 0; j < 4096; j++)
3       i = i;
```

Ultimate Automizer and CPAchecker exceed the 10 minute time limit on this loop, and 2LS and Termite return "unknown".

8 Related Work

Summarization for termination. At a high level, our procedure proves that a loop terminates by first computing a transition formula that summarizes the behavior of its body, and then performing termination analysis on the transition

formula. There are several approaches to termination analysis that similarly apply summarization to handle nested loops and procedure calls [8, 14, 47, 52]. There are various ways of formulating such an analysis. Berdine et al. [8] generates loop body summaries using a program transformation and a conventional state-based invariant generator (e.g., polyhedra analysis). Tsitovich et al. [47] takes an approach more similar to ours: summarization is an operation that replaces a subgraph of a control flow graph by edges that summarize that subgraph, and it is applied recursively to summarize nested loops. We take an algebraic view, inspired by Tarjan [45, 46], in which we generate an ω-regular expression representing the paths through a program and then define the analysis by recursion on that expression.

The contribution of Section 5 is to provide a unified framework in which these analyses can be understood. In view of the algebraic framework, prior work can be understood in terms of (1) the method used to summarize loops (i.e, the $(-)^*$ operator), and (2) the method used to prove termination (i.e, the $(-)^\omega$ operator). A concrete benefit of our framework in light of this prior work is that our approach handles recursive procedures and irreducible control flow, which are not supported by some of the prior approaches (including 2LS) [14, 47, 52].

Complete ranking function synthesis. A ranking function synthesis algorithm is *complete* if it is guaranteed to find a ranking function for a loop if one exists. Such techniques are related to our work in that we sought a termination analysis for which we can make guarantees about its behavior. Complete ranking function synthesis algorithms exist for a variety of classes of ranking functions, such as linear [39], linear-lexicographic [11], nested [37], multi-phase [7], . . .). These algorithms apply only to very restricted classes of loops, and in particular there are no complete ranking function synthesis algorithms that operate on nested loops or recursive procedures. The seminal work on Terminator gives a general method for applying complete ranking function synthesis algorithms to general programs by using them in a counter-example guided refinement loop [20]. Our framework of *algebraic termination analysis* provides another general method, which allows the *completeness* guarantee to carry over to a *monotonicity* guarantee for the whole analysis.

Conditional termination. In a compositional setting it is natural to formulate the termination problem as the problem of finding a sufficient condition under which a fragment of code is guaranteed to terminate (i.e., a *mortal precondition*), rather than the decision problem of universal termination. Approaches to conditional termination include quantifier elimination [16], abstract interpretation [22, 48–50], abductive inference [36], conflict-driven learning [25], incremental backwards reasoning [29], and constraint-based methods [9]. Our approach is unique in that we provide a conditional termination analysis that is both monotone and can be applied to a general program model.

Bozga et al. [10] is closest to our work in that they give an algorithm for which there are guarantees about its behavior beyond soundness, albeit for a limited class of loops. They give a technique for synthesizing the set of mortal states of a loop, provided a logical formula representing the exact transitive closure of that loop in a logical theory that admits quantifier elimination. In Section 6.1, we use a related idea to *under-approximate* the mortal states of a loop using an *over-approximation* of the transitive closure of the loop.

Control flow refinement. Section 6.2 defines a mortal precondition combinator that improves the precision of a given mortal precondition operator by exposing phase structure in loops. There are several related approaches for improving analysis results by program transformation [6, 23, 27, 28, 31, 40, 41]. In particular, the *transition invariant predicates* from Section 6.2 are essentially a transition-predicate analogue of the (state-based) *splitter predicates* from [41]; our method for checking whether a candidate transition predicate is invariant and partitioning the transition space are new. Cyphert et al.'s work [23] on refinement of path expressions is closest to ours in that it is based on an algebraic program analysis and provides a guarantee of improvement. The refinement strategy is based on altering the path expression algorithm, whereas phase analysis alters the algebra of the analysis. Operating at the algebra level enables us to formulate and prove a monotonicity theorem.

9 Conclusion

This paper presents a termination analysis that is both *compositional* and *monotone*. We extended Tarjan [45, 46]'s path expression method from safety analysis to termination analysis, by using ω-regular expressions to represent languages of infinite paths and ω-algebras to interpret those expressions. One direction for future work is to apply this framework to other analyses that require reasoning about infinite and potentially infinite paths, such as non-termination analysis, resource bound analysis, and verification of linear temporal properties.

Acknowledgments

This work was supported in part by the NSF under grant number 1942537 and by ONR under grant N00014-19-1-2318. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2008. Automatic Inference of Upper Bounds for Recurrence Relations in Cost

Analysis. In *SAS*. 221–237. https://doi.org/10.1007/978-3-540-69166-2_15

[2] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013. Software verification for weak memory via program transformation. In *European Symposium on Programming*. Springer, 512–532. https://doi.org/10.1007/978-3-642-37036-6_28

[3] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial orders for efficient bounded model checking of concurrent software. In *International Conference on Computer Aided Verification*. Springer, 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

[4] Corinne Ancourt, Fabien Coelho, and François Irigoin. 2010. A Modular Static Analysis Approach to Affine Loop Invariants Detection. *Electr. Notes Theor. Comp. Sci.* 267, 1 (Oct. 2010), 3–16. https://doi.org/10.1016/j.entcs.2010.09.002

[5] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. The MIT Press.

[6] G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, and A. Gupta. 2009. Refining the Control Structure of Loops using Static Analysis. In *EMSOFT*. https://doi.org/10.1145/1629335.1629343

[7] A. M. Ben-Amram and S. Genaim. 2017. On Multiphase-Linear Ranking Functions. In *CAV*. 601–620. https://doi.org/10.1007/978-3-319-63390-9_32

[8] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. 2007. Variance analyses from invariance analyses. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 211–224. https://doi.org/10.1145/1190216.1190249

[9] Cristina Borralleras, Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2017. Proving Termination Through Conditional Termination. In *TACAS*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 99–117. https://doi.org/10.1007/978-3-662-54577-5_6

[10] Marius Bozga, Radu Iosif, and Filip Konečný. 2012. Deciding Conditional Termination. In *TACAS*, Cormac Flanagan and Barbara König (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 252–266. https://doi.org/10.1007/978-3-642-28756-5_18

[11] A. R. Bradley, Z. Manna, and H. B. Sipma. 2005. Linear ranking with reachability. In *CAV*. 491–504. https://doi.org/10.1007/11513988_48

[12] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *TACAS*. 387–393. https://doi.org/10.1007/978-3-662-49674-9_22

[13] Q. Carbonneaux, J. Hoffmann, and Z. Shao. 2015. Compositional Certified Resource Bounds. In *PLDI*. https://doi.org/10.1145/2813885.2737955

[14] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. 2018. Bit-Precise Procedure-Modular Termination Analysis. *ACM Trans. Program. Lang. Syst.* 40, 1 (2018), 1:1–1:38. https://doi.org/10.1145/3121136

[15] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. 2007. Proving That Programs Eventually Do Something Good. In *POPL*. 265–276. https://doi.org/10.1145/1190216.1190257

[16] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. 2008. Proving Conditional Termination. In *CAV*, Aarti Gupta and Sharad Malik (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 328–340. https://doi.org/10.1007/978-3-540-70545-1_32

[17] Byron Cook, Heidy Khlaaf, and Nir Piterman. 2015. On Automation of CTL* Verification for Infinite-State Systems. In *CAV*. 13–29. https://doi.org/10.1007/978-3-319-21690-4_2

[18] Byron Cook and Eric Koskinen. 2011. Making Prophecies with Decision Predicates. In *POPL*. 399–410. https://doi.org/10.1145/1925844.1926431

[19] Byron Cook and Eric Koskinen. 2013. Reasoning About Nondeterminism in Programs. In *PLDI*. 219–230. https://doi.org/10.1145/2491956.2491969

[20] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination Proofs for Systems Code. In *PLDI*. 415–426. https://doi.org/10.1145/1133981.1134029

[21] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2009. Summarization for termination: no return! *Formal Methods in System Design* 35, 3 (2009), 369–387. https://doi.org/10.1007/s10703-009-0087-8

[22] Patrick Cousot and Radhia Cousot. 2012. An Abstract Interpretation Framework for Termination. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 245–258. https://doi.org/10.1145/2103656.2103687

[23] John Cyphert, Jason Breck, Zachary Kincaid, and Thomas Reps. 2019. Refinement of Path Expressions for Static Analysis. *Proc. ACM Program. Lang.* 3, POPL, Article 45 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290358

[24] Daniel Dietsch, Matthias Heizmann, Alexander Nutz, Claus Schätzle, and Frank Schüssele. 2020. Ultimate Taipan with Symbolic Interpretation and Fluid Abstractions - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12079)*, Armin Biere and David Parker (Eds.). Springer, 418–422. https://doi.org/10.1007/978-3-030-45237-7_32

[25] Vijay D'Silva and Caterina Urban. 2015. Conflict-Driven Conditional Termination. In *CAV*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 271–286. https://doi.org/10.1007/978-3-319-21668-3_16

[26] A. Farzan and Z. Kincaid. 2015. Compositional Recurrence Analysis. In *FMCAD*. IEEE, 57–64. https://doi.org/10.1109/FMCAD.2015.7542253

[27] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. 2019. Inferring Inductive Invariants from Phase Structures. In *CAV*. Cham, 405–425. https://doi.org/10.1007/978-3-030-25543-5_23

[28] A. Flores-Montoya and R. Hähnle. 2014. Resource analysis of complex programs with cost equations. In *APLAS*. https://doi.org/10.1007/978-3-319-12736-1_15

[29] Pierre Ganty and Samir Genaim. 2013. Proving Termination Starting from the End. In *CAV*. 397–412. https://doi.org/10.1007/978-3-642-39799-8_27

[30] Laure Gonnord, David Monniaux, and Gabriel Radanne. 2015. Synthesis of Ranking Functions Using Extremal Counterexamples. *SIGPLAN Not.* 50, 6 (June 2015), 608–618. https://doi.org/10.1145/2813885.2737976

[31] S. Gulwani, S. Jain, and E. Koskinen. 2009. Control-flow Refinement and Progress Invariants for Bound Analysis. In *PLDI*. https://doi.org/10.1145/1543135.1542518

[32] S. Gulwani, K.K. Mehra, and T.M. Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *POPL*. https://doi.org/10.1145/1594834.1480898

[33] S. Gulwani and F. Zuleger. 2010. The Reachability-bound Problem. In *PLDI*. https://doi.org/10.1145/1806596.1806630

[34] Zachary Kincaid, Jason Breck, John Cyphert, and Thomas Reps. 2019. Closed Forms for Numerical Loops. *Proc. ACM Program. Lang.* 3, POPL, Article 55 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290368

[35] Z. Kincaid, J. Cyphert, J. Breck, and T.W. Reps. 2018. Non-Linear Reasoning for Invariant Synthesis. *PACMPL* 2(POPL) (2018), 54:1–54:33. https://doi.org/10.1145/3158142

[36] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and Non-Termination Specification Inference. In *PLDI* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 489–498. https://doi.org/10.1145/2737924.2737993

[37] J. Leike and M. Heizmann. 2014. Ranking Templates for Linear Loops. In *TACAS*. 172–186. https://doi.org/10.1007/978-3-642-54862-8_12

[38] Sebastian Ott. 2016. *Implementing a Termination Analysis using Configurable Program Analysis*. Master's thesis. University of Passau.

[39] A. Podelski and A. Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*. 239–251. https://doi.org/10.1007/978-3-540-24622-0_20

[40] X. Rival and L. Mauborgne. 2007. The Trace Partitioning Abstract Domain. *TOPLAS*. 29, 5 (2007). https://doi.org/10.1145/1275497.1275501

[41] R. Sharma, I. Dillig, T. Dillig, and A. Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *CAV*. https://doi.org/10.1007/978-3-642-22110-1_57

[42] Jake Silverman and Zachary Kincaid. 2019. Loop Summarization with Rational Vector Addition Systems. In *CAV*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 97–115. https://doi.org/10.1007/978-3-030-25543-5_7

[43] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2015. Difference Constraints: An Adequate Abstraction for Complexity Analysis of Imperative Programs. In *FMCAD*. 144–151. https://doi.org/10.1109/fmcad.2015.7542264

[44] Robert Endre Tarjan. 1979. Applications of Path Compression on Balanced Trees. *J. ACM* 26, 4 (Oct. 1979), 690–715. https://doi.org/10.1145/322154.322161

[45] R. E. Tarjan. 1981. Fast Algorithms for Solving Path Problems. *J. ACM* 28, 3 (July 1981), 594–614. https://doi.org/10.1145/322261.322273

[46] R. E. Tarjan. 1981. A Unified Approach to Path Problems. *J. ACM* 28, 3 (July 1981), 577–593. https://doi.org/10.1145/322261.322272

[47] Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. 2011. Loop Summarization and Termination Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6605)*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.). Springer, 81–95. https://doi.org/10.1007/978-3-642-19835-9_9

[48] Caterina Urban. 2013. The Abstract Domain of Segmented Ranking Functions. In *SAS*, Francesco Logozzo and Manuel Fähndrich (Eds.). 43–62. https://doi.org/10.1007/978-3-642-38856-9_5

[49] Caterina Urban and Antoine Miné. 2014. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In *European Symp. on Programming*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 412–431. https://doi.org/10.1007/978-3-642-54833-8_22

[50] Caterina Urban and Antoine Miné. 2014. A Decision Tree Abstract Domain for Proving Conditional Termination. In *SAS*, Markus Müller-Olm and Helmut Seidl (Eds.). Springer International Publishing, Cham, 302–318. https://doi.org/10.1007/978-3-319-10936-7_19

[51] Shaowei Zhu and Zachary Kincaid. 2021. Termination Analysis Without the Tears (extended version). *CoRR* abs/2101.09783 (2021). arXiv:2101.09783 https://arxiv.org/abs/2101.09783

[52] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 280–297. https://doi.org/10.1007/978-3-642-23702-7_22