Alternation for Termination

William R. Harris[1], Akash Lal[2], Aditya V. Nori[2], and Sriram K. Rajamani[2]

[1] University of Wisconsin; Madison, WI, USA
[2] Microsoft Research India; Bangalore, India

Abstract. Proving termination of sequential programs is an important problem, both for establishing the total correctness of systems and as a component of proving more general termination and liveness properties. We present a new algorithm, TREX, that determines if a sequential program terminates on all inputs. The key characteristic of TREX is that it alternates between refining an over-approximation and an under-approximation of each loop in a sequential program. In order to prove termination, TREX maintains an over-approximation of the set of states that can be reached at the head of the loop. In order to prove non-termination, it maintains of an under-approximation of the set of paths through the body of the loop. The over-approximation and under-approximation are used to refine each other iteratively, and help TREX to arrive quickly at a proof of either termination or non-termination.

TREX refines the approximations in alternation by composing three different program analyses: (1) local termination provers that can quickly handle intricate loops, but not whole programs, (2) non-termination provers that analyze one cycle through a loop, but not all paths, and (3) global safety provers that can check safety properties of large programs, but cannot check liveness properties. This structure allows TREX to be instantiated using any of the pre-existing techniques for proving termination or non-termination of individual loops.

We evaluated TREX by applying it to prove termination or find bugs for a set of real-world programs and termination analysis benchmarks. Our results demonstrate that alternation allows TREX to prove termination or produce certified termination bugs more effectively than previous techniques.

1 Introduction

Proving termination of sequential programs is an important problem, both for establishing total correctness of systems and as a component for proving other liveness properties [12]. However, proving termination efficiently for general programs remains an open problem. For an illustration of the problem, consider the example program shown in Fig. 1, and in particular the loop `L2` on lines 8–16. This loop terminates on all inputs, but for an analysis to prove this, it must derive two important facts: (1) the loop has an invariant $d > 0$ and (2) under this invariant, the two paths through the loop cannot execute together infinitely often. Existing analyses can discover one or the other of the above facts, but not both.

Some analyses construct a proof of termination in the form of a lexicographic linear ranking function (LLRF) [4]. These analyses can prove termination of `L2` by constructing a valid LLRF if they are given $d > 0$ as a loop invariant. However, LLRF-based

```
1   void f(int d) {
2     int x, y, k, z := 1;
3     ...
4   L1:
5     while (z < k) { z := 2 * z; }
6     ...
7   L2:
8     while (x > 0 && y > 0) {
9       if (*) {
10  P1:   x := x - d;
11        y := *;
12        z := z - 1;
13      } else {
14        y := y - d;
15      }
16    }
17  }
```

```
1   void main() {
2     if (*) {
3       f(1);
4     } else {
5       f(2);
6     }
7   }
```

Fig. 1. Example illustrating the effect of alternation.

tools have been designed to analyze only loops with affine assignments and conditions, and are unable to handle pointers, or perform inter-procedural, whole program analysis (which is required to establish the desired invariant).

Techniques that construct a transition invariant (TI) as proofs, such as TERMINATOR [10], can handle arbitrary programs with procedures and pointers, but are hampered by the way they construct a proof of termination. To illustrate this, consider how TERMINATOR analyzes loop `L2`. TERMINATOR first attempts to prove termination of `L2` by analyzing it in isolation from the rest of the program. However, TERMINATOR fails, as it is not aware of the additional fact that whenever the loop is reached, the invariant $d > 0$ holds. It thus generates a potential counterexample that may demonstrate that the loop does not always terminate. A counterexample to termination is a "lasso", which consists of a "stem" sequence of statements that executes once followed by a "cycle" sequence of statements that may then be executed infinitely often. For the example, TERMINATOR may generate a lasso with the stem "`d := 1; z := 1`" that leads to `L2`, followed by the cycle "`assume (x > 0); assume(y > 0); x := x - d; y := *; z := z - 1`" that executes infinitely often. If TERMINATOR ignores the stem, it cannot prove that the cycle will not execute infinitely often. Thus, it uses the state of the program after executing the stem, "`d = 1, z = 1`", to construct a new cycle "`assume(d = 1); assume (z = 1); assume (x > 0); assume(y > 0); x := x - d; y := *; z := z - 1`" whose behaviors under-approximate those of the original cycle. In the under-approximation, the conditions $d = 1$ and $z = 1$ are assumed to hold at the beginning of *every* iteration of the loop (see Section 3.4 of [10] for a discussion).

In this way, TERMINATOR constructs an under-approximation of the counterexample cycle in the hope that it can at least find a proof of termination for the under-approximation. With the added assumptions at the head of the cycle, it can find multiple proofs that the under-approximation eventually terminates. One such proof establishes that the expression $z-1$ is both bounded from below by 0 and must decrease through every iteration of the cycle. TERMINATOR then attempts to validate $z-1$ as a proof of termination of the entire loop by determining if there are any paths over which $z-1$ is not bounded and decreasing. There are, as the value of z is not bounded over the executions

of the loop. Thus TERMINATOR will find another counterexample to $z - 1$ as a proof of termination. For instance, it may find a trace that executes loop `L1` once, reaches `L2` with state $d = 1, z = 2$, and executes the same cycle as the previous counterexample. Similarly to how TERMINATOR handled the last counterexample, it constructs an under-approximate cycle "`assume(d = 1); assume(z = 2); assume(x > 0); assume(y > 0); x := x - d; y := *; z := z - 1;`" and attempts to prove its termination. Similar to the last counterexample, it determines that $z - 2$ is bounded from below by 0 and decreases each time through the loop. Again, this fact does not hold for all paths through the loop, so TERMINATOR will iterate again on another counterexample. In this way, TERMINATOR will converge on a proof of termination slowly, if at all.

To address these shortcomings in existing techniques, we propose TREX, a novel approach to proving termination of whole programs. TREX addresses the shortcomings of LLRF-based techniques and TERMINATOR with an algorithm that *alternates* between refining an over and under-approximation of the program. TREX analyzes loops in the program one at a time. For each loop L, it simultaneously maintains an over-approximation as a loop invariant for L (which is a superset of the states that can be reached at the loop-head) and an under-approximation as a subset of all the paths through L.

TREX first applies a loop termination prover to try to prove that no set of paths in the under-approximation can execute together infinitely often. If the loop termination prover can prove this, then it produces a *certificate* of the proof. TREX then checks if the certificate is a valid proof that no set of paths in the entire loop may execute infinitely often. If so, then the certificate demonstrates that the loop terminates on all inputs. If not, then TREX adds to the under-approximation paths that invalidate the certificate. TREX then reanalyzes the program using the new, expanded under-approximation. This technique is similar to those employed in TERMINATOR.

If TREX fails to prove that paths in the under-approximation do not execute infinitely often, then it applies a non-termination prover to find a sufficient condition for non-termination. This sufficient condition is a precondition under which the loop will *not* terminate. TREX then queries a safety prover to search for a program input that reaches the loop and satisfies this precondition. If the safety prover finds such an input, then the input is a true counterexample to termination. If the safety prover determines that the loop precondition is unreachable, then the negation of the precondition is an invariant for the loop. TREX conjoins this predicate to its existing invariant and reanalyzes the program using the new, strengthened over-approximation. This technique is novel to TREX.

In this way, TREX composes three analyzes for three distinct problems: (1) efficient local termination provers that can analyze a loop represented as a finite sets of paths, (2) non-termination provers that analyze a single trace, and (3) safety provers that prove global safety properties of programs. This composition allows each analysis to improve the performance of the other. The composition allows TREX to apply a loop termination prover that produces a *lexicographic linear ranking functions* (LLRF) as a certificate of termination. Using LLRFs as certificates, as opposed to TIs, improves the performance of the safety prover in validating certificates. The non-termination prover allows

LLRF-based loop termination provers to reason about loops that cannot be proved terminating when analyzed in isolation. Finally, the safety prover directs the search of the non-termination prover in finding counterexamples to termination. Using this approach, TREX is able to prove termination or non-termination of programs that are outside the reach of existing techniques, including the example in Fig. 1. §2 gives an informal discussion as to how TREX handles this example.

The contributions of this paper are as follows:

1. We present TREX, a novel algorithm for proving termination of whole programs. TREX simultaneously maintains over and under-approximations of a loop to quickly find proofs of termination or non-termination. This allows it to compose several program analyses that until now were disparate: termination provers for multi-path loops, non-termination provers for cycles, and global safety provers.
2. We present an empirical evaluation of TREX. We evaluated TREX by applying it to a set of systems drivers and benchmarks for termination analysis, along with versions both that we injected with faults. The results of our evaluation demonstrate that TREX's use of alternation allows it to quickly prove that programs either always terminate or produce verified counterexamples to their termination.

The rest of this paper is organized as follows. In §2, we illustrate by example how TREX proves termination or non-termination for an example program. In §3, we review known results on which TREX builds. In §4, we give a formal presentation of the TREX algorithm. In §5, we present an empirical evaluation of TREX. In §6 we discuss related work, and in §7 we conclude.

2 Overview

We now informally present the TREX algorithm. We first describe the core algorithm for deciding if a single loop in a single-procedure program terminates under all program inputs, and then illustrate the algorithm using a set of examples. If the program contains nested loops, function calls, and pointers, the algorithm can be extended. We present such extensions in §4.2.

To analyze a loop L, TREX maintains two key pieces of information: (i) a loop invariant O of L, and (ii) U, which is a subset of the set of all paths that can execute in the body of loop L. Note that paths in U can be obtained by concatenating arbitrarily many paths through L. The overapproximation O is initialized to a weak invariant such as $\mathbf{true}$, and U is initialized to an empty set of paths. TREX analyzes each loop iteratively. In each iteration, it first attempts to find a certificate that proves that no set of paths in U can execute together infinitely often, assuming the loop invariant O.

First, suppose that TREX cannot find a proof certificate. Then TREX finds a path τ that is a concatenation of paths in U such that no proof of termination of τ exists. It then uses a *non*-termination prover [14] to derive a loop precondition φ such that if the program reaches L in a state $\sigma \in \varphi$, then it will then execute τ infinitely often. TREX calls a safety prover to determine if some initial program state σ_I can reach such a σ along an execution trace. If so, then the trace, combined with τ, is a witness that the loop does not always terminate. If a safety prover determines that no such states σ_I and

σ exist, then TREX strengthens the over-approximation of O with the knowledge that φ can never hold at the head of the loop L.

Now, suppose that TREX does find a proof certificate for the under-approximation. TREX then checks to see if the certificate is valid for all paths in L. If the certificate is not valid, then TREX finds a path τ over the body of L that invalidates the certificate, and expands U to include τ. TREX then performs another refinement step using the strengthened over-approximation or expanded under-approximation. In this way, the under-approximation U is used to find potentially non-terminating cycles, and if such cycles are unreachable, this information is used to refine the over-approximation O. Dually, if the certificate for U is not a valid certificate for all the paths through L with the over-approximation O, this information is used to expand U. We now illustrate the advantages of this approach using a set of examples.

Alternation Between Over and Under-approximations. Because TREX simultaneously maintains over and under-approximations of a loop, it can often quickly find proofs that the loop terminates, even when the proofs rely on program behavior that is not local to the loop. For example, consider loop `L2` from Fig. 1. Recall from §1 that existing termination provers may have difficulty proving termination of `L2`. A technique that relies on a fixed over-approximation may not be able to discover automatically the needed loop invariant $d > 0$, but a technique that relies solely on under-approximations may struggle to find a proof of termination for the loop, as it is misled by information gathered along a trace leading to the loop.

TREX handles this example by alternating between over and under-approximations. It first tries to prove termination of the loop with an over-approximation that the loop can be reached in any state, and is unable to find such a proof. TREX thus generates a potential counterexample to termination in the form of a cycle through the loop: `assume(x > 0 && y > 0); y := y - d`. It then applies a non-termination prover to this cycle to find a sufficient condition φ such that if execution reaches the loop in a state that satisfies φ, then the subsequent execution will not terminate. The non-termination prover determines that such a sufficient condition is the predicate $d \leq 0$. TREX then queries a safety prover to decide if the condition $d \leq 0$ at `L2` is reachable, but the safety prover determines that $d \leq 0$ is in fact unreachable. Thus TREX refines the over-approximation of the loop to record that all states reachable at line 4 are in $\neg(d \leq 0) \equiv d > 0$. TREX then applies a loop termination prover to the loop under this stronger over-approximation. Such a technique quickly proves that `L2` always terminates.

Using LLRFs As Certificates for Termination Proofs. Existing techniques for proving termination of programs produce a *transition invariant* (TI) as a certificate of proof of termination, while existing termination provers for loops produce *lexicographic linear ranking functions* (LLRF). TREX is parametrized to use either TIs or LLRFs as certificates in proving termination of whole programs. This implies that it can construct a set of LLRFs that serves as a proof of termination for a whole program. While TIs are more expressive than LLRF's in that they can be used to encode proofs of termination for more loops than LLRFs, LLRFs can often be constructed faster, and the loss of expressiveness typically does not matter in practice. We find that in practice, using LLRFs

as certificates instead of TIs results in an acceptable loss of expressiveness while allowing significant gains in performance, both in finding the certificate and in validating candidate certificates.

To gain an intuition for the advantage of using LLRFs, consider again in Fig. 1 the loop `L2`. Recall that `L2` is problematic for an analysis that constructs a TI using under-approximations. However, suppose that an analysis based on constructing TIs was given $d > 0$ as a loop invariant. The analysis could then analyze the loop in isolation and would eventually find a TI that proves termination. However, the best known approach to TI synthesis constructs proofs one at a time for single paths through potentially multiple iterations of the loop. For each path, the analysis then attempts to validate the constructed proof using an expensive safety check. However, if an LLRF-based analysis is given the loop invariant $d > 0$, and both the paths "`x := x - d; y := *; z := z - 1`", and "`y := y - d`" through the loop, it can prove termination of the loop by solving a single linear constraint system. Furthermore, the validation of resulting LLRF is considerably simpler.

```
1    int d = 1;
2    int x;
3
4    if(*) d := d - 1;
5    if(*) foo();
6    ...
7    //k such conditionals
8    //without decrements of d.
9    ...
10   if(*) foo();
11   if(*) d := d - 1;
12
13   while (x > 0) {
14     x := x - d;
15   }
```

Fig. 2. Example to illustrate detecting non-termination.

Proving Non-termination. Finally, TREX can find non-terminating executions efficiently. For the program in Fig. 2, suppose that the function `foo` has p paths through its body. There are thus $O(2^k p^k)$ different lassos in the program that end with the cycle at lines 13–15. Of these, only the lassos with stems that include the decrements to d at lines 4 and 11 lead to non-termination. The current best known technique for finding termination bugs, TNT [14], searches the program for lassos in an arbitrary manner. Thus TNT may only find such a bug by enumerating the entire space of lassos.

TREX can provide TNT with a goal-directed search strategy for finding termination bugs. For the program in Fig. 2, TREX first analyzes the loop at lines 13–15, and is unable to prove termination of the loop. It next attempts to find an execution for which the loop does not terminate. However, instead of applying TNT to one of the *lassos* in the program to verify it as a *complete witness to non-termination*, TREX applies TNT to the sole *path through the loop* to derive a *sufficient condition for non-termination*. For the example, TNT determines that if the loop is reached in a state that satisfies $d < 0$, then execution of the loop will not terminate. TREX then queries a safety prover to determine if a state that satisfies $d < 0$ is reachable at the head of the loop. Suppose that the function `foo` does not modify d. Modular safety checkers such as SMASH [11] can use knowledge about the target set of states $d < 0$ to build a safety summary for `foo` which states that d is not modified by `foo`. TREX uses such a prover to quickly find a path that reaches the loop head in a state that satisfies $d < 0$. It is the path that decrements d at lines 4 and 11.

3 Preliminaries

TREX builds on existing work on proving termination and non-termination. We recall some preliminaries and definitions from previous work.

3.1 Termination Certificates

TREX is parametrized by the certificates that it uses to prove termination of individual loops. A certificate typically defines a measure μ that is bounded below by zero, (i.e. $\mu \geq 0$) and decreases on every iteration of the loop. Previous work shows how to find such measures automatically using lexicographic linear ranking functions and transition invariants. The exact details of these certificates are not important for an understanding of TREX, but for the sake of completeness, their definitions are given in Appendix A.

3.2 Proving Non-Termination

Recent work [14] addresses a dual problem to proving termination, that of proving *non*-termination of a given path through a program. Let a pair of paths $(\tau_{stem}, \tau_{cycle})$ be a *lasso*. The problem of proving non-termination is to determine if it is possible for τ_{stem} to execute once followed by infinite consecutive executions of τ_{cycle}. [14] establishes that $(\tau_{stem}, \tau_{cycle})$ is non-terminating if and only if there exists a *recurrent set* of states defined as follows:

Defn. 1 *For a lasso* $(\tau_{stem}, \tau_{cycle})$*, a* recurrent set φ *is a set of states such that* (i) φ *is reachable from the beginning of the program over* τ_{stem}*; and* (ii) *For every state* $\sigma \in \varphi$*, there is a state* $\sigma' \in \varphi$ *such that* σ' *can be reached from* σ *by executing* τ_{cycle}*.*

In this work, we introduce the notion of a *partial recurrent set*, which is a relaxation of a recurrent set.

Defn. 2 *A set of states* φ *is a* partial recurrent set *for a sequence of statements* τ *if it satisfies clause (ii) of Defn. 1, with* τ *in place of* τ_{cycle}*.*

One can reduce the problem of finding a recurrent set for a given lasso to solving a non-linear constraint system [14]. This is the approach implemented by TNT. The TNT technique relies on a constraint template to guide the constraint solving, and gives a heuristic for iteratively refining the template until a recurrent set is found. In practice, if a recurrent set exists, then it typically can be found with a relatively small template. TNT can be easily extended to find a partial recurrent set as well.

4 Algorithm

We now formally present the TREX algorithm, given in Fig. 3. We first describe TREX for single-procedure programs without pointers, function calls, or nested loops. We describe in §4.2 an enhancement of TREX that deals with pointers, function calls, and nested loops. TREX attempts to prove termination or non-termination of each loop

TREX (P)
Input: Program P
Returns: $\mathbf{Termination}$ if P terminates on all inputs,
$\mathbf{NonTermination}(\tau_{stem}, \tau_{cycle})$ if P may execute τ_{stem} once, and then execute τ_{cycle} infinitely many times.

```
 1: for each loop L in the program do
 2:   O := true // Initialize over-approximation.
 3:   U := { } // Initialize under-approximation.
 4:
 5:   loop
 6:     result := GetCertificate(O, U)
 7:     if (result = Termination(C)) then
 8:       result' := CheckValidity(C, O, L)
 9:       if (result' = Valid) then
10:         break // Analyze next program loop.
11:       else if (result' = Invalid(τ)) then
12:         U = U ∪ {τ}
13:         continue
14:       end if
15:     else if (result = Cycle(τ_cycle)) then
16:       φ = PRS(τ_cycle)
17:       if Reachable(φ) then
18:         τ_stem := SafetyTrace(φ)
19:         return NonTermination(τ_stem, τ_cycle)
20:       else
21:         O := O \ φ
22:         continue
23:       end if
24:     end if
25:   end loop
26: end for
```

Fig. 3. The TREX algorithm

in isolation. When TREX analyzes each loop L, it maintains an over-approximation O, which is a superset of the set of states reachable at the loop head of L, and an under-approximation U, which is a subset of the paths through the loop body. At lines 2 and 3, O is initialized to $\mathbf{true}$ (denoting all states), and U is initialized to the empty set of program paths. We use L_O to denote the loop L with each path prefixed with an assumption that O holds, and similarly for U_O.

The core of the TREX algorithm iterates through the loop in lines 5-25 of Fig. 3. Inside this loop, TREX refines the over-approximation O to smaller sets of states, adds more paths to the under-approximation U, and tries to prove either termination or non-termination of the loop L. At line 6, TREX calls $GetCertificate$ to find a certificate of proof for the under-approximation U.

First, suppose that the call $GetCertificate(O, U)$ returns $\textbf{Termination}(C)$. In this case, $GetCertificate$ has found a proof C that no set of paths in U execute together infinitely often under invariant O. In this case, TREX checks if C is a valid certificate for the entire loop L_O by calling the function $CheckValidity$ in line 8. The call $CheckValidity(C, O, L)$ returns **Valid** if the certificate C is a valid proof of termination for the loop L_O. In this case, TREX determines that L terminates, and analyzes the next loop. Otherwise, $CheckValidity$ returns $\textbf{Invalid}(\tau)$, where $\tau \in L^+ \setminus U$ is a path such that C does not prove that a cycle of τ will not execute infinitely often. In this case, TREX adds the path τ to the under-approximation U and continues to iterate.

Now suppose that $GetCertificate$ does not find a certificate for U_O and returns $\textbf{Cycle}(\tau_{cycle})$. Here, $\tau_{cycle} \in U^+$ is a trace formed by concatenating some sequence of paths through U. At line 16, TREX calls PRS, which computes for τ_{cycle} a partial recurrent set φ. If $\sigma_J \in \varphi$, then executing τ_{cycle} from σ_J results in a state $\sigma_F \in \varphi$. Thus if φ is reachable from a program input σ_I, then program P will not terminate on σ_I. On line 17, TREX calls a safety prover to determine if such a σ_I exists. If so, then the safety prover produces a trace τ_{stem} along with an initial state that reaches φ. TREX then presents the lasso $(\tau_{stem}, \tau_{cycle})$ as a true counterexample to termination. Otherwise, suppose the safety prover determines that φ is unreachable. In that case, TREX refines the over-approximation O by removing from O the set of states φ. It then continues to iterate.

4.1 Sub-procedures Called by TREX

```
1   //x is an input variable
2   int x;
3
4   int main() {
5     while (x > 0) {
6        if(*) foo();
7        else foo();
8     }
9   }
10
11  void foo() {
12    x--;
13  }
```

Fig. 4. Example illustrating inter-procedural analysis.

The TREX algorithm as presented in Fig. 3 depends on four procedures: $Reachable$, $CheckValidity$, $GetCertificate$, and PRS. Definitions of $Reachable$ and PRS are standard. $Reachable$ answers a safety query for a program, and thus can be implemented using any static analysis tool or model checker that provides either a proof of safety or counterexample trace. Our implementation answers such queries by using the SMASH algorithm [11]. PRS constructs a partial recurrent set for an execution trace. The implementation of such a procedure used by our implementation of TREX is described in [14].

Procedures $GetCertificate$, and $CheckValidity$ can be instantiated to compute and validate any certificate of a termination proof, such as TIs or LLRFs. The work in [9] gives instantiations of these procedures for TIs. If the procedures are instantiated to use TIs, then the resulting version of TREX is similar to TERMINATOR, modulo the fact that TREX uses counterexamples to refine an over-approximation of each loop, while TERMINATOR does not attempt to maintain an over-approximation. Furthermore, TREX can be instantiated to use LLRFs to reason about programs, given suitable definitions of $GetCertificate$ and $CheckValidity$. In Appendix B.2, we give novel implementations of such functions.

4.2 Handling nested loops, function calls and pointers

For TREX to reason about nested loops, function calls, and pointers, it is necessary that its sub-procedures reason about these features. The procedures $\mathit{Reachable}$ and $\mathit{CheckValidity}$ depend primarily on a safety prover. In the context of safety, handling nested loops and function calls is a well-studied problem, and our safety checker supports such features. However, the procedures $\mathit{GetCertificate}$ and PRS must be extended from their standard definitions to handle such features. Both procedures take as input a finite set of paths. The current state-of-the-art techniques for implementing $\mathit{GetCertificate}$ and PRS can only reason about paths defined over a fixed set of variables and linear updates to those variables. They cannot reason about program statements that manipulate pointers, because pointer dereferences introduce non-linear behavior. Thus to apply such techniques, an analysis must first rewrite program paths that perform pointer manipulations to a semantically equivalent form expressed purely in terms of linear updates.

TREX rewrites program paths to satisfy this condition by following a strategy used in symbolic-execution tools, and also by TERMINATOR, which is to *concretize* the values of pointers. Note that all paths added to U are produced by $\mathit{CheckValidity}$, which takes as input an entire program, as opposed to a single loop. Thus if $\mathit{CheckValidity}$ determines that a certificate is not valid for an entire loop L, then it produces a counterexample in the form of a lasso $(\tau_{stem}, \tau_{cycle})$, where τ_{cycle} is a path through the loop and τ_{stem} is a path up to the loop. In the absence of pointer dereferences, function calls, or nested loops, τ_{cycle} is directly added to U. In the presence of pointer dereferences, TREX rewrites the cycle before adding it to U as follows: for an instruction `*p = *q + 5` where `p` and `q` point to scalar variables `x` and `y` respectively during the execution of τ_{stem}, TREX replaces the instruction with `x = y + 5`. This amounts to under-approximating the behavior of paths through a loop by assuming that the aliasing conditions of τ_{stem} hold in every iteration of the loop.

TREX reasons about function calls and nested loops by in-lining instructions along the path τ_{cycle} before adding the path to U. For example, suppose that we apply TREX to the program in Fig. 4. In the course of analysis, TREX expands an under-approximation of the loop in lines 5–8 by adding a path through the loop, which goes through the function `foo`. To find a certificate for a new proof of termination that includes this path, TREX applies $\mathit{GetCertificate}$ to this path, which only looks at the instructions in the path: `assume(x > 0); x = x - 1`. $\mathit{GetCertificate}$ produces an LLRF `x`. TREX then applies $\mathit{CheckValidity}$, which uses an interprocedural safety analysis to verify that `x` is indeed a ranking function for the entire loop, i.e., in all executions of the program, the value of `x` decreases on every iteration of the loop.

4.3 Limitations of TREX

If TREX terminates, then it produces a proof of termination or a valid counterexample that witnesses non-termination. However, TREX may not terminate for the following reasons: (i) the underlying safety prover or non-termination prover may not terminate; or (ii) the main loop in Fig. 3 lines 5–25 may not terminate. The main loop may not terminate because finding the termination proof or non-termination witness may require

TREX to reason about program features beyond what are supported by the loop termination and non-termination provers used by TREX. Such program features include non-linear arithmetic or manipulating recursive data-structures. Proving termination in the latter case is addressed in [3]. It would be interesting to instantiate TREX with the prover presented in [3], provided that a corresponding non-termination prover could be derived.

5 Experiments

We empirically evaluated TREX over a set of experiments designed to determine if:

- TREX can prove termination and find bugs for programs explicitly designed to be difficult to analyze for termination. To this end, we applied TREX to several handcrafted benchmarks.
- TREX can prove termination and find bugs for real-world programs. To this end, we applied TREX to several drivers for the Windows Vista operating system.

To evaluate TREX, we implemented the algorithm described in §4, instantiated with the LLRF-based termination prover described in Appendix B.2 and the non-termination prover described in §3.2. We also compared TREX with the current state of the art in proving termination. The only other termination prover that we are aware of that can analyze arbitrary C programs is TERMINATOR. We did not have access to the implementation of TERMINATOR discussed in [10], so we reimplemented it using the description provided in that work. We refer to this implementation as R-TERMINATOR. To allow for a fair comparison, the implementations of both TREX and R-TERMINATOR use the same safety prover, SMASH [11]. All experiments were performed on a machine with an AMD Athlon 2.2 GHz processor and 2GB RAM.

5.1 Micro-benchmarks

```
int x,d,z;
d=0; z=0;

while(x > 0) {
   z ++;
   x = x - d;
}
```

Fig. 5. A non-terminating loop.

We first evaluated if TREX could find difficult termination bugs in small program snippets. To do so, we first applied R-TERMINATOR and TREX to the loop shown in Fig. 5, based on the program in Fig. 1. R-TERMINATOR did not find the bug in this loop: as described in §1, it successively tries as proofs of termination ranking functions $c_i - z$ for different constants c_i. TREX found this bug within 5 seconds, requiring 1 alternation. This example thus indicates that for a non-terminating loop with variables spurious to proving termination, z in Fig. 1, the spurious variables can cause R-TERMINATOR not to find a proof of termination or non-termination.

Next, we applied TREX and R-TERMINATOR on snippets of code extracted from real Windows Vista drivers, the same used in [2]. The results of the experiments are given in Tab. 1. For each driver snippet, Tab. 1 reports the number of loops, the number of buggy (non-terminating) loops, the number of times that TREX called a non-termination prover during analysis (#NT), the number of times TREX called a termination prover (#TC), the

Name	Num Loops	Buggy Loops	TREX			R-TERMINATOR		TREX
			#NT	#TC	Time (s)	#TC	Time (s)	speedup
01	3	0	0	3	13.8	4	32.1	2.3
02	3	1	1	2	15.3	5	48.0	3.1
03	1	1	1	0	7.9	1	5.9	0.7
04	1	0	0	1	3.1	1	12.3	3.9
05	1	0	0	1	6.4	1	8.8	1.4
06	1	0	0	1	3.0	2	13.8	4.6
07	2	0	0	2	10.2	2	11.8	1.2
08	2	0	0	2	9.4	2	11.0	1.2
09	2	1	–	–	T/O	–	T/O	–
10	1	0	0	1	2.5	2	10.3	4.1

Table 1. Results of applying TREX to Windows drivers snippets. The timeout (T/O) limit was set to 500 seconds.

Name	Num Loops	Buggy Loops	TREX			R-TERMINATOR	
			# NT	# TCs	Time (s)	# TCs	Time (s)
01	3	0	0	3	22.3	3	19.9
04	1	1	1	0	4.9	1	5.4
05	1	1	1	0	7.1	1	9.1
06	1	1	1	1	9.7	2	12.1
07	2	0	0	2	7.6	2	9.8
08	2	1	1	1	8.1	1	7.4
10	1	1	1	0	9.8	0	4.4

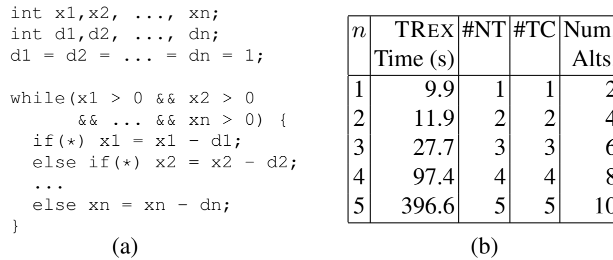
Table 2. Results of experiments over driver snippets modified to contain termination bugs.

time taken by TREX, and similarly for R-TERMINATOR. In general, TREX was significantly faster than R-TERMINATOR. In most cases, the speedup was caused directly by the fact that TREX uses LLRF's as termination certificates, whereas R-TERMINATOR uses TI's. By using LLRF's, TREX needs to construct fewer certificates during analysis, and thus needs to query a safety prover fewer times in order to validate certificates.

For these programs, TREX called its non-termination prover at most once. In each case, the call verified that the loop is indeed non-terminating. Program "02" highlights the advantage of applying a non-termination prover in this way. When analyzing program "02," R-TERMINATOR constructed and failed to validate multiple candidate termination certificates obtained by under-approximating the behavior of cycles. R-TERMINATOR eventually could not construct a new candidate and reported a possible termination bug. When applied to program "02," TREX failed to find a proof of termination, but then immediately alternated to apply a non-termination prover, which quickly found a verified termination bug. Finally, note that program "09" has a complicated loop about which neither TREX nor R-TERMINATOR can find a proof, and thus time out.

The original driver snippets contain relatively few termination bugs. Thus to further measure TREX's ability to find bugs, we modified each driver snippet that had no termination bug as follows. We introduced variables "inc_1, inc_2, ...", and code that non-deterministically initializes them to 0 or 1. We then replaced increment or decrement statements of the form "x = x $\pm$ 1", with "x = x $\pm$ inc_n", where a different "n" is used for each increment statement. The results are given in Table 2. Note that our modification did not always introduce a termination bug, as in some cases, the increment or decrement was irrelevant to the termination argument for the loop.

In general, TREX and R-TERMINATOR analyze these loops in similar amounts of time. In cases where TREX completed in less time than R-TERMINATOR, it was typically because R-TERMINATOR produced and then failed to validate more candidate termination certificates. In such cases, R-TERMINATOR would typically choose as a ranking function a variable “x”, where a statement such as “x = x - 1” had been modified to “x = x - inc” and “inc” was initialized to 1 on some but not all paths through the loop. R-TERMINATOR would only discover later in its analysis, after an expensive safety query, that “x” need not always decrease. In contrast, TREX did not choose “x” as a ranking function in this case because it never considers the concrete values of the “inc” variables while trying to find a ranking function. We believe that the difference in performance between TREX and R-TERMINATOR would increase for when applied to larger programs containing bugs as described above. This is because it typically takes less time to answer a non-termination query than it does safety query, as the former is a local property of a loop while the latter is a global property of a program.



```
int x1,x2, ..., xn;
int d1,d2, ..., dn;
d1 = d2 = ... = dn = 1;

while(x1 > 0 && x2 > 0
      && ... && xn > 0) {
  if(*) x1 = x1 - d1;
  else if(*) x2 = x2 - d2;
  ...
  else xn = xn - dn;
}
```

(a)

n	TREX Time (s)	#NT	#TC	Num. Alts.
1	9.9	1	1	2
2	11.9	2	2	4
3	27.7	3	3	6
4	97.4	4	4	8
5	396.6	5	5	10

(b)

Fig. 6. (a) A family of loops requiring significant alternation to analyze. (b) TREX results.

A Micro-benchmark Forcing Alternation We evaluated the performance of TREX when analyzing loops for which multiple alternations are required to find a proof of termination or a bug. Consider the class of loops defined in Fig. 6. Each value of n defines a loop. To prove such a loop terminating, TREX must perform $2n$ alternations between searching for an LLRF to prove the loop terminating and searching for a PRS to prove the loop non-terminating. The results of applying TREX to the loops defined by $n \in [1, 5]$ are given in Fig. 6(b). TREX found a proof of termination in each case. The results indicate that alternation between the LLRF search and PRS search scales quite well for up to 6 alternations, but that performance begins to degrade rapidly when the analysis requires more than 8 alternations. In practice, this is not an issue, as most loops require less than 3 alternations to analyze.

We also applied R-TERMINATOR to these programs, but R-TERMINATOR timed out in each case. In its analysis, R-TERMINATOR under-approximates cycles in order to produce the x_i as candidates for proofs of termination. However, when R-TERMINATOR applies a safety prover to validate these candidates, the safety prover does not terminate. This is because the safety prover, based on predicate abstraction, uses a weakest precondition operator to find predicates relevant to its analysis. In the

safety queries made by R-TERMINATOR, these predicates are not sufficient: the safety prover needs an additional predicate $d_i > 0$ to establish that some x_i decreases each time through the loop. In contrast, TREX uses a non-termination prover to find that $d_i \leq 0$, and thus establishes that $d_i > 0$ as a loop invariant. Thus when TREX makes a subsequent call to the safety prover, the call terminates.

We evaluated TREX's ability to find bugs for such a loop. For the loop defined by $n = 5$, we injected a fault that initialized $d_3 = 0$. For this loop, TREX found the resulting termination bug using 5 alternations in 22.2 seconds.

Name	LOC	#Loops	TREX Time (s)	R-TERMINATOR Time (s)
Driver-1	0.8K	2	80	85
Driver-2	2.3K	4	1128	2400
Driver-3	3.0K	10	54	120
Driver-4	5.3K	17	945	T/O
Driver-5	6.0K	24	24	T/O
Driver-6	6.5K	16	68	62

Table 3. Results of experiments over Windows Drivers. Time out was set to 1 hour.

5.2 Windows Drivers

We applied TREX to complete Windows Drivers to evaluate its ability to analyze programs of moderate size that manipulate pointers and contain multiple procedures. The drivers were chosen randomly from the Microsoft's Static Driver Verifier regression suite. We could not directly compare TREX to R-TERMINATOR over the drivers used in [10], as these were not available. The results of the evaluation are given in Table 3. The drivers used are well-tested, and thus we did not find any bugs in them. However, the results show that TREX is faster than R-TERMINATOR in most cases. Similar to the micro-benchmarks presented in §5.1, this is because R-TERMINATOR produced many more termination certificates, resulting in more safety queries.

6 Related Work

TREX brings together threads of work in proving termination that were disparate up to now. Our work shares the most in common with TERMINATOR [10]. TERMINATOR iteratively reasons about under-approximations of a program to construct a proof of termination. TREX simultaneously refines under and over-approximations of a program.

TREX relies on an analysis that proves termination of loops represented as a set of guarded linear transformations of program variables. Many existing techniques prove termination of such loops by constructing linear ranking functions [2, 4–7, 17]. Such techniques are efficient, but can only be applied to a restricted class of loops and cannot reason about the contexts in which loops execute. In this work, we show how all such techniques can be brought be bear in analyzing general programs, provided they can be extended to generate counterexample traces on failure. In Appendix B.2, we describe how to do this by extending a simplification of the technique of [4]. The technique of

[4] additionally uses constraint solving to find efficiently a loop invariant that is used to prove termination. For simplicity, we do not consider how to extend a technique that generates invariants, though in principle the same extension should apply to such a technique.

TREX also relies on techniques that prove that a given lasso does not terminate [14]. TREX applies such a technique to simultaneously search for counterexamples to termination and to guide the search for a proof of termination. TREX can also be used as a search strategy for finding non-termination bugs. The search strategy proposed in [14] simply enumerates potential cycles using symbolic execution. [8] gives a method for deriving a sufficient precondition for a loop to terminate. However, this approach does not lend well to refinement if the computed precondition is not met. TREX applies [14] iteratively to derive a sufficient precondition for termination that is guaranteed to be a true precondition of a loop.

Multiple safety provers [1, 11, 13] demonstrate that alternating between over and under approximations is more effective for proving safety properties than an analysis based exclusively on one or the other. For these provers, an over-approximation of the program is an abstraction of its transition relation, and the under-approximation is a set of tests through the program. The abstraction directs test generation, while the tests guide which parts of the abstraction are refined. TREX demonstrates that the insight of maintaining over and under approximations can be applied to prove termination properties of programs as well. However, for TREX, the over-approximation maintained is an invariant for a loop under analysis, and the under-approximation is a set of concrete paths through the loop. The invariant directs what new paths through the loop are considered, and the concrete paths guide the refinement of the loop invariant.

7 Conclusion

Safety provers that simultaneously refine under and over-approximations of a program can often prove safety properties of programs effectively. In this work, we have shown that the same refinement scheme can be applied to prove termination properties of programs. We derived an analysis based on this principle, implemented it, and applied it to a set of termination analysis benchmarks and real-world systems code. Our results demonstrate that alternation between approximations significantly improves the effectiveness and performance of termination analysis.

References

1. Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In *ISSTA*, pages 3–14, 2008.
2. Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Variance analyses from invariance analyses. *SIGPLAN Not.*, 42(1):211–224, 2007.
3. Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, pages 386–400, 2006.
4. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *CAV*, 2005.

5. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *ICALP*, pages 1349–1361, 2005.
6. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In *CONCUR*, pages 488–502, 2005.
7. Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking abstractions. In *ESOP '08*, pages 148–162, 2008.
8. Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving conditional termination. In *CAV*, pages 328–340, 2008.
9. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *SAS*, pages 87–101, 2005.
10. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.
11. Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, 2010.
12. Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, pages 16–28, 2009.
13. Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: a new algorithm for property checking. In *FSE*, pages 117–127, 2006.
14. Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *POPL*, pages 147–158, 2008.
15. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 465–486, 2004.
16. Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS*, pages 32–41, 2004.
17. Ashish Tiwari. Termination of linear programs. In *CAV*, pages 70–82, 2004.

A Termination Certificates

The choice of certificate can greatly affect the expressiveness and performance of the resulting termination prover. Here, we give a basic background on certificates defined in previous work, along with current state of the art techniques for their synthesis.

Linear Ranking Functions In general, a *ranking function* f is a function from a domain D with relation $<$ to another domain D' that is well-founded with respect to a relation $<'$, where f is monotone with respect to $<$ and $<'$. The existence of such a function f is a sufficient proof that D itself is well-founded. In the case where D represents the state space of a program and $<$ represents its transition relation, a ranking function is thus a proof that the program terminates on all inputs. For exactly every program that terminates on all inputs, there is a ranking function, so the problem of constructing a ranking function for a program is undecidable in general. However, there has been much work on techniques for efficiently generating ranking functions of restricted classes. One such class is that of *linear ranking functions*.

Defn. 3 *For a loop L with cutpoint γ (such as the head of L), a linear ranking function r is a linear expression over the program variables such that:*

– *r has a constant lower bound, assumed without loss of generality to be 0, every time that execution enters the body of L from γ.*
– *Between any two successive visits to γ over the body of L, the value of r decreases.*

The existence of such an expression r is proof that the body of L cannot be executed infinitely often without execution leaving L.

```
1  while (x >= 0
2        && y >= 0) {
3    if (*) {
4      x--;
5    } else {
6      y--;
7    }
8  }
```

Fig. 7. A loop with a linear ranking function.

Ex. 1 *Consider the loop in Figure 7 with the loop head serving as the cutpoint. The guard $x \geq 0 \wedge y \geq 0$ implies that $x + y$ is bounded from below by 0 whenever it enters the loop. Additionally, it can be shown that every path once through the loop from the cutpoint back to itself decreases the expression $x + y$. Thus $x + y$ is a linear ranking function for the loop.*

Linear ranking functions are a particularly interesting and useful class of ranking functions. Given a loop represented as a set of guarded linear updates of the program state, a variety of techniques exist for finding a linear ranking function for the loop. Such techniques are based on linear constraint solving, and thus are very efficient in practice [15].

Lexicographic Linear Ranking Functions Linear ranking functions typically can be found quickly when they exist, but there are many terminating loops that do not have linear ranking functions. Loop `L2` in Fig. 1 with the precondition $d > 0$ is such a loop. However, linear ranking functions can be generalized to *lexicographic linear ranking functions*, which can prove termination for a much more general class of loops, including the loop `L2`.

Defn. 4 *For a loop L with cutpoint γ, a* lexicogrpahic linear ranking function *(LLRF) is an ordered tuple $(r_1, r_2, \ldots, r_n)$ of linear expressions over the program variables such that for each path p through the loop, there exists some r_i such that:*

– *r_i has a constant lower bound for executions of p.*
– *r_i decreases over every execution of p.*
– *For $j < i$, expression r_j is non-increasing over the execution of p.*

Observe that for $k > i$, it need not be the case that r_k is decreasing or even non-increasing over an execution of p. The existence of an LLRF is proof that L cannot execute infinitely often without execution exiting L.

Ex. 2 *Consider loop* `L2` *in Fig. 1. Although y may not strictly decrease over each path through the loop,* `P1` *can only cause it to increase a finite number of times, as* `P1` *has x as a ranking function. Thus* `L` *has lexicographic linear ranking function $\langle x, y \rangle$.*

LLRFs are useful proof objects, as they are strictly more powerful than linear ranking functions, yet can be found almost as quickly using techniques based on linear constraint-solving. In considering the two proof objects, we primarily focus on lexicographic linear ranking functions and treat linear ranking functions as a special case.

Transition Invariants Transition invariants [16] are equivalent in expressive power to general ranking functions.

Defn. 5 *Suppose that a loop has transition relation τ, with transitive closure of denoted as τ^+. A* transition invariant *of the loop is a binary relation R over pre and post states such that $\tau^+ \subseteq R$. If R can be expressed as a finite union of well-founded relations, then R is* disjunctively well-founded*.*

A loop terminates if and only if there exists a disjunctively well-founded transition invariant for a loop [16]. [9] gives a technique to find a disjunctively well-founded transition invariant for a program by maintaining a disjunctively well-founded relation, iteratively adding well-founded disjuncts until the relation over-approximates the program or cannot be expanded further (in which case, termination is not proven). While general, this technique is often not as efficient as techniques that synthesize linear ranking functions in cases where both can be applied. Intuitively, this is due to the fact that the technique does not immediately construct a proof of termination. Rather, it constructs a candidate proof, and then uses a potentially expensive safety check that must reason about the transitive closure of the transition relation of the loop to validate the candidate.

B Instantiations of TREX

In this section, we describe two instantiations of TREX, one using transition invariants, and the other using LLRFs.

B.1 TREX Instantiated with Transition Invariants

To instantiate TREX with transition invariants, we instantiate the function $\mathit{GetCertificate}(O, L)$ to a procedure that finds a transition-invariant based proof of termination for each individual path through L_O using a technique based on linear constraint solving. $\mathit{CheckValidity}(C, O, L)$ checks if the binary relation C is an over-approximation of L_O, and thus a transition invariant. Methods for performing both tasks are described in [9].

B.2 Instantiating TREX with LLRFs

We now describe an instantiation of TREX using LLRFs as certificates for proofs of termination. Recall that compared to TIs, LLRFs are limited in terms of expressiveness, but can often by synthesized faster. Our experience instantiating TREX with LLRFs indicates that the loss in expressiveness typically is not an issue in practice. We now give definitions of the TREX functions that define its instantiation for LLRFs. We first give an intuitive argument that when an LLRF exists for a given loop, TREX will find it in less time than it would take to find a TI for the same loop.

Motivation for using LLRFs Recall the role of termination proof certificates in the TREX algorithm. At each iteration, TREX first calls $GetCertificate$ on an under-approximation U of a loop L. Assuming $GetCertificate$ finds a certificate C that no set of paths in U executes infinitely often, then TREX calls $CheckValidity$ to determine if C is a proof of termination for L. TREX achieves optimal performance with regard to these functions if it is instantiated to manipulate a proof certificate such that (1) the *number* of candidate certificates required to be synthesized is minimized and (2) the complexity of validating each certificate is minimized.

The following argument gives an intuition that during the analysis of a given loop, TREX instantiated with LLRF will need to generate no more proof certificates than TREX instantiated with TI, and will likely need to generate fewer. Suppose that an underapproximation U of a loop L is represented as a set of paths $\{p_1, p_2, \ldots, p_n\}$. To construct a candidate TI for U, the TI synthesizer described in [9] first constructs a well-founded binary relation for each path p_i. This relation that shows that p_i cannot execute infinitely often on its own. The synthesizer then combines the relations to form a proof that any path in the set $(p_1^+ \mid p_2^+ \mid \ldots \mid p_n^+) \subset \mathsf{Paths}(U)$ cannot execute infinitely often. Observe that the resulting TI may not prove that all interleavings of paths in U do not execute infinitely often.

However, given a set of paths U, an LLRF synthesizer as described in [5] considers at once all possible combinations of all paths through U. As a result, an LLRF is a proof that no subset of paths in the set $(p_1 \mid p_2 \mid \ldots \mid p_n)^+$ can execute infinitely often together. Thus with each synthesis, an LLRF synthesizer proves termination for more paths through L than a TI synthesizer.

The second factor in determining usefulness of an LLRF instantiation compared to a TI instantiation is the complexity of validating an LLRF compared to that of validating a TI. First, consider the process of validating a binary relation R as a TI for a loop L. Let τ be the transition relation of the body of loop L. Then the task of validating R is equivalent to deciding if $\tau^+ \subseteq R$, a *binary reachability problem* [9]. TERMINATOR casts this as a safety problem by performing the code transformation in Fig. 8 and verifying that the assertion already holds. Note that the each time through the loop, the state of the program relevant to the candidate TI is non-deterministically copied to prime-state variables. This non-determinism can greatly increase the complexity of the resulting safety query.

In contrast to a TI, an LLRF makes an assertion about how program expressions are affected by *one step* through the body of a loop. Thus a safety query that validates a candidate LLRF need only validate an instrumentation of one step through the body. The

```
                          while (cond) {
                            if (nondet()) {
                              assert(r1 < r1'
                                     || r2 < r2'
                                     || ...
while (cond) {                       || rn < rn');
  body;             ⇒         v_0' = v_0;
}                             v_1' = v_1;
                              ...
                              v_m' = v_m;
                            }
                            body;
                          }
```

Fig. 8. Program transformation to validate a TI.

```
                          while (cond) {
                            assert(r1 < r1'
                                   || (r1 <= r1' && r2 < r2)
                                   || ...);
while (cond) {              v_0' = v_0;
  body;             ⇒       v_1' = v_1;
}                           ...
                            v_m' = v_m;
                            body;
                          }
```

Fig. 9. Program transformation to validate an LLRF.

exact instrumentation is given in Fig. 9. While similar to the program transformation of Fig. 8, there is a syntactically small but crucial difference: the state of all variables in the candidate LLRF are deterministically copied to the primed state on *each* iteration through the loop. This difference can have a significant impact on the performance of a safety prover when validating the LLRF. Note that checking for the validity of the assertion in Fig. 9 is an implementation of $\mathit{CheckValidity}$.

Implementation of an Instantiation to LLRFs Appendix B.2 discusses the advantages of instantiating TREX to compute certificates in the form of LLRFs. Furthermore, Appendix B.2 gives an implementation of $\mathit{CheckValidity}$ for LLRFs. All that remains to instantiate TREX to use LLRFs as its termination proof certificate is an implementation of $\mathit{GetCertificate}$ for LLRFs. We now present such a procedure $\mathit{GetLLRF}$ in Fig. 10.

Recall that TREX requires two properties of $\mathit{GetLLRF}$ for it to act as an implementation of $\mathit{GetCertificate}$. First, if $\mathit{GetLLRF}$ presents a candidate termination proof, then the candidate must be in the form of an LLRF. Second, if $\mathit{GetLLRF}$ cannot compute a proof of termination, then it must present a potential counterexample to termination in the form of a single trace that may execute through the loop infinitely often. There are known techniques that synthesize LLRFs for loops [4]. However, previous work does not describe how to extract a potential counterexample in the case that termination cannot be proved. We present such an LLRF synthesizer $\mathit{GetLLRF}$ in Fig. 10.

At a high level, $\mathit{GetLLRF}$ determines for each path $p \in L$ what sets of other paths in L must not execute infinitely often in order for p not to execute infinitely often. If these dependencies among paths imply that no path can execute infinitely often, then $\mathit{GetLLRF}$ constructs an LLRF from the dependency relation. Otherwise, $\mathit{GetLLRF}$ uses the dependency relation to find a counterexample.

The definition of $\mathit{GetLLRF}$ is given in Fig. 10. $\mathit{GetLLRF}$ assumes that the input loop L is represented as a finite set of paths, each represented as a guarded linear transformation over program variables. For each path $p_i \in L$, the loop over lines 3 - 19 determines what paths must not execute infinitely often if p_i is not to execute infinitely often. $\mathit{GetLLRF}$ encodes the dependency in a propositional formula ψ_{deps}. ψ_{deps} is defined over a set of propositional variables, with one variable for every path through the loop. For path p, the truth of variable v_p is interpreted as "path p must execute only a finite number of times." For example, the formula $((v_{p_1} \lor v_{p_2}) \land v_{p_4}) \Rightarrow v_{p_3}$ has the intuitive meaning "path p_3 executes a finite number of times if one of the paths p_1 or p_2 execute a finite number of times and path p_4 executes a finite number of times."

For path p_i, the formula ψ_{deps} is constructed as follows. $\mathit{GetLLRF}$ first applies a function $\mathit{RankFinder}$ to p_i, which produces a system of inequalities whose solutions correspond to ranking functions of p_i [15]. If the system is unsatisfiable, then $\mathit{GetLLRF}$ immediately returns p_i as a counterexample to termination. If the system is satisfiable, then for each path p_j, $\mathit{GetLLRF}$ applies $\mathit{NonIncFinder}(p_j)$, which computes a system of constraints whose solutions are all non-increasing expressions over the execution of p_j [15]. $\mathit{GetLLRF}$ then attempts to solve the conjunction of $\mathit{RankFinder}(p_i)$ and $\mathit{NonIncFinder}(p_i)$ to find an expression that is bounded and decreasing over p_i and non-increasing over all other paths through L, and thus a ranking function for p_i.

$GetLLRF(L)$
Inputs: A loop L represented as a set of paths.
Returns: $\mathbf{Termination}(C)$ where C is an LLRF
or $\mathbf{NonTermination}(\sigma)$ where σ is a set of paths
that may execute infinitely often

```
 1: let L = {p_1, p_2, ..., p_n}
 2: ψ_pre := true
 3: for each p_i ∈ P do
 4:    ψ_deps := true
 5:    C(i) := RankFinder(p_i)
 6:    if Unsatisfiable(C(i)) then
 7:       return(NonTermination(p_i))
 8:    end if
 9:    for each p_j ∈ Paths such that j ≠ i do
10:       C(j) := NonIncFinder(p_j)
11:    end for
12:    if Unsatisfiable(⋀_{j=1}^{n} C(j)) then
13:       Ω := GetCompleteUnsatCores(C, i)
14:       for each Q ∈ Ω do
15:          ψ_deps := ψ_deps ∧ (⋁_{q∈Q} v_q)
16:       end for
17:    end if
18:    ψ_pre := ψ_pre ∧ (ψ_deps ⇒ v_{p_i})
19: end for
20: Θ := ψ_pre ⟹ ⋀_{i=1}^{n} v_{p_i}
21: if IsValid(Θ) then
22:    llrf := ConstructLLRF(ψ_pre)
23:    return(Termination(llrf))
24: else
25:    for each α ∈ GetSatAssignment(¬Θ) do
26:       P' := ⋃_{α(v_p)=False} p
27:       if GetLLRF(P') = NonTermination(τ) then
28:          return(NonTermination(τ))
29:       end if
30:    end for
31:    let llrf := ConsLLRFFromSubCases(L)
32:    return(Termination(llrf))
33: end if
```

Fig. 10. The $GetLLRF$ algorithm

Let $C(i)$ denote the set of all constraints requiring an expressions to be bounded and decreasing over p_i, and let $C(j)$ for $j \neq i$ be the set of constraints containing the expression to be non-increasing over p_j. Consider the significance of an *unsatisfiable core* of this conjunction. Each constraint in an unsatisfiable core is contributed by some $C(k)$. However, if it can be proved that path p_k cannot execute infinitely often, then p_k cannot cause p_i to execute infinitely often. In this case, the constraints in p_k can be safely ignored. Formally, let an *unsatisfiable path core* for path p_i be a set of paths $Q \subseteq L \setminus \{p_i\}$ such that: (1) $\bigwedge_{j \in Q \cup \{p_i\}} C(p_j)$ is unsatisfiable, and (2) for any $R \subset Q$, we have that $\bigwedge_{j \in R \cup \{p_i\}} C(p_j)$ is satisfiable. A *complete* set of unsatisfiable path cores for path p_i is a set Ω of unsatisfiable path cores such that for every unsatisfiable path core Q of p_i, it is the case that Q contains some element of Ω. If at least one path in each member of a complete set is proved terminating, then p_i itself terminates.

Such a complete set of unsatisfiable path cores is returned by the procedure $GetCompleteUnsatCores$. This procedure works by enumerating subsets of $P \setminus \{p_i\}$ from smaller sets to bigger sets. For each subset Q, it invokes the constraint solver on the constraints corresponding to Q to check for satisfiability. In the worst case, this method takes exponential time in n, but in practice n (the number of paths through the loop) is small, so this is not a bottleneck. Lines 14-16 construct the dependency formula ψ_{deps} from Ω, the set of unsatisfiable cores. Line 20 conjoins the dependency formulas for each of the paths $p_i \in L$ to get a formula ψ_{pre}, and ψ_{pre} is used to generate formula Θ in line 22. Θ is valid if and only if the dependency relationships for termination among paths imply that each path executes a finite number of times. If Θ is valid, then the loop terminates, and an LLRF can be constructed from the dependency relationship. Given that the synthesis of LLRFs has been addressed in previous work, we omit the details for this discussion.

Now suppose that Θ is invalid. For a satisfying assignment to $\neg\Theta$, if $\{v_{f1}, v_{f2}, \ldots, v_{fk}\}$ is the set of all variables assigned to **false**, then the set of paths $\{p_{f1}, p_{f2}, \ldots, p_{fk}\}$ may execute together infinitely often. Recall that TREX requires that a counterexample to termination produced by $GetLLRF$ be in the form of a single trace. However, if the input loop had a single path through its body that acted as a counterexample, then it would have been found at line 6. Thus for every unsatisfying assignment, $GetLLRF$ constructs a loop whose set of paths are all concatenations of pairs of paths in $\{p_{f1}, p_{f2}, \ldots, p_{fk}\}$. $GetLLRF$ then recurses on the resulting set of paths. If the recursive call finds a singleton counterexample trace, then the algorithm returns this trace as a counterexample. If every recursive call determines that its loop terminates, then the original loop in fact terminates. In this case, the algorithm constructs an LLRF. Again, we omit the details of this construction.