Ramsey vs. lexicographic termination proving

Byron Cook[†], Abigail See[♯] and Florian Zuleger[♭]

[†]Microsoft Research & University College London
[♯]University of Cambridge
[♭]TU Wien

Abstract. Termination proving has traditionally been based on the search for (possibly lexicographic) ranking functions. In recent years, however, the discovery of termination proof techniques based on Ramsey's theorem have led to new automation strategies, *e.g.* size-change, or iterative reductions from termination to safety. In this paper we revisit the decision to use Ramsey-based termination arguments in the iterative approach. We describe a new iterative termination proving procedure that instead searches for lexicographic termination arguments. Using experimental evidence we show that this new method leads to dramatic speedups.

1 Introduction

The traditional method of proving program termination (*e.g.* from Turing [25]) is to find a single monolithic ranking function that demonstrates progress towards a bound during each transition of the system. Often, in this setting, we must use lexicographic arguments (*i.e.* ranking functions with range more complex than the natural numbers), as simple linear ranking functions are not powerful enough even in some trivial cases. Recent tools (*e.g.* [3], [8], [14], [15], etc) have moved away from single ranking functions and towards termination arguments based on Ramsey's theorem (*e.g.* [7], [9], [11], [22], etc). The advantage of these new approaches is that we do not need to find lexicographic termination arguments, which are perceived to be difficult to find for large programs. Instead, in these new frameworks, we typically need only to find a set of simple linear ranking functions. The important distinction here is that lexicographic ordering does not matter, thus making the *finding* of termination arguments much easier.

The difficulty with these new termination methods is establishing validity of the termination argument in hand: in the Ramsey-based setting a valid termination argument typically must hold for the *transitive closure* of the program's transitions, rather than only for individual transitions. Thus, the proof of a termination argument's validity is much harder. In size-change [15] or variance analysis [3] the result is imprecision: the tools are fast but can only prove a limited set of programs due to inaccuracies in the underlying abstractions that facilitate reasoning about the transitive closure. In iterative-based approaches (*e.g.* [8], [14]) the result is lost performance and scalability, as symbolic model

checking tools are ultimately used to reason about the program transition relation's transitive closure—something generally accepted as difficult.

In this paper we revisit the use of the Ramsey-based termination arguments used in the iterative-based approach to termination proving used in tools such as ARMC [23], TERMINATOR [8], and the termination proving module of CPROVER [6, 14]: rather than iteratively finding Ramsey-based termination arguments, we instead aim to iteratively find traditional lexicographic termination arguments. The advantage of this approach is that the validity checking step in the iterative process is much easier. The difficulty is that, outside of termination proving for rewrite systems, scalable methods for finding lexicographic ranking functions for whole programs are previously unknown.

We describe such a method. In our approach we keep information from all past failed proof attempts and use it to iteratively strengthen a lexicographic termination argument. Using experimental evidence we demonstrate dramatic performance improvements made possible by the new approach.

Related work. In this work we draw inspiration from the APROVE termination proving tool for rewrite systems [12], which proves termination of whole programs using what are effectively lexicographic arguments. The difficulty with APROVE, however, is that it has limited support for the discovery of supporting invariants. In our procedure we get the best of both worlds: lexicographic termination arguments are used, and invariants are found on demand via a reduction to tools for proving safety properties.

In our tool, during each iterative step of the proof search, we make use of constraint-based ranking function synthesis techniques from Bradley, Manna, and Sipma [4]. The difference here is that we iteratively enrich the termination argument using successful calls to a constraint-based tool on slices of the program, whereas constraint-based ranking function synthesis tools (*e.g.* [4], [5], [21], etc) were originally applied to entire programs.

Kroening *et al.* [14] optimize Ramsey-based iterative termination arguments using transitivity: attempts are made to strengthen a Ramsey-based termination argument such that it becomes a transitive relation, thus facilitating faster reasoning about the termination argument's validity. Note that in some simple cases the transitive and lexicographic arguments for a program can be similar, though lexicographic arguments are more strictly defined. The difference in our work is that we make use of all past failed termination proofs to find lexicographic termination arguments. Our choice results in increased time spent looking for termination arguments, but less time spent proving their validity.

Here we are addressing the performance of the iterative approach to termination proving, not techniques such as size-change or variance analysis. Fogarty and Vardi's experiments [10] indicate that Ramsey-based termination arguments are superior to lexicographic-based arguments in size-change.

Limitations. We are focusing primarily on arithmetic programs (*e.g.* programs that do not use the heap). In some cases we have soundly abstracted C programs with heap to arithmetic programs (*e.g.* using a technique due to Magill *et*

```
1    while x>0 and y>0 do
2         if * then
3              x := x - 1;
4         else
5              x = *;
6              y = y - 1;
7         fi
8    done
```

Fig. 1. Example terminating program. The symbol `*` is used to represent non-deterministic choice.

```
1    assume (x>0 and y>0);
3    x := x - 1;
```

(Cycle 1)

```
1    assume (x>0 and y>0);
5    x := *;
6    y := y - 1;
```

(Cycle 2)

Fig. 2. Two cycles in the program in Fig. 1

al. [16]); in other cases, as is standard in many tools (*e.g.* SLAM [2]), we essentially ignored the heap. Techniques that more accurately and efficiently reason about mixtures of heap and arithmetic are an area of open interest. Additionally, later in the paper we discuss some curious cases where linear lexicographic termination arguments alone are not powerful enough to prove termination, but linear Ramsey-based ones are. For these rare cases we describe some ad hoc strategies that facilitate the use of linear lexicographic termination arguments. In principle, however, if these approaches do not work, we would need to default to Ramsey-based arguments.

2 Example

Consider the example program in Fig. 1. When attempting to prove termination of this example the TERMINATOR tool would, during its iterative process, end up examining two cycles in the program, as seen in Fig. 2. We know that the first cycle cannot be executed forever because `x` always decreases and is bound by 0. The second cycle also cannot be executed forever, as `y` always decreases and is bound by 0.

But what of paths that consist of a mixture of Cycle 1 and Cycle 2? To prove termination of any such path, we must verify that over any finite sequence (of any length) consisting of Cycle 1 and Cycle 2, at least one of `x` or `y` decreases and is bound by 0. If `oldx` and `oldy` are the values of `x` and `y` at the some previous position of the sequence, we must verify that at the end of the sequence:

$$(\texttt{x} < \texttt{oldx} \text{ and } 0 \leq \texttt{oldx}) \text{ or } (\texttt{y} < \texttt{oldy} \text{ and } 0 \leq \texttt{oldy}).$$

```
copied := 0;
while x>0 and y>0 do
     if copied=1 then
          assert((x<oldx and 0≤oldx) or (y<oldy and 0≤oldy));
     else if * then
          copied := 1;
          oldx := x;
          oldy := y;
     fi
     if * then
         x := x - 1;
     else
         x := *;
         y := y - 1;
     fi
done
```

Fig. 3. Termination argument validity check for the program in Fig. 1, with Ramsey-based termination argument (`x` $<$ `oldx` and $0 \leq$ `oldx`) or (`y` $<$ `oldy` and $0 \leq$ `oldy`). The instrumented code used by TERMINATOR's validity check is underlined.

Following Podelski & Rybalchenko's transition invariants [22], if we can find a finite set of ranking functions such that over any sub-sequence of transitions from one reachable program state to another, (*i.e.* over any pair of states in the transitive closure of the program's transitions), at least one of the ranking functions decreases and is bound by 0, then we have proved termination. We refer to this type of termination argument as a *Ramsey-based termination argument.*

To prove the validity of the termination argument discussed above, TERMINATOR would then use a known program transformation [8] to produce the program in Fig. 3. The `assert` command in this program fails iff the Ramsey-based termination argument is not valid. Model checking techniques for safety (*e.g.* SLAM [2], BLAST [13], CPROVER [6], IMPACT [18], WHALE [1], etc) can then be used to prove/disprove the `assert`. The problem with this strategy is that the safety proof is unnecessarily tricky: we need to prove that, after the `copied := 1` statement, *each* time the `assert` statement is reached it cannot fail. The safety prover is then effectively forced to find and prove an inductive transition invariant [22] that implies that the termination argument holds for every iteration of the loop after the assignment `copied := 1`. Experimentally we find that the performance of this strategy suffers dramatically as the complexity of the loop body increases. For simple programs (*e.g.* device drivers) with few nested loops, this approach suffices, but for more complex programs, problems arise.

In this paper we instead propose to use more sophisticated constraint techniques to find the *lexicographic termination argument* that, in this case, orders the ranking function `y` before `x`. In our notation from before we might express

```
copied := 0;
while x>0 and y>0 do
    if copied=1 then
        assert((x<oldx and 0≤oldx and y≤oldy) or (y<oldy and 0≤oldy));
        exit();
    else if * then
        copied := 1;
        oldx := x;
        oldy := y;
    fi
    if * then
        x := x - 1;
    else
        x := *;
        y := y - 1;
    fi
done
```

Fig. 4. Termination argument validity check for lexicographic termination argument. The differences from Fig. 3 are underlined.

this argument as:

$$(\texttt{x} < \texttt{oldx} \text{ and } 0 \leq \texttt{oldx} \text{ and } \texttt{y} \leq \texttt{oldy}) \text{ or } (\texttt{y} < \texttt{oldy} \text{ and } 0 \leq \texttt{oldy}).$$

Here, we require that either `y` decreases towards a bound, or `x` decreases towards a bound *and* `y` *does not increase*. To prove the validity of this termination argument we need only prove that this condition holds over any one cycle, rather than over any sequence of cycles. Therefore we call on a safety prover to prove that the `assert` found in Fig. 4 cannot fail.

The advantage of the problem in Fig. 4 over that in Fig. 3 is the call to `exit`(): we need only prove that the *first* call to the `assert` cannot fail, as only one call is possible. In many cases this change results in an enormous overall performance advantage, as no inductive transition invariant is required. The difficulty here is that we must use more powerful constraint-solving techniques to find the lexicographic termination argument. Experimentally we find that the increased time spent up-front looking for a stronger termination argument pays off in the end.

3 Procedure

In this section we describe our new lexicographic-based iterative termination proving procedure.

Programs, locations, paths. As usual (*e.g.* [17]) we assume that programs are represented as graphs with locations and edges labeled with transition relations.

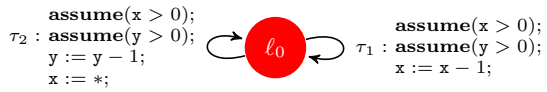

Fig. 5. Graph-based representation of the program from Fig. 1

Here we represent the transitions between edges as commands with assignment or **assume** statements (from Nelson [20]). For example, the program in Fig. 1 would be represented as the graph in Fig. 5. In formulae describing sets of states we can specify locations in the program's graph using the variable pc, which ranges over program locations ℓ_0, ℓ_1, etc. A *path* is a feasible sequence of transitions between states. A *cycle* is a path whose start and end states have the same program location ℓ, and that does not visit ℓ in between. We map from commands to relations on states in the usual way, *e.g.* $[\![\mathtt{x} := \mathtt{x} + 1]\!] = \{(s,t) \mid t(\mathtt{x}) = s(\mathtt{x}) + 1 \wedge \forall v \in \textsc{Vars} \setminus \{\mathtt{x}\}, t(v) = s(v)\}$. We map sequences of transitions to relations using relational composition, *e.g.* $[\![\langle \tau_1, \tau_2, \tau_3 \rangle]\!] = [\![\tau_1]\!]; [\![\tau_2]\!]; [\![\tau_3]\!]$.

Termination arguments. A ranking function is a map from the state-space of the program to a well-ordered set. Ranking functions are used to measure the progress of the terminating process. A *linear* ranking function is of the form $r_1 x_1 + \cdots + r_m x_m + r_{m+1}$ where $x_1, x_2, \ldots, x_m$ are the program variables. Our linear ranking functions range over the well-ordered set of the natural numbers with the relation $\leq$. Given a ranking function f, we define its *ranking relation* as

$$T_f = \{(s,t) \mid f(s) > f(t) \wedge f(s) \geq 0\}$$

i.e. all pairs of states over which f decreases and is bound by 0. Transitions in the ranking relation contribute to the progress of f. Similarly, we define a ranking function's *unaffecting relation* as

$$U_f = \{(s,t) \mid f(s) \geq f(t)\}$$

i.e. all pairs of states over which f is not increased. Transitions in the unaffecting relation do not impede the progress of f. Given a binary relation ρ over the state-space, we say that a ranking function f is *unaffected* by ρ if $\rho \subseteq U_f$.

We now consider $\Pi = \langle \rho_1, \rho_2, \ldots, \rho_n \rangle$, a finite sequence of n binary relations over the state-space, representing n cycles that are found during our iterative procedure. We define a *linear lexicographic ranking function* (LLRF) for Π as a finite sequence of n linear ranking functions $\langle f_1, f_2, \ldots, f_n \rangle$ such that $\forall i \in \{1, 2, \ldots, n\}$: **a)** $\rho_i \subseteq T_{f_i}$, and **b)** $\forall j < i, \rho_i \subseteq U_{f_j}$. That is, f_i decreases and is bound by 0 over ρ_i, and $f_1, f_2, \ldots, f_{i-1}$ are all unaffected by ρ_i. Given a lexicographic ranking function, we can define the *lexicographic ranking relation* L as all pairs of states that, for some $i \in \{1, 2, \ldots, n\}$, are contained within $U_{f_1} \cap U_{f_2} \cap \cdots \cap U_{f_{i-1}} \cap T_{f_i}$. Clearly $\bigcup \Pi \subseteq L$. Note that for any lexicographic ranking function, its lexicographic ranking relation is well-founded by construction. This

```
input: program P

T := ∅, empty termination argument
Π := ⟨⟩, empty sequence of relations
UNUSED := ⟨⟩, empty sequence of (Π, T) pairs
repeat
   if ∃ cycle π in P s.t. [[π]] ⋢ T then
      let n = length(Π) = length⟨ρ1, ρ2, ..., ρn⟩
      SUCCESSES := ∅, empty set of (Π, T) pairs
      for i = 1 to n + 1 do
         let Πi = ⟨ρ1, ρ2, ..., ρi−1, [[π]], ρi, ..., ρn⟩
         if ∃ a LLRF Fi for Πi
            let Li = the lexicographic ranking relation for Fi
            SUCCESSES := {(Πi, Li)} ∪ SUCCESSES
      if |SUCCESSES| ≥ 1
         randomly choose one (Πi, Li) ∈ SUCCESSES and remove it
         Π := Πi
         T := Li ⊇ ⋃ Π
         UNUSED := (sequence of SUCCESSES) ⊕ UNUSED
      else
         if |UNUSED| ≥ 1 then
            (Π, T) := head(UNUSED)
            UNUSED := UNUSED\{(Π, T)}
         else
            report "Unknown"
   else
      report "Success"
end.
```

Fig. 6. Lexicographic-based iterative termination procedure. $\oplus$ denotes concatenation of finite sequences.

is the reason why we need only verify that each individual transition obeys the lexicographic termination argument, rather than the transitive closure. In this paper termination arguments will take the form of lexicographic ranking relations.

Termination procedure. Our iterative lexicographic-based termination proving procedure is found in Fig. 6. We begin with an empty termination argument, T. We search for a witness (a cycle π) to the failure of the validity of this argument. Our implementation of the search for a witness is an adaptation on the reduction to safety proving from Cook, Podelski, and Rybalchenko [8].

Our procedure then goes on to keep and use all of the witnesses (Π) to the failure of T. If there are none, we have proved termination. Otherwise if we find a witness, we add it to Π in the form of a relation. Each time a relation is added, a new LLRF is synthesized for Π. Each new termination argument T contains

```
1    while x>0 and y>0 and d>0 do
2      if * then
3        x := x - 1;
4        d := *;
5      else
6        x = *;
7        y = y - 1;
8        d = d - 1;
9      fi
10   done
```

(a)

```
1    while x>0 and y>0 and z>0 do
2      if * then
3        x := x-1;
4      else if * then
5        y := y-1;
6        z := *;
7      else
8        z := z-1;
9        x := *;
10     fi
11   done
```

(b)

Fig. 7. Example terminating programs.

$\bigcup \Pi$, so we continue to add to Π until (hopefully) T is a valid termination argument for the program P. It is therefore useful to think of Π rather than T as representing the progress of the algorithm.

Once we have a sequence of relations $\Pi = \langle \rho_1, \rho_2, \ldots, \rho_n \rangle$, the LLRF for Π is synthesized by finding a linear ranking function f_i for each relation ρ_i in Π. We additionally attempt to satisfy the Unaffected constraints: That is, $\forall i \in \{1, 2, \ldots, n\}$, we require that ρ_i does not increase any of $f_1, f_2, \ldots, f_{i-1}$. We have then constructed a linear lexicographic ranking function $\langle f_1, f_2, \ldots, f_n \rangle$ for Π. Previously known constraint-based techniques using Farkas' Lemma (*e.g.* [4],[5],[21]) are used to find the sequence of functions satisfying the above.

Note that for each new Π, we synthesize the LLRF anew, which allows each individual ranking function f for a particular relation ρ to change from one iteration to the next. This is necessary, as permanently designating a ranking function to each relation can lead to a failure to find a solution that does in fact exist. As an example, consider the loop in Fig. 7(a), which is the same as Fig. 1 except it features a decoy variable `d`. The lexicographic termination argument $\langle y, x \rangle$ we found earlier for Fig. 1 is clearly valid for this loop too. We examine two cycles: Lines 1,3,4, which induces $\rho_1 = [\![x > 0 \wedge y > 0 \wedge d > 0 \wedge x' = x - 1 \wedge y' = y]\!]$; and Lines 1,6,7,8, which induces $\rho_2 = [\![x > 0 \wedge y > 0 \wedge d > 0 \wedge y' = y - 1 \wedge d' = d - 1]\!]$.

Suppose we find ρ_2 first, and choose $f_2 = d$ as its ranking function. Suppose we then find ρ_1. We need a LLRF for either $\langle \rho_1, \rho_2 \rangle$ or for $\langle \rho_2, \rho_1 \rangle$. If we require that $f_2 = d$ from the previous iteration, then this means we must find f_1 a linear ranking function for ρ_1 such that one of the two following options holds:

a) $\langle f_1, d \rangle$ is a LLRF for $\langle \rho_1, \rho_2 \rangle$. So we need f_1 to be unaffected by ρ_2.
b) $\langle d, f_1 \rangle$ is a LLRF for $\langle \rho_2, \rho_1 \rangle$. So we need $f_2 = d$ to be unaffected by ρ_1.

Clearly **b)** is unsatisfiable because d isn't unaffected by ρ_1. **a)** is also unsatisfiable because to be a linear ranking function for ρ_1, f_1 must be of the form

$r_x x + r_y y + c$ with $r_x > 0$, and therefore f_1 isn't unaffected by ρ_2. Therefore if we require the ranking function for ρ_2 to stay the same throughout the execution of our procedure, we may find no solutions, due to an earlier unlucky choice of ranking function. However, if we allow f_2 to be changed from d, we will be able to find our solution $\langle f_2, f_1 \rangle = \langle y, x \rangle$, which is a valid lexicographic ranking function for $\langle \rho_2, \rho_1 \rangle$, and for the whole loop.

Fortunately, synthesizing LLRFs for a small (and fixed order) Π is cheap, so the re-synthesis of the LLRFs has little effect on performance. This statement is not without a caveat: incremental approaches to safety proving in practice allow us to resume the validity checking from where we left off in the previous iteration, thus major changes to the ranking function can make for additional work in the safety prover. As a further optimization we could imagine using the interpolants found in the safety prover to help guide the search for even better termination arguments.

Choosing the lexicographic ordering. As mentioned previously, the relations in Π must be put in some lexicographic order $\langle \rho_1, \rho_2 \dots \rho_n \rangle$ for a lexicographic ranking function to be found. As shown in Fig. 6, this is done by insertion — the relation that has just been found is inserted into the previous lexicographic ordering. This means that after the n^{th} relation is found, there are n places it can be inserted, *i.e.* n choices of ordering to consider. For each of the n orderings, we attempt to find a LLRF. If there are one or more orderings that yield solutions, we choose at random an ordering and its corresponding lexicographic ranking relation to form our new Π and T respectively.

The advantage of this method is that should we find that a certain ordering yields no solutions, we do not investigate it further. That is, if there does not exist a LLRF for some ordering Π, then there does not exist a LLRF for any ordering obtained by inserting relations into Π, and we do not investigate any such orderings. The disadvantage of this method is that it can be *too* selective, leading us to a dead end. We demonstrate this possibility in Fig. 7(b), then present our solution. We investigate three cycles: Lines 1,3, which induces $\rho_1 = [\![x > 0 \wedge y > 0 \wedge z > 0 \wedge x' = x - 1 \wedge y' = y \wedge z' = z]\!]$; Lines 1,5,6, which induces $\rho_2 = [\![x > 0 \wedge y > 0 \wedge z > 0 \wedge y' = y - 1 \wedge x' = x]\!]$; and Lines 1,8,9, which induces $\rho_3 = [\![x > 0 \wedge y > 0 \wedge z > 0 \wedge z' = z - 1 \wedge y' = y]\!]$. Suppose that during our procedure, the first two relations we find are ρ_1 and ρ_2. They have ranking functions $f_1 = x$ and $f_2 = y$ respectively. Note that ρ_1 does not increase y and ρ_2 does not increase x, so we may choose either $\langle \rho_1, \rho_2 \rangle$ or $\langle \rho_2, \rho_1 \rangle$ with LLRF $\langle x, y \rangle$ or $\langle y, x \rangle$ respectively.

- Suppose we choose $\langle \rho_2, \rho_1 \rangle$ with LLRF $\langle y, x \rangle$. Next we find ρ_3, and see that inserting it to form the new ordering $\langle \rho_2, \rho_3, \rho_1 \rangle$ yields a LLRF $\langle f_2, f_3, f_1 \rangle = \langle y, z, x \rangle$. This is a valid lexicographic ranking function for the whole loop, and so we have proved termination.
- Suppose we choose $\langle \rho_1, \rho_2 \rangle$ with LLRF $\langle x, y \rangle$. Next we find ρ_3, but there does not exist a LLRF for any one of $\langle \rho_3, \rho_1, \rho_2 \rangle$ or $\langle \rho_1, \rho_3, \rho_2 \rangle$ or $\langle \rho_1, \rho_2, \rho_3 \rangle$, so we have reached a dead end.

This example demonstrates that by investigating *only* the orderings obtained by inserting the new relation into the previous ordering, we may be unable to find an existing solution due to an earlier choice of ordering. Of course, we could investigate *all* possible permutations of the n relations to avoid this problem, but that strategy becomes infeasible once n becomes large [5], as on the n^{th} iteration we would need to investigate $n!$ cases rather than n.

In our solution (*i.e.* Fig. 6), in the event of more than one feasible ordering, we choose one randomly and keep the others in UNUSED, so that if a dead end is later reached, we may backtrack to the last random choice made, and investigate an alternative ordering. Cases such as the above that require the backtracking failsafe are uncommon. The insertion strategy with backtracking is fast because we only attempt to find n lexicographic ranking functions on the n^{th} iteration. The approach is robust because we will eventually investigate all lexicographic ranking functions we found, if necessary.

4 Towards finding the *right* ranking function

In many cases, there is more than one choice of Π that admits a LLRF, and for each Π, there may be more than one possible LLRF. Such cases give us the opportunity to consider which choices might be better than others, *i.e.* which termination argument is likely to be faster to validate using existing safety proving techniques. Note that in our setting the sequence Π affords us a great deal of information when trying to determine which argument to choose. In this section we describe several heuristics that we have found useful. We close this section with a discussion of some cases where no (linear) lexicographic termination argument exists, but linear Ramsey-based arguments can be found.

Shorter lexicographic ranking functions. Checking the validity of a lexicographic ranking function (as demonstrated in Fig. 4) becomes more difficult as the lexicographic ranking function becomes longer. This is because for a lexicographic ranking function of length n, we are checking, for each transition, whether any one of n conjunctive formulae hold.

We implemented an optimization that chooses a LLRF that uses the fewest unique ranking functions as possible. Then, if we have some of the f_i equal, we may eliminate the repeated ranking functions by keeping just the *first* occurrence of each unique ranking function. The resulting LLRF is shorter, and its lexicographic ranking relation contains $\bigcup \Pi$, so it forms our new termination argument. In one example from our experimental evaluation we found that proving termination was possible in 27s with this optimization turned on, and 157s without.

Unaffecting lexicographic ranking functions. Recall that a lexicographic ranking function $\langle f_1, f_2, \ldots, f_n \rangle$ for $\Pi = \langle \rho_1, \rho_2, \ldots, \rho_n \rangle$ must satisfy the Unaffecting constraints: every ρ_i must satisfy $\rho_i \subseteq U_{f_j} \forall j < i$. However we do not require $\rho_i \subseteq U_{f_j}$ for any $j > i$.

Intuitively, when attempting to prove the validity of a termination argument (which, ultimately, happens via the search for an inductive argument in the safety prover), it seems that checking the validity of a lexicographic ranking function is easier when the relations interfere minimally with the other relations' ranking functions, *i.e.* increase them as little as possible. That is, we wish to satisfy as many of the extra Unaffecting constraints $\{\rho_i \subseteq U_{f_j} \mid j > i\}$ as possible. This motivates the following definition.

Given a lexicographic ranking function $\langle f_1, f_2, \ldots, f_n \rangle$ for $\Pi = \langle \rho_1, \rho_2, \ldots, \rho_n \rangle$, its *Unaffecting Score* U is

$$U = \sum_{1 \leq i < j \leq n} 1_{U_{f_j}}(\rho_i)$$

where the indicator function $1_{U_f}(\rho)$ equals 1 if $\rho \subseteq U_f$ and 0 otherwise. In other words, U is the number of extra unaffecting constraints satisfied. Note that we always have $0 \leq U \leq \frac{n(n-1)}{2}$, and requiring $U = 0$ is equivalent to the usual lexicographic ranking function constraints.

We implemented a constraint-based optimization that chooses a LLRF with highest possible Unaffecting Score. In our experiments the example mentioned above (that required 157s without optimizations) was proved terminating in 82s with this optimization turned on.

When linear lexicographic ranking relations are not enough. Existence of a linear Ramsey-based termination argument for a loop does not imply existence of a linear lexicographic termination argument for the same loop. We illustrate two simple but typical examples. See Fig. 8. For both examples we present a simple solution that alters the problem slightly, allowing us to continue to use lexicographic techniques to prove termination. Note that both of these simple workarounds are not new—variations upon these themes have been used in previous tools (*e.g.* AProVE [12]). Our intention here is to illustrate the type of problems that arise when moving from Ramsey-based to lexicographic termination arguments.

In Fig. 8(a), the variable `x` starts as any integer, then increases or decreases (as appropriate) until it equals 0, upon which the loop terminates. A valid Ramsey-based termination argument for the loop is:

$$(\texttt{x} < \texttt{oldx} \text{ and } 0 \leq \texttt{oldx}) \text{ or } (\texttt{-x} < \texttt{-oldx} \text{ and } 0 \leq \texttt{-oldx}).$$

However there does not exist a LLRF for the loop. Neither $\langle x, -x \rangle$ nor $\langle -x, x \rangle$ is valid, as every transition decreases one of the functions and increases the other. A solution to this problem is shown in Fig. 9(a). The variable `c` is introduced to record which of the two options was taken upon entry to the loop the first time through. In our procedure we instrument such variables into the representation of the program. Then, in the case where we cannot find a LLRF — and before resorting to a Ramsey-based termination argument — we would attempt to build the following lexicographic termination argument that case splits on `c`:

```
while x<>0 do
    if x>0 then
        x := x-1;
    else
        x := x+1;
    fi
done
```

(a)

```
assume(m>0);
while x<>m do
    if x>m then
        x := 0;
    else
        x := x+1;
    fi
done
```

(b)

Fig. 8. Example programs where Ramsey-based linear termination arguments exist, but linear lexicographic termination arguments do not.

$\langle f_1 \rangle = \langle x \rangle$ for `c=1` and $\langle f_2 \rangle = \langle -x \rangle$ for `c=2`. The relation would be encoded as

(`c` $= 1$ and `x` $<$ `oldx` and $0 \leq$ `oldx`) or (`c` $= 2$ and $-$`x` $<$ $-$`oldx` and $0 \leq -$`oldx`).

This extension aims to deal with cases where there is a split-case at the beginning of the loop, necessitating seemingly conflicting ranking functions that prohibit construction of a lexicographic ranking function, but the loop is nonetheless terminating because the two cases are largely separate.

In Fig. 8(b), `m` and `x` start as any integers with `m` positive. If `x` is greater than `m`, `x` is set to zero. `x` is now less than `m`, so `x` increases until it equals `m`, upon which the loop terminates. A valid Ramsey-based termination argument for the loop is:

(`x` $<$ `oldx` and $0 \leq$ `oldx`) or (`m-x` $<$ `oldm-oldx` and $0 \leq$ `oldm-oldx`).

However there does not exist a LLRF for the loop. Neither $\langle x, m - x \rangle$ nor $\langle m - x, x \rangle$ is valid, as every transition decreases one of the functions and increases the other. A simple solution to this problem is shown in Fig. 9(b). The variable `iters` records how many iterations of the loop have occurred. We then attempt to prove termination lexicographically by only checking transitions for which `iters` ≥ 1, then `iters` ≥ 2, `iters` ≥ 3, etc. up to some finite limit at which point we give up. (Our failure to find a LLRF by the usual procedure means that we have already failed to prove termination for `iters` ≥ 0). When examining the path found we can easily discover if the prefix of the cycle contributes to well-foundedness using an extra constraint check. In our example, we need only attempt to prove for `iters` ≥ 1 (shown in Fig. 8(b)) to find that the lexicographic ranking function $\langle f_2 \rangle = \langle m - x \rangle$ is valid. This extension aims to deal with loops which include an initialization procedure that occurs over the first few iterations (if at all), necessitating ranking functions that conflict with those needed for the main termination argument. It allows us to construct lexicographic termination arguments that do not need to take into account the first few iterations of the loop.

```
copied := 0;
c := 0;
while x<>0 do
   if copied=1 then
      assert(  (c=1 and x<oldx and 0≤oldx)
                              or
               (c=2 and -x<-oldx and 0≤-oldx)
            );
      exit();
   else if * then
      copied := 1;
      oldx := x;
   fi
   if x>0 then
      if c=0 then c:=1;
      x := x-1;
   else
      if c=0 then c:=2;
      x := x+1;
   fi
done
```

(a)

```
copied := 0;
iters := 0;
assume(m>0);
while x<>m do
   if iters > 1 then
      if copied=1 then
         assert(m-x<oldm-oldx and 0≤oldm-oldx);
         exit();
      else if * then
         copied := 1;
         oldx := x;
         oldm := m;
      fi
   fi
   if x>m then
      x := 0;
   else
      x := x+1;
   fi
   iters := iters+1;
done
```

(b)

Fig. 9. Modified validity check transformations for programs in Fig. 8. The modifications to the standard validity check are underlined.

5 Experimental evaluation

To evaluate our approach we have implemented the algorithm from Fig. 6 as an option in the T2 termination proving tool[1]. The underlying safety prover used to check termination argument validity in T2 is a re-implementation of Impact [18]. We then applied the tool to a set of 390 termination proving benchmarks, drawn from a variety of applications (*e.g.* device drivers, the Apache webserver, Postgres SQL server, integer approximations of numerical programs from a book on numerical recipes [24], integer approximations of benchmarks from LLBMC [19], etc). Note that, as we mentioned earlier, in some cases we have soundly abstracted C programs with data-structures to pure arithmetic programs using a technique due to Magill *et al.* [16]. In other cases we have ignored the heap. We have used the same input files for all experiments and configurations, thus the treatment of heap is orthogonal to the investigation here.

To see the difference between Ramsey-based and lexicographic-based iterative termination proving, we compared our new procedure to T2's re-implementation of the original Terminator procedure (which includes an integration of the optimization from Kroening *et. al* [14]). We ran the two variants of T2 on the 390 termination benchmarks, with a timeout of 300s. See Fig. 10 for the results (in logarithmic scale). Here we have excluded 266 cases where both tools were able to prove/disprove termination in under 3 seconds, as well as 42 cases where

[1] A source-code based release of this tool together with benchmarks is scheduled for release in 2013.

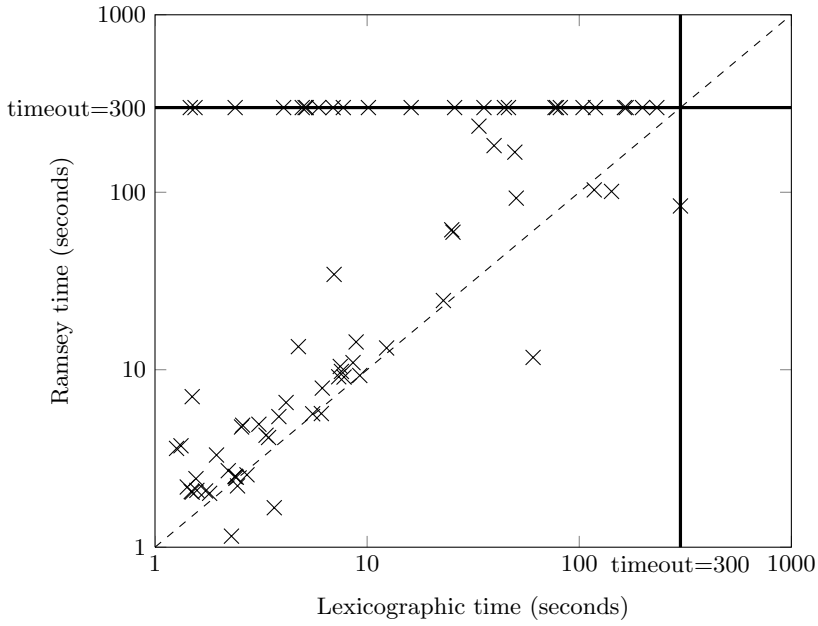

Fig. 10. Results of experimental evaluation comparing the lexicographic-based iterative termination prover from Fig. 6 to a re-implementation of TERMINATOR [8]. In total 390 termination benchmarks were used, with a timeout of 300s. Depicted here are the 82 cases in which radical differences in performance are seen (there are 42 cases where both tools timeout, and 266 easily solved cases) The lexicographic approach resulted in 26 fewer timeouts (*i.e.* the Ramsey-based termination procedure timed out on 68 benchmarks). The dotted line indicates equal performance of both methods. Note that on a log-log plot, results lying on a line parallel to the dotted line represent one method performing at a rate proportional to the other. Results were computed using an Intel 2.80Ghz processor running Windows 7. A source-code release of the tool and benchmarks is scheduled for 2013.

both tools timed out. The remaining 82 cases are shown in the figure. The most dramatic aspect of the results is the decrease in timeouts: 26.

6 Conclusion

In this paper we have reconsidered the form of termination argument used in iterative-based termination proving [8]: rather than iteratively finding Ramsey-based termination arguments, we have instead developed a method that iteratively finds traditional lexicographic termination arguments. This approach has some disadvantages (*i.e.* more complex ranking function synthesis) and advantages (*i.e.* easier termination argument validity checking). Overall the experimental evidence indicates that the advantages outweigh the disadvantages.

References

1. Albarghouthi, A., Gurfinkel, A., and Chechik, M. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI* (2012).
2. Ball, T., and Rajamani, S. K. Automatically validating temporal safety properties of interfaces. In *SPIN* (2001).
3. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., and O'Hearn, P. W. Variance analyses from invariance analyses. In *POPL* (2007).
4. Bradley, A., Manna, Z., and Sipma, H. The polyranking principle. In *ICALP* (2005).
5. Bradley, A. R., Manna, Z., and Sipma, H. B. Linear ranking with reachability. In *CAV* (2005).
6. Clarke, E. M., Kroening, D., Sharygina, N., and Yorav, K. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS* (2005).
7. Codish, M., Genaim, S., Bruynooghe, M., Gallagher, J., and Vanhoof, W. One loop at a time. In *WST* (2003).
8. Cook, B., Podelski, A., and Rybalchenko, A. Termination proofs for systems code. In *PLDI* (2006).
9. Dershowitz, N., Lindenstrauss, N., Sagiv, Y., and Serebrenik, A. A general framework for automatic termination analysis of logic programs. *Communication and Computing 12*, 1/2 (2001).
10. Fogarty, S., and Vardi, M. Büchi complementation and size-change termination. In *TACAS* (2009).
11. Geser, A. Relative termination. Doctoral dissertation, University of Passau, 1999.
12. Giesl, J., Thiemann, R., and Schneider-Kamp, P. The dependency pair framework: Combining techniques for automated termination proofs. In *LPAR* (2004).
13. Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W. Temporal-safety proofs for systems code. In *CAV* (2002).
14. Kroening, D., Sharygina, N., Tsitovich, A., and Wintersteiger, C. M. Termination analysis with compositional transition invariants. In *CAV* (July 2010).
15. Lee, C. S., Jones, N. D., and Ben-Amram, A. M. The size-change principle for program termination. In *POPL* (2001).
16. Magill, S., Berdine, J., Clarke, E., and Cook, B. Arithmetic strengthening for shape analysis. In *SAS* (2007).
17. Manna, Z., and Pnueli, A. *Temporal verification of reactive systems: Safety.* 1995.
18. McMillan, K. L. Lazy abstraction with interpolants. In *CAV* (2006).
19. Merz, F., Falke, S., and Sinz, C. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *VSTTE* (2012).
20. Nelson, G. A generalization of Dijkstra's calculus. *TOPLAS 11*, 4 (1989).
21. Podelski, A., and Rybalchenko, A. A complete method for the synthesis of linear ranking functions. In *VMCAI* (2004).
22. Podelski, A., and Rybalchenko, A. Transition invariants. In *LICS* (2004).
23. Podelski, A., and Rybalchenko, A. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL* (2007).
24. Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes: The Art of Scientific Computing.* 1989.
25. Turing, A. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines* (1949).