Automatic Termination Proofs for Programs with Shape-Shifting Heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

[1] Microsoft Research
[2] Queen Mary, University of London

Abstract. We describe a new program termination analysis designed to handle imperative programs whose termination depends on the mutation of the program's heap. We first describe how an abstract interpretation can be used to construct a finite number of relations which, if each is well-founded, implies termination. We then give an abstract interpretation based on separation logic formulæ which tracks the depths of pieces of heaps. Finally, we combine these two techniques to produce an automatic termination prover. We show that the analysis is able to prove the termination of loops extracted from Windows device drivers that could not be proved terminating before by other means; we also discuss a previously unknown bug found with the analysis.

1 Introduction

Consider the code fragment in Fig. 1, which comes from the source code of a Windows device driver. Does this loop guarantee termination? It's *supposed to*: failure of this loop to terminate would have catastrophic effects on the stability and responsiveness of the computer. Why would it be a problem if this loop didn't terminate? First of all, the device that this code is managing would cease to function. Secondly, due to the fact that this code executes at kernel-level priority, non-termination would cause it to starve other threads running on the system. Note that we cannot simply kill the thread, as it can be holding kernel locks and modifying kernel-level data-structures—forcibly killing the thread would leave the operating system in an inconsistent state. Furthermore, if the loop hangs, the machine might not actually crash.[1] Instead, the thread will likely just hang until the user resets the machine. This means that the bug cannot be diagnosed using post-crash analysis tools.

This example highlights the importance of termination in systems level code: in order to improve the responsiveness and stability of the operating system it is vital that we can automatically check the termination of loops like this one. In this case, in order to prove the termination of the loop, we need to show the following conditions:

1. `DeviceExtension->ReadQueue.Flink` is a pointer to a circular list of elements (via the `Flink` field).

[1] Although hanging kernel-threads can trigger other bugs within the operating system.

T. Ball and R.B. Jones (Eds.): CAV 2006, LNCS 4144, pp. 386–400, 2006.

```
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {
    irp = (IRP *)((CHAR *)(entry)-(ULONG *)(&((IRP *)0)->Tail.Overlay.ListEntry));
    stack = IoGetCurrentIrpStackLocation (irp);
    if (stack->FileObject == FileObject) {
        RemoveEntryList(entry);
        if (IoSetCancelRoutine (irp, NULL)) {
            return irp;
        } else {
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
```

Fig. 1. Code from a Windows device driver which contains a termination bug found by the analysis described in this paper. The bug has catastrophic effects on the responsiveness of the computer when it occurs.

2. During the execution of this loop, `entry` is always getting closer to taking the value of `&DeviceExtension->ReadQueue`.
3. The loop will terminate when `entry` is finally assigned the value equaling `&DeviceExtension->ReadQueue`.
4. The assignments to other parts of the heap occurring during the loop's execution (*e.g.,* the side-effects from executing `InitializeListHead`) do not affect conditions 1, 2, and 3.

Unfortunately, there *is a termination bug* in Fig. 1: in some cases this loop may violate condition 4.

To date, automatically checking the termination of loops like this one has been beyond any known tool. This is because the termination argument is based on the semantics of imperative heap-mutation during the loop's execution. Today's termination analysis tools (that is: tools that both find *and* check termination arguments automatically) simply do not support an analysis at this level of depth; instead they only support arguments involving the values of arithmetic variables. Examples of such tools include Terminator [5] and PolyRank [2].

In this paper, we present a new termination prover which supports loops of this sort. In cases where loops have termination bugs the prover is able to provide information which can be used to automatically find a counterexample. The prover implements an abstract analysis, based on formulæ expressed in separation logic, which keeps track of the relative differences between heap objects while abstracting away the exact details. This analysis produces a finite collection of depth relations such that the program is well-founded if each individual relation is. The correctness of the termination argument relies on a result of Podelski & Rybalchenko [12]. The candidate depth relations constructed are checked for well-foundedness with the use of projection and the RankFinder tool [11].

Separation logic [14] is used as the basis of our analysis because it lets us symbolically carry around just enough information to prove that loops are *making*

```
MUTANT (P, I, ℓ) {
  Y := SONAR*_P[{I}]
  if ⊤ ∈ Y return "Loop may crash", with ⊤
  foreach y ∈ Y such that pc(y) = ℓ {
    s := SEED(y)
    Z := SONAR+_P[{s}]
    foreach z ∈ Z such that pc(z) = ℓ {
      if ¬WF(z) return "Loop may diverge", with (y, z)
    }
  }
  return "Loop ℓ-terminates"
}
```

Fig. 2. MUTANT algorithm

progress while abstracting enough information such that the tool produces a compact over-approximation of the reachable states. Furthermore, separation logic mitigates the need for a global alias check when size information is changed: the alteration of the size of one piece of the heap does not affect any others that are held in different components of a separating conjunction. These characteristics are what make the new termination proof method powerful, yet still tractable.

This paper begins with a description of the algorithm, followed by the details of the separation logic analysis, and then experimental results. Our experiments include loops extracted from Windows device drivers that could not be handled using TERMINATOR [5] due to its overly-coarse model of heaps [13].

2 Termination Via Separation Analysis and Rank Synthesis

Our termination checking algorithm, MUTANT, is displayed in Fig. 2. The input is a program P, an abstract initial state expressed as a separation logic formula I, and a program location ℓ. The algorithm is designed to prove that the program P cannot visit location ℓ infinitely-often during its execution when started in states satisfying I. We call this condition ℓ-termination. If we wish to prove that P terminates, we can prove ℓ-termination for each program location. We can also optimize by focusing only on a subset of the locations (*i.e.,* a set of *cutpoints* [9]).

MUTANT first calls an analysis engine, SONAR (defined in Section 3), to calculate the finite set of reachable abstract states Y. For a program P, SONAR_P is the binary transition relation on abstract states, and SONAR^*_P denotes its reflexive transitive closure. $\text{SONAR}^*_P[\{I\}]$ denotes the post-image under the pre-state I. During this analysis SONAR also proves that P cannot commit any of a basic set of safety violations, such as an access to a deallocated heap object. If SONAR returns $\top$, then it cannot guarantee that P is safe from this class of errors—and our method cannot prove termination.

Next, for each reachable symbolic state y at program location ℓ, the algorithm constructs a new state with additional history variables that symbolically record a snapshot of the depths of pieces of y's heap. We call this step *seeding*, and use

the notation SEED(y) to represent the output of this operation. If $s = \text{SEED}(y)$ then, when symbolically executing instructions starting from s, we can see how the effects of these operations relate to the original values from y.

MUTANT then calls SONAR again to compute the states reachable from SEED(y) in *at least one step* and which are at the same program location (*i.e.,* $\mathsf{pc}(y) = \mathsf{pc}(z) = \ell$). Each of the pairs (y, z) in the abstract semantics determines an over-approximation of transitions in the concrete semantics of the program, and together they over-approximate all transitions in the concrete semantics.

The SONAR analysis uses heap predicates together with certain auxiliary variables that describe heap depths. For example $\mathsf{ls}^k(a, b)$ describes a linked list of length k running from a to b. Seeding maps this formula to $k_s{=}k \wedge \mathsf{ls}^k(a, b)$, where k_s is a symbolic constant used to record the initial value of k. Running SONAR starting from this state can change k but not k_s. So, if the final state is $k_s{>}k \wedge \mathsf{ls}^k(a, b)$ then this indicates that the linked list has decreased in length (as can happen, *e.g.*, by removing an element from the list).

Because we seed y before running SONAR again to obtain z, the single abstract state z will actually contain information, in the form of an assertion, which relates initial (seeded) values of heap-depth variables to their final values (for this run of SONAR). In the above example it is just $k_s{>}k$. The WF(z) procedure extracts this information from z and treats it as a binary relation T_i, which relates the relative differences between the depths of pieces of heaps referenced by z. WF(z) then calls the RANKFINDER tool [11] to determine if this relation is well-founded. Note that the well-foundedness of each z is checked independently of the others.

In essence, MUTANT constructs a finite set $T_1, \ldots, T_n$ of binary relations, whose union over-approximates changes to the auxiliary variables that track heap depths. If one of the determined relations T_i is not well-founded, then MUTANT's attempted proof of ℓ-termination fails. However, if all of the found pairs denote well-founded relations, then ℓ-termination has been proved. The correctness of this assertion comes from [12], which shows that: to establish that ℓ is not visited infinitely often, it is sufficient to find a finite union of well-founded relations that over-approximates the transitions through location ℓ.

The algorithm is different from the one in the TERMINATOR tool [5]. As described in [3], TERMINATOR uses counterexample-guided abstraction refinement to add disjuncts to a collection $T_1, \ldots, T_n$ of well-founded relations, and then uses a *binary reachability analysis* [4] to check the subset inclusion. Checking the inclusion is the expensive part of TERMINATOR. In contrast, here we never do the inclusion check. Rather, SONAR produces a finite set of T_i's (determined by the (y, z) pairs), which together satisfy the inclusion by construction. As with TERMINATOR, we still have to check for well-foundedness of each T_i.

3 Tracking Depths of Abstracted Heaps

SONAR implements an analysis that is sound for safety properties of programs. It uses an abstract domain based on separation logic forumlæ, and as a result, is set up to express deep properties (meaning properties that depend on areas of

the heap not immediately referenced by program variables) of mutating heaps. Reachability between program states is computed using a fixed-point algorithm built from single-step symbolic execution (notationally: $\rightsquigarrow$) together with a case analysis or concretion step ($\rightarrow_E$) which incrementally reveals the pointer structure of abstracted or summarized heap objects, and an abstraction step ($\rightarrow$) which enables convergence to fixed-points.

SONAR is based on SPACEINVADER [7]. The difference between the two analysis engines is in SONAR's tracking of depths of inductive heap predicates. Depth does not necessarily refer to lengths of pointer chains in the heap, but instead refers to the number of inductive unfoldings a formula represents. For lists, this corresponds to length. In this section we describe the underlying fundamentals behind SONAR. We focus on linked lists in the exposition, but the method of proving ℓ-termination generalizes to data structures expressed using other inductive predicates in separation logic, such as trees, doubly-linked lists, etc.

Programs. SONAR supports a simple language of goto programs extended with the usual four heap operations: allocate, deallocate, load, and store. A program P is a function mapping a fixed finite subset of naturals $\{0, \ldots, end - 1\}$ to commands C, given by:

$$
\begin{array}{rll}
E & ::= \mathsf{nil} \mid x \mid x' & \text{expressions} \\
S & ::= \mathtt{skip} \mid x{:=}\, E \mid x{:=}\, \mathtt{new}() & \text{safe commands} \\
A(E) & ::= \mathtt{dispose}(E) \mid x{:=}\, [E] \mid [E]{:=}\, E & \text{heap accessing commands} \\
B & ::= E{=}E & \text{simple Boolean formulæ} \\
G & ::= B \mid \neg B & \text{branch tests} \\
C & ::= S \mid A(E) \mid \mathtt{goto}\, n \mid \mathtt{if}(G)\ \{\mathtt{goto}\, n\}\ \mathtt{else}\ \{\mathtt{goto}\, n\} &
\end{array}
$$

where $n \in \{0, \ldots, end\}$; and variables $x, y, \ldots$ range over some infinite set Var; and primed variables $x', y', \ldots$ range over some disjoint infinite set Var$'$. Primed variables cannot appear in programs. They are included in expressions since these also appear in formulæ (below). For convenience of later definitions, commands S are syntactically distinguished from commands $A(E)$. The difference between the two is that for a command S, execution is always safe, while execution of a command $A(E)$ may be unsafe, due to access of heap location E.

Symbolic Heaps and Depths. SONAR operates over an abstract domain that represents sets of concrete program states as sets of separation logic formulæ called symbolic heaps. Note the inclusion of depth formulæ K, and depth variable annotations N on list segment predicates $\mathsf{ls}^N(E, E)$:

$$
\begin{array}{ll}
N ::= 1 \mid k \mid k' & \Pi ::= \mathsf{true} \mid B \mid K \mid \Pi \wedge \Pi \\
K ::= N{=}N \mid N{>}N & \Sigma ::= \mathsf{emp} \mid H \mid \Sigma * \Sigma \mid \mathsf{junk} \\
H ::= E{\mapsto}E \mid \mathsf{ls}^N(E, E) & Q ::= \Pi \wedge \Sigma
\end{array}
$$

Depth variables $k, l, \ldots, k', l', \ldots$ and primed depth variables $k', l', \ldots$ range over DVar and DVar$'$, respectively, and denote natural numbers. DVar is infinite and

disjoint from Var and Var′, while DVar′ is an infinite subset of DVar. Note that formulæ are considered up to symmetry of $=$, permutations across $\wedge$ and $*$ (*e.g.*, $\Pi \wedge B_0 \wedge B_1$ and $\Pi \wedge B_1 \wedge B_0$ are equated), unit laws for true and emp, and idempotency of $- * \mathsf{junk}$ (*e.g.*, $\mathsf{junk} * \mathsf{junk}$ and junk are equated).

Symbolic heap formulæ consist of two parts: a Boolean formula Π built from $=$, $>$, and $\wedge$ which is independent of the heap and has the usual classical arithmetic meaning; and a heap formula Σ which expresses heap shape. The meaning of a symbolic heap Q is the same as $\exists \vec{x'}, \vec{k'}. Q$ in the usual semantics of separation logic [14], where we existentially quantify all the primed variables. The empty heap, which contains no allocated cells at all, is described by emp. A heap consisting of a single cell at location E with contents F is described by $E \mapsto F$. The separating conjunction $*$ describes composition of disjoint heaps: heaps with shape $\Sigma_0 * \Sigma_1$ consist of two subheaps with no allocated locations in common, one with shape Σ_0 and the other with shape Σ_1. Non-empty heaps, usually consisting of unreachable cells, are described by junk. Finally, $\mathsf{ls}^N(E, F)$ describes acyclic singly-linked lists of length $N \geq 1$. Cyclic lists, such as that in the introductory example, can be expressed using multiple predicates: *e.g.*, $\mathsf{ls}^k(x, y') * \mathsf{ls}^j(y', x)$. Note that the ls^1 and $\mapsto$ predicates are not equivalent, since $x \mapsto x$ admits cycles (of length one), while $\mathsf{ls}^1(x, x)$ is inconsistent.

The definition of the abstract transition relation asks several types of questions about symbolic heaps: entailment of an equality ($Q \vdash E{=}F$), entailment of a disequality ($Q \vdash E{\neq}F$), or inconsistency ($Q \vdash \mathsf{false}$). We also sometimes ask the negations of these questions. Sound implementations of these queries can be obtained from those defined in [7].

Symbolic Execution ($\rightsquigarrow$). The symbolic execution relation captures the effect of executing a straight-line command from a symbolic heap. That is, $Q_0 \overset{C}{\rightsquigarrow} Q_1$ means that Q_1 over-approximates the concrete states which can result from executing C from states satisfying Q_0. We do not show the axioms which define symbolic execution of basic commands S and $A(E)$ as they are reported in [1, 7], but for illustration we show the axiom for loading the contents of a memory address E into x:

$$Q * E \mapsto F \overset{x:=[E]}{\rightsquigarrow} x{=}F[x'/x] \wedge (Q * E \mapsto F)[x'/x] \tag{1}$$

where x' is globally fresh. This axiom says that if we load the contents of E into x in a state which looks like Q with a separate single heap cell at location E with contents F, then the resulting state will look the same except that now x will have the value of the contents of E in the pre-state, $F[x'/x]$. As usual [9], we think of x' as standing for the value of x that was overwritten, and the renaming is necessary to account for the changing value of x.

Rearrangement ($\rightarrow_E$). Symbolic execution does not operate on arbitrary pre-states. For instance, the axiom for load (1) requires that the source heap cell be explicitly known. In order to put symbolic heaps into the form required for symbolic execution of a command, we use a rearrangement relation $\rightarrow_E$, defined by the following axioms:

$$\textsc{SubstE}\ \frac{}{z'{=}E \wedge Q \ \rightarrow\ Q[E/z']} \qquad \textsc{SubstN}\ \frac{}{l'{=}N \wedge Q \ \rightarrow\ Q[N/l']}$$

$$\textsc{JunkGT}\ \frac{}{k'{>}N \wedge Q \ \rightarrow\ Q} \qquad \textsc{JunkLT}\ \frac{}{N{>}k' \wedge Q \ \rightarrow\ Q} \qquad \textsc{Transitivity}\ \frac{}{N{>}k' \wedge k'{>}N' \wedge Q \ \rightarrow\ N{>}N' \wedge Q}$$

$$\textsc{Junk}\ \frac{}{Q * H(x', E) \ \rightarrow\ Q * \mathsf{junk}} \qquad \textsc{JunkCycle}\ \frac{}{Q * H_0(x', y') * H_1(y', x') \ \rightarrow\ Q * \mathsf{junk}}$$

$$\textsc{AppendLsNil}\ \frac{}{Q * H_0(E, x') * H_1(x', F) \ \rightarrow\ Q * \mathsf{ls}^{k''}(E, \mathsf{nil})}\ Q \vdash F{=}\mathsf{nil}$$

$$\textsc{AppendLsGuard}\ \frac{}{Q * H_0(E, x') * H_1(x', F_0) * H_2(F_1, G) \ \rightarrow\ Q * \mathsf{ls}^{k''}(E, F_0) * H_2(F_1, G)}\ Q \vdash F_0{=}F_1$$

Here formulæ $H(E, F)$ are of form $E{\mapsto}F$ or $\mathsf{ls}^N(E, F)$; and k', x', y' do not occur other than where explicitly indicated; and k'' is fresh.

Fig. 3. Abstraction relation ($\rightarrow$)

$$\begin{array}{rcll}
Q * F{\mapsto}G & \rightarrow_E & Q * E{\mapsto}G & \text{if } Q \vdash E{=}F \quad \textsc{Switch} \\
Q * \mathsf{ls}^1(F, G) & \rightarrow_E & Q * E{\mapsto}G & \text{if } Q \vdash E{=}F \quad \textsc{SwitchLs} \\
Q * \mathsf{ls}^k(F, G) & \rightarrow_E & k'{>}k \wedge k'{=}1 \wedge Q[k'/k] * E{\mapsto}G & \text{if } Q \vdash E{=}F \quad \textsc{Unroll1} \\
Q * \mathsf{ls}^k(F, G) & \rightarrow_E & k'{>}k \wedge Q[k'/k] * E{\mapsto}x' * \mathsf{ls}^k(x', G) & \text{if } Q \vdash E{=}F \quad \textsc{Unroll}{>}1
\end{array}$$

where k' and x' are globally fresh. Note that these axioms are directed toward a heap location of interest E, increasing the determinacy of symbolic execution.

Rearrangement reveals the pointer structure of heaps which are abstracted or summarized (by an ls predicate). This is achieved by performing case analysis: a symbolic heap rewrites to a set of symbolic heaps, each of which, modulo renaming k, is logically stronger (represents fewer concrete states). Given that we are proving ℓ-termination, it is also crucial for rearrangement to track the changing depths of list segment predicates. This is captured by the $k'{>}k$ in the right-hand side of Unroll$>$1, which indicates that the length of the list starting from x' in the post-state is less than that of the list starting from x in the pre-state.

Abstraction for Fixed-Point Computations ($\rightarrow$). Abstraction is accomplished by certain separation logic implications that rewrite a symbolic heap to a logically weaker one. The abstraction relation on symbolic heaps $Q_0 \rightarrow Q_1$ is defined by the axioms shown in Fig. 3.

As opposed to rearrangement above, which takes lists apart and strengthens the individual symbolic heap formulæ in a symbolic state, abstraction constructs larger lists, weakening the symbolic heap formulæ. This step is very coarse for depth information since, in the examples we have investigated, *increasing* list lengths are not generally a progress measure for ℓ-termination—instead it is

$$\textsc{Crash}\quad \frac{Q \overset{P(n)}{\rightsquigarrow} \top}{\langle n\,, Q\rangle \rightsquigarrow_P \top} \qquad \textsc{Heap Access}\quad \frac{Q_0 \rightarrow_E Q_2 \qquad Q_2 \overset{A(E)}{\rightsquigarrow} Q_3 \qquad Q_3 \rightarrow^* Q_1}{\langle n\,, Q_0\rangle \rightsquigarrow_P \langle n+1\,, Q_1\rangle}\ P(n) \equiv A(E)$$

$$\textsc{Safe}\quad \frac{Q_0 \overset{S}{\rightsquigarrow} Q_2 \qquad Q_2 \rightarrow^* Q_1}{\langle n\,, Q_0\rangle \rightsquigarrow_P \langle n+1\,, Q_1\rangle}\ P(n) \equiv S \qquad \textsc{Goto}\quad \frac{}{\langle n\,, Q\rangle \rightsquigarrow_P \langle m\,, Q\rangle}\ P(n) \equiv \texttt{goto}\, m$$

$$\textsc{If False}\quad \frac{}{\langle n\,, Q\rangle \rightsquigarrow_P \langle m\,, E{=}F \wedge Q\rangle}\ \begin{array}{l} P(n) \equiv \texttt{if}(E{\neq}F)\ \{\texttt{goto}\, l\}\ \texttt{else}\ \{\texttt{goto}\, m\} \\ \text{and } Q \nvdash E{\neq}F \end{array}$$

Fig. 4. Transition relation ($\rightsquigarrow_P$)

decreasing list lengths, captured by the rearrangement relation, which furnish progress measures for ℓ-termination.

The Transition Relation ($\rightsquigarrow_P$). In the Mutant algorithm, for a program P, $\textsc{Sonar}_P = \rightsquigarrow_P$. The transition relation $\rightsquigarrow_P$ relates configurations consisting of a program location and a symbolic heap to another program location and a symbolic heap or crash (notationally $\top$): $\langle n\,, Q_0\rangle \rightsquigarrow_P \langle m\,, Q_1\rangle$ or $\langle n\,, Q_0\rangle \rightsquigarrow_P \top$ where $m \leq \mathit{end}$ and $n \leq \mathit{end} - 1$ are values of the program counter. The program stops when execution reaches *end* either by falling through an S or $A(E)$ instruction, or by a goto. That is, configurations $\langle \mathit{end}\,, Q_0\rangle$ are stuck.

The rules shown in Fig. 4 define the transition relation in terms of the symbolic execution relation $\rightsquigarrow$, the rearrangement relation $\rightarrow_E$, and the reflexive transitive closure of the abstraction relation $\rightarrow^*$. We have shown only one of the four axioms for conditional branches; the others can be defined similarly from [7].

The key rule is Heap Access, which says that when the current instruction will attempt to access a heap cell E, the symbolic state Q_0 is first rearranged to reveal the heap cell at E, yielding state Q_2, from which the current instruction is executed, yielding state Q_3, which is then abstracted, yielding the final state Q_1. The definition allows for flexibility regarding the amount of abstraction that is performed, and how often. By default, Sonar fully abstracts at each step, but when this strategy loses too much precision to prove memory safety, we abstract fully only at the program point ℓ in question.

The first call $Y := \textsc{Sonar}^*_P[\{I\}]$ of the analysis in the Mutant algorithm requires that the transition relation $\rightsquigarrow_P$ (*i.e.*, $\textsc{Sonar}_P$) be an over-approximation of the concrete semantics in the usual sense: that it over-approximates reachability. If σ_1 is a concrete state that is reachable from an initial state σ_0 satisfying initial symbolic heap I, then there is a reachable symbolic heap Q that is satisfied by σ_1. The concrete semantics does not operate on depth variables, and the relevant notion of satisfaction involves existentially quantifying *all* of the depth variables in Q. This sense of over-approximation follows essentially from the soundness result of [7].

The second call to Sonar in the algorithm requires a different notion of over-approximation for *transitions*, which we discuss in Section 4.

A Small Example. To see how this analysis tracks the progress of heap updates for ℓ-termination proofs we consider advancing a pointer to a list to the next node. The initial state $i_s{=}i \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x) * \mathsf{ls}^i(x,\mathsf{nil})$ indicates that the heap shape is an acyclic singly-linked list of length $j+i$ starting from pointer y and ending with nil. The pointer x splits this list into two sublists of length j and i. Consider the program fragment: `n: x = x->next;`. In MUTANT's input format this is represented by a program P where $P(n) = x{:=}[x]$. For this example, $\leadsto_P$ contains two transitions, one for the case where the sublist $\mathsf{ls}^i(x,\mathsf{nil})$ is of length 1:

$$\begin{aligned}&\langle n\,,\, i_s{=}i \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x) * \mathsf{ls}^i(x,\mathsf{nil})\rangle \\ &\qquad \leadsto_P \langle n+1\,,\, x{=}\mathsf{nil} \wedge i_s{>}i \wedge i_s{=}1 \wedge j_s{=}j \wedge \mathsf{ls}^{k''}(y,\mathsf{nil})\rangle\end{aligned} \tag{2}$$

and one for the case where $\mathsf{ls}^i(x,\mathsf{nil})$ is of length greater than 1:

$$\begin{aligned}&\langle n\,,\, i_s{=}i \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x) * \mathsf{ls}^i(x,\mathsf{nil})\rangle \\ &\qquad \leadsto_P \langle n+1\,,\, i_s{>}i \wedge j_s{=}j \wedge \mathsf{ls}^{k''}(y,x) * \mathsf{ls}^i(x,\mathsf{nil})\rangle\end{aligned} \tag{3}$$

Here, we have seeded the initial state with $i_s{=}i$ and $j_s{=}j$ to keep track of the initial depths i_s and j_s so that we can observe that $i_s{>}i$. This indicates that the sublist $\mathsf{ls}^i(x,\mathsf{nil})$ in the post-state is shorter than that in the pre-state. It is inequalities like this which are the reason for well-foundedness of the computed transition relations.

In the derivations of the transitions (2) and (3), first the state $i_s{=}i \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x) * \mathsf{ls}^i(x,\mathsf{nil})$ is rewritten to two intermediate states by the application of the rearrangement axioms UNROLL1 and UNROLL>1, respectively:

$$\begin{aligned}&i_s{=}i \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x) * \mathsf{ls}^i(x,\mathsf{nil}) \\ &\qquad \rightarrow_x i_s{=}i' \wedge i'{>}i \wedge i'{=}1 \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x) * x{\mapsto}\mathsf{nil}\end{aligned} \tag{4}$$

$$\begin{aligned}&i_s{=}i \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x) * \mathsf{ls}^i(x,\mathsf{nil}) \\ &\qquad \rightarrow_x i_s{=}i' \wedge i'{>}i \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x) * x{\mapsto}x' * \mathsf{ls}^i(x',\mathsf{nil})\end{aligned} \tag{5}$$

Now, in both of the resulting states, the heap cell at x is explicit, therefore the symbolic execution rules can be applied to the right-hand side of (4) and (5):

$$\begin{aligned}&i_s{=}i' \wedge i'{>}i \wedge i'{=}1 \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x) * x{\mapsto}\mathsf{nil} \\ &\qquad \overset{x:=[x]}{\rightsquigarrow} x{=}\mathsf{nil} \wedge i_s{=}i' \wedge i'{>}i \wedge i'{=}1 \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x') * x'{\mapsto}\mathsf{nil}\end{aligned}$$

$$\begin{aligned}&i_s{=}i' \wedge i'{>}i \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x) * x{\mapsto}x' * \mathsf{ls}^i(x',\mathsf{nil}) \\ &\qquad \overset{x:=[x]}{\rightsquigarrow} x{=}x' \wedge i_s{=}i' \wedge i'{>}i \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x'') * x''{\mapsto}x' * \mathsf{ls}^i(x',\mathsf{nil})\end{aligned}$$

Finally, the resulting states are abstracted by $\rightarrow$. For the first state, we apply the SUBSTN and APPENDLSNIL rules to abstract as much as possible, and for the second state we apply SUBSTN, SUBSTE, and APPENDLSGUARD, yielding:

$$x{=}\mathsf{nil} \wedge i_s{=}i' \wedge i'{>}i \wedge i'{=}1 \wedge j_s{=}j \wedge \mathsf{ls}^j(y,x') * x'{\mapsto}\mathsf{nil} \tag{6}$$

$$\begin{array}{c} \rightarrow^* x{=}\mathsf{nil} \wedge i_s{>}i \wedge i_s{=}1 \wedge j_s{=}j \wedge \mathsf{ls}^{k''}(y, \mathsf{nil}) \\ x{=}x' \wedge i_s{=}i' \wedge i'{>}i \wedge j_s{=}j \wedge \mathsf{ls}^{j}(y, x'') * x''{\mapsto}x' * \mathsf{ls}^{i}(x', \mathsf{nil}) \\ \rightarrow^* i_s{>}i \wedge j_s{=}j \wedge \mathsf{ls}^{k''}(y, x) * \mathsf{ls}^{i}(x, \mathsf{nil}) \end{array} \quad (7)$$

The right-hand sides of (6) and (7) are the resulting states of (2) and (3).

4 Checking Well-Foundedness of the Over-Approximation

Now that the abstract transition relation has been described, the remaining ingredients of the ℓ-termination proof method of Fig. 2 are SEED and WF.

Considering Fig. 2, we know that y is an element of Y (*i.e.*, of $\textsc{Sonar}^*_P[\{I\}]$). $\textsc{Seed}(y)$ computes s, which is a new state in which the values of all the depth variables occurring in y are symbolically recorded:

$$\textsc{Seed}(Q) \triangleq (\textstyle\bigwedge_{k \in fdv(Q)} k_s{=}k) \wedge Q$$

where $fdv(Q)$ denotes the (unprimed or primed) depth variables in Q. Each k_s is a fresh symbolic constant (*i.e.*, uninterpreted nullary function symbol), which we formally represent as an unprimed variable. Now assume, as is done in Fig. 2, that $z \in \textsc{Sonar}^+_P[\{s\}]$.

The procedure $\mathrm{WF}(z)$ is used to try to prove that z represents a well-founded binary relation. Formally, this procedure proves well-foundedness of the relation that represents an over-approximation (determined by (y, z)) of all concrete executions that visit y and then visit z.

WF implements this procedure by first computing a representative set of arithmetic inequalities. Due to seeding and the fact that symbolic execution maintains the relationship between the seeded information and the updated information, the arithmetic component of the relation (y, z) represents an over-approximation of the changes in depths due to executing from y to z, and exists entirely in the symbolic state z. Hence we can extract them via a projection $\alpha(z)$:

$$\begin{array}{rr} \alpha(\mathsf{true} \wedge \Sigma) \triangleq \mathsf{true} & \alpha(K \wedge \Sigma) \triangleq K \\ \alpha(B \wedge \Sigma) \triangleq \mathsf{true} & \alpha(\Pi_0 \wedge \Pi_1 \wedge \Sigma) \triangleq \alpha(\Pi_0 \wedge \Sigma) \wedge \alpha(\Pi_1 \wedge \Sigma) \end{array}$$

The formula resulting from this projection is interpreted as a binary relation over the naturals from the seed variables to the other variables. As an example, consider this abstraction applied to the post-state of the transition of the previous section (3): $\alpha(i_s{>}i \wedge j_s{=}j \wedge \mathsf{ls}^{k''}(y, x) * \mathsf{ls}^{i}(x, \mathsf{nil})) = i_s{>}i \wedge j_s{=}j$. We now take this as one of the disjuncts in the transition invariant from Section 2, $T_n = i_s{>}i \wedge j_s{=}j$, which clearly represents a well-founded relation from i_s, j_s to i, j, over the naturals.

After projecting out the inequalities, WF calls the RANKFINDER tool [11] to attempt to prove well-foundedness:

$\mathrm{WF}(z)$ **{ return** ($\textsc{RankFinder}(\alpha_k(z))$ reports "Rank function found") **}**

Checking well-foundedness relies on a second notion of over-approximation that is relevant to the second call $Z := \textsc{Sonar}_P^+[\{s\}]$ of the depth analysis in the MUTANT algorithm. The formulation and proof of this second sense of over-approximation is non-trivial, and for space reasons we can only give an outline of it here. It involves setting up an instrumented semantics which manipulates the depth variables k; the reason for the additional semantics is that the standard concrete semantics of heap mutation does not mention the auxiliary depth variables used in our analysis. The instrumented semantics mixes both concrete and abstract semantics. For example, an assignment statement $[x]{:=}\, y$ alters concrete heap cell x, but can also bump a depth variable down by one, corresponding to an application of the UNROLL>1 rearrangement rule. The crucial point is that the rearrangement rules ($\rightarrow_E$) are sound for the updates of depth variables in the instrumented semantics. Overall, what we require, first, is that if the projection $\alpha(Q)$ denotes a well-founded relation, then that implies well-foundedness of executions in the instrumented semantics starting from seeded states; in essence, α constrains the changes to depth variables. Then, the soundness of MUTANT requires a simulation argument connecting the instrumented semantics with a standard concrete semantics of heap mutation.

5 A Complete Example

To illustrate the analysis in action, we consider trying to prove ℓ-termination of the simple program in Fig. 5, where ℓ is location 5, and the initial state is $\mathsf{ls}^k(x, \mathsf{nil})$. As in Fig. 2, the first step of $\textsc{Mutant}(P, I, 5)$ is to compute $\textsc{Sonar}_P^*[I]$. First SONAR computes the transition relation:

$$\begin{aligned}
\langle 3\,,\, \mathsf{ls}^k(x,\mathsf{nil})\rangle &\rightsquigarrow_P^* \langle 5\,,\, y{=}x \wedge \mathsf{ls}^k(x,\mathsf{nil})\rangle\\
\langle 5\,,\, y{=}x \wedge \mathsf{ls}^k(x,\mathsf{nil})\rangle &\rightsquigarrow_P^+ \langle 5\,,\, y{\mapsto}x * \mathsf{ls}^k(x,\mathsf{nil})\rangle\\
\langle 5\,,\, y{\mapsto}x * \mathsf{ls}^k(x,\mathsf{nil})\rangle &\rightsquigarrow_P^+ \langle 5\,,\, \mathsf{ls}^{k'}(y,x) * \mathsf{ls}^k(x,\mathsf{nil})\rangle\\
\langle 5\,,\, \mathsf{ls}^{k'}(y,x) * \mathsf{ls}^k(x,\mathsf{nil})\rangle &\rightsquigarrow_P^+ \langle 5\,,\, \mathsf{ls}^{k'}(y,x) * \mathsf{ls}^k(x,\mathsf{nil})\rangle
\end{aligned}$$

We show only those transitions in $\rightsquigarrow_P^+$ involving program location 5, since the algorithm will consider only those states. In this case:

C program	program in SONAR format

`1 void main()`
`2 {`
`3  y = x;`
`4  while (x!=NULL) {`
`5  x = x->next`
`6  }`
`7 }`
	$P(3) = y{:=}\,x$
	$P(4) = \texttt{if}(x{\neq}\mathsf{nil})\ \{\texttt{goto}\ 5\}\ \texttt{else}\ \{\texttt{goto}\ 7\}$
	$P(5) = x{:=}\,[x]$
	$P(6) = \texttt{goto}\ 4$

Fig. 5. Simple example program: list traversal

$$Y = \{\langle 5\ ,\ y{=}x \wedge \mathsf{ls}^{k}(x,\mathsf{nil})\rangle, \langle 5\ ,\ y{\mapsto}x * \mathsf{ls}^{k}(x,\mathsf{nil})\rangle, \langle 5\ ,\ \mathsf{ls}^{k'}(y,x) * \mathsf{ls}^{k}(x,\mathsf{nil})\rangle\} \cup Y'$$

where Y' contains the states not at program location 5. Note that $\top \notin Y$, meaning that executing P from I is guaranteed to be safe with respect to the basic set of (memory) safety properties we consider.

Of the three reachable states in Y, we need only to consider $q = \langle 5\,,\mathsf{ls}^{k'}(y,x) * \mathsf{ls}^{k}(x,\mathsf{nil})\rangle$ since execution from the other two states will result either in the state $\langle 5\,,\mathsf{ls}^{k'}(y,x) * \mathsf{ls}^{k}(x,\mathsf{nil})\rangle$ itself or in the loop exiting. The next step in Fig. 2 is to seed q, which yields: $s = \langle 5\,,k'_s{=}k' \wedge k_s{=}k \wedge \mathsf{ls}^{k'}(y,x) * \mathsf{ls}^{k}(x,\mathsf{nil})\rangle$. The variables k and k' are set equal to the fresh constants k_s and k'_s. Later, during the successive call to SONAR, we will be able to see how the values of k and k' change relative to k'_s and k_s.

The next step is to compute $(\text{SONAR}_P)^+[\{s\}]$, which equals:

$$\langle 5\ ,\ k'_s{=}k' \wedge k_s{=}k \wedge \mathsf{ls}^{k'}(y,x) * \mathsf{ls}^{k}(x,\mathsf{nil})\rangle \ \rightsquigarrow_P^+\ \langle 5\ ,\ k_s{>}k \wedge \mathsf{ls}^{k'}(y,x) * \mathsf{ls}^{k}(x,\mathsf{nil})\rangle$$
$$\langle 5\ ,\ k_s{>}k \wedge \mathsf{ls}^{k'}(y,x) * \mathsf{ls}^{k}(x,\mathsf{nil})\rangle \ \rightsquigarrow_P^+\ \langle 5\ ,\ k_s{>}k \wedge \mathsf{ls}^{k'}(y,x) * \mathsf{ls}^{k}(x,\mathsf{nil})\rangle$$

From this we see that the set of states at program location 5 reachable from s after executing the loop one or more times is $Z = \{\langle 5\ ,\ k_s{>}k \wedge \mathsf{ls}^{k'}(y,x) * \mathsf{ls}^{k}(x,\mathsf{nil})\rangle\}$. Let r be the element in Z from Fig. 2: $r = \langle 5\ ,\ k_s{>}k \wedge \mathsf{ls}^{k'}(y,x) * \mathsf{ls}^{k}(x,\mathsf{nil})\rangle$. All that remains is to prove $\text{WF}(r)$, which we do by calculating: $\alpha_k(r) = k_s{>}k$ and then calling RANKFINDER$(k_s{>}k)$. In this case RANKFINDER reports that the relation is well-founded.

6 Experimental Results

In the experimental results described in [4], TERMINATOR [5] was used to try to prove that Windows device driver dispatch routines always return to their calling context. A number of false bugs were reported in those experiments due to TERMINATOR's inaccuracy with respect to heaps. In this section we revisit 21 loops from [4] in which ℓ-termination was not provable. Fig. 6 displays the results of these experiments (which were run on a 3.6GHz Pentium 4 machine). The symbol ✓ is used to indicate the 16 cases in which MUTANT was able to prove ℓ-termination. The symbol ⊘ is used to represent failed proof attempts.

Loop	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Time (s)	0.0	0.0	8.0	0.3	1.7	13	296	0.1	5.4	0.0	8.2	821	0.0	1.6	152	0.0	2.6	3.5	58	32	261
Result	✓	⊘	✓	✓	✓	✓	✓	⊘	⊘	✓	✓	✓	✓	✓	✓	✓	⊘	✓	⊘	✓	✓
WF checks	1	4	16	3	5	9	15	2	4	1	6	39	1	3	16	1	28	9	85	20	37

Fig. 6. Results of experiments using MUTANT on loops from extracted from Windows device drivers falsely reported as non-terminating by TERMINATOR (see [4]). The symbol ✓ indicates that MUTANT was able to prove the loop ℓ-terminating; The symbol ⊘ means that a termination bug was found.

The number of disjuncts in the transition invariant, described in Section 2, is reported in the bottom row.

Each failed well-foundedness check leads directly to a counterexample in the code (the production of counterexamples could be automated but isn't in the current setup). Note that for now we have to extract each loop from a Windows device driver loop by hand: MUTANT currently does not support C functions and address-of (&) operator on stack variables, so some manual translation akin to a compiler front-end was required to construct equivalent programs. Furthermore, loop preconditions were inserted by hand. These preconditions could probably be automatically computed via the analysis like the one described in [7].

Example 19 is the code from Fig. 1. As we see from Fig. 6, this loop has a termination bug. The problem is that `InitializeListHead` creates a self-loop from `&irp->Tail.Overlay.ListEntry.Flink` to `&irp->Tail.Overlay.ListEntry` and that `irp->Tail.Overlay.ListEntry.Flink` aliases `entry`, meaning that after the call to `InitializeListHead`, `entry` equals `entry->Flink`. Example 18 is based on fixed code provided to us from the Windows kernel team after we reported the bug. These experiments reveal a strong difference in MUTANT's running time between analyzing terminating versus non-terminating loops.

Example 8 is the only false bug reported by MUTANT: the loop actually does ℓ-terminate, but our analysis is unable to prove it. This example amounts to reversing a panhandle list. The initial state describes such lists, which cycle back to a list node other than the head node, with the formula: $\mathsf{ls}^i(c, x') * \mathsf{ls}^j(x', y') * \mathsf{ls}^k(y', x')$. The program is essentially a common in-place list reversal algorithm. When the program is run starting from a panhandle list, first the handle is reversed in the usual way, then the cycle is reversed, and finally the handle is reversed once again. Notice, in particular, that the handle is walked twice, and so the quantity which is decreasing with each loop iteration is $2i + j + k$, which our analysis does not detect. Finally, note that MUTANT correctly proves the termination of list reversal when starting either with an acyclic or cyclic list.

7 Conclusion

In this paper we have introduced a novel method of automatically proving the termination of loops whose correctness depends on the mutation of the heap. As the experimental results demonstrate, MUTANT is able to prove the termination of loops that TERMINATOR was previously unable to handle. MUTANT is completely automatic (*e.g.*, it does not require the user to provide ranking functions). MUTANT provides information which may lead to concrete counterexamples when a termination proof fails.

Related Work. Our work differs from the previous research on termination proof methods in that we have proposed the first known tool to support entirely automatic termination proofs of imperative programs with deep heap updates. To the best of our knowledge, the experimental results in Section 6 represent the first known successful application of this type of tool to industrial systems with loops that imperatively construct or destruct heap-based data structures. Note

that absolutely no user intervention is required (*i.e.*, ranking functions or proof hints). Yahav's dissertation [15] discusses experiments in which imperative list-processing loops are proved terminating (the programs come from [8]). This work is less automatic than MUTANT: for the reason that the user must first examine the loop and specify a single (possibly lexicographically ordered) ranking function. MUTANT/TERMINATOR automatically proves all of the examples from [8] in less than 10s total. Note that 6.5s of this 10s was spent solving the one arithmetic (non-heap) example using the standard TERMINATOR algorithm.

MUTANT also uses the relatively new TERMINATOR proof-rule (finding a disjunctively well-founded over-approximation), which was originally proposed in [12]. While this use is not an original contribution, it means that the flavor of the analysis is different from previous approaches (such as [16], [10] or [6]).

Our algorithm works in reverse order with respect to TERMINATOR's original method for arithmetic programs. TERMINATOR iteratively refines the set of well-founded relations based on false counterexamples to the termination property. The relations are well-founded by construction, the difficultly is proving that they over-approximate the meaning of the loop or recursive function. MUTANT first computes an over-approximation and proves that it is disjunctively well-founded. The over-approximation is given, the question is *are the disjuncts well-founded*?

Acknowledgments

We are grateful to Andreas Podelski, Andrey Rybalchenko, Moshe Vardi, Tal Lev-Ami, Roman Manevich, Noam Rinetzky, Mooly Sagiv, Eran Yahav, and Greta Yorsh for discussions, and the anonymous referees for helpful comments. Distefano and O'Hearn acknowledge support from the EPSRC.

References

[1] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.
[2] A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
[3] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *SAS*, 2005.
[4] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
[5] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In *CAV*, 2006.
[6] D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? on the automated verification of linked list structures. In *FSTTCS*, 2004.
[7] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
[8] N. Dor, M. Rodeh, and S. Sagiv. Checking cleanness in linked lists. In *SAS*, 2000.
[9] R. W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, 1967.

[10] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL*, 2001.
[11] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.
[12] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, 2004.
[13] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, 2005.
[14] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
[15] E. Yahav. *Property-Guided Verification of Concurrent Heap-Manipulating Programs*. PhD thesis, 2004.
[16] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *ESOP*, 2003.