From Shape Analysis to Termination Analysis in Linear Time

Roman Manevich[1(✉)], Boris Dogadov[2], and Noam Rinetzky[2]

[1] Ben-Gurion University of the Negev, Beer-Sheva, Israel
romanm@cs.bgu.ac.il
[2] Tel Aviv University, Tel Aviv, Israel
{borisdog,maon}@post.tau.ac.il

Abstract. We present a novel algorithm to conservatively check whether a (recursive) heap-manipulating program terminates. Our algorithm can be used as a post-processing phase of any shape analysis satisfying some natural properties. The running time of the post-processing phase is linear in the size of the output of the chosen shape analysis.

The main idea is to partition the (unbounded but finite) set of allocated objects in every state into a bounded set of regions, and track the flow of objects between heap regions in every step of the program. The algorithm proves the existence of the well-founded relation over states by showing that in every loop iteration at least one object (which was allocated before entering the loop) moves to a strictly lower-ranked heap region. The partitioning of objects into regions, the flow of objects between regions, and the ranks of regions are computed automatically from the output of the underlying shape analysis. Our algorithm extends the state of the art in terms of complexity, the class of supported data structures, and its generality.

We successfully applied a prototype of our analysis to prove termination of a suite of benchmarks from existing literature, including (looping, recursive, and concurrent) list manipulating programs, looping list-sorting programs, and looping programs that manipulate trees and graphs. The overhead of the termination phase in our experiments is at most 14 % of the overall analysis time.

1 Introduction

Proving termination of heap manipulating programs is both important and challenging. System codes, whose reliability is crucial for the stack of software built on top of them, often use low-level manipulation of linked data structures to gain efficiency. Proving termination requires synthesizing ranking functions that correlate two different types of unbounded data—data structure invariants and the number of loop iterations. This in turn requires tracking highly complex shape-numeric invariants [14,19].

This work was funded by EU FP7 project ADVENT (308830) and by the Broadcom Foundation and Tel Aviv University Authentication Initiative.

S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part I, LNCS 9779, pp. 426–446, 2016.
DOI: 10.1007/978-3-319-41528-4_23

Almost all existing approaches (with Brotherson et al. [8,9] as a notable exception) address the undecidable problem of proving termination of heap-manipulating programs by reducing it to another undecidable problem, e.g., proving termination of a numeric program [4,5,19], proving termination of a term-rewriting system [21], or proving the termination of a constraint-logic program [27]. In contrast, we take a direct approach. In the first phase, we apply any program-wide shape analysis that partitions the heap into finitely many heap regions and computes the following two relations: (i) an abstract transition relation, and (ii) an abstract relation that tracks how program statements cause individual objects to change membership in heap regions. In the second phase, we conservatively check termination by solving a linear time problem over these two relations. By decoupling the reasoning used to synthesize heap invariants from the reasoning used for checking termination, we gain the following notable advantages:

- We avoid the need to track complex heap measures during the shape analysis.
- Our termination checking phase is agnostic to most details of the programming language. Its only requirement is that the loops form a hierarchical tree structure.
- Our termination checking phase can be latched on top of any shape analysis, which satisfies certain natural assumptions. This makes our approach quite flexible, allowing to adapt the shape analysis to a given class of programs and data structures[1] and to support combined shape domains [28,29].
- Our algorithm handles recursive procedures quite precisely via the call stack-as-list approach [24,25]. As far as we know, we are the first to automatically prove termination of recursive heap-manipulating procedures with side-effects.
- We can apply summarization to the result of the shape analysis phase. Then, we can check the termination of each loop and procedure only with respect to these summaries in a modular way.

Overview of Our Approach. The termination analysis begins by processing the abstract transition relation computed by the underlying shape analysis to produce a bounded graph which conservatively tracks the flow of objects between heap regions. The nodes of the graph are pairs of abstract states and regions. The graph contains two kind of edges, pertaining to over- and under- approximation information: *evolution edges* and *must-evolution edges*. The former over-approximates the object flow information by refining every abstract transition into a set of *evolution edges* going from every region in the source abstract state to every region in the target abstract state which *may* contain some of its objects, while the latter under-approximates the flow information by producing a similar set of evolution edges which records *must flow* information.

[1] For example, by choosing separation logic approaches for programs manipulating inductive data structures and shape graph approaches for programs manipulating arbitrary graph data structures.

Using the (may) evolution graph, we compute an acyclic graph, called the *condensation graph*, by collapsing strongly-connected components and removing self-loops. The condensation graph entails a well-founded relation over heap regions: Every region is given a *rank* based on its depth in the condensation graph, and as the latter is acyclic and the number abstract states and regions is bounded, there is a bounded number of ranks. This well-founded relation can be point-wise lifted to a well-founded relation over abstract states which maps every region in every state to its rank. The resulting relation is preserved by any abstract transition (i.e., the either stays the same of strictly decreases). To prove that every loop in the program terminates, we find for every cycle in the abstract transition relation terminates, a *cutting set*—a set of transitions that strictly decrease this relation in every iteration through the cycle. Specifically, our algorithm looks for a cutting set containing only must-evolution edges.

Main Contributions. The main contributions of this paper are summarized below:

1. **Novel Well-founded Relation Over Heaps.** We show how to use the result of a shape analysis to construct a well-founded relation over the transitions of any heap-manipulating program (Sect. 5). This avoids the need to synthesize a different well-founded relation for each program. It also allows us to dynamically find the best set of cut points in the abstract transition relation, in contrast to existing analyses that choose them a priori.
2. **Generic Framework.** Our algorithm (Sect. 3) is parameterized by (the atomic statements of) the programming language (Sect. 2) and a class of shape analyses satisfying very weak and natural assumptions (Sect. 4). It is independent of the iteration strategy used by the shape analysis, e.g., whether it is forward or backward, compositional [10], and whether it uses widening. This allows us to mix and match existing shape analyses, and take advantage of any advances in this field by simply replacing the underlying shape analysis. We utilize this flexibility to support recursion (Sect. 6).
3. **Featherweight Reasoning.** Our termination checking phase analyzes loops and recursive procedures separately by summarizing their behavior. Its overall running time is linear time in the size of the program and the output of the shape analysis. The predictable performance makes it an appealing upgrade for existing shape analyses.
5. **Experimental Evaluation.** We have implemented a prototype of our analysis and applied it to a suite of benchmarks (Sect. 7). Our analysis successfully handles programs operating on inductive data structures as well as a garbage-collection benchmark operating on arbitrary graphs, programs with recursive procedures, and intricate examples introduced by previous works. We have also proved lock-freedom for concurrent programs manipulating linked lists with a constant number of threads.

2 Programming Language and Running Example

We formalize our results using a simple imperative programming language for sequential procedure-less programs, which we later extend to handle (recursive) procedures and bounded parallelism. We assume that the reader is familiar with the way programs are represented using *control-flow graphs* (see, e.g., [1,20]), and only define the necessary terminology.

Syntax. A program $P = (V, E, v_{entry}, v_{exit}, c)$ is a *control-flow graph (CFG)*: Nodes $v \in V$ correspond to *program points* and edges $e \in E$ indicate possible transfer of control. Every node v is associated with a primitive command $c(v)$. P starts its execution at program point v_{entry} and terminates at v_{exit}. We assume that the program has a *reducible* CFG, i.e., every loop has a single *entry node*, which we refer to as the *loop header*. For simplicity, we assume that every loop has a unique entry node which is also its unique *exit node*, and identify loops by their headers. We also assume E does not contain self loops, i.e., $\forall v \in V. (v, v) \notin E$.

We say that $L = (V', E', v'_{entry}, v'_{exit}, c|_{V'})$ is the *sub-CFG* of P pertaining to V' if $\{v'_{entry}, v'_{exit}\} \subseteq V' \subseteq V$, $E' = E \cap V' \times V'$, v'_{entry} is the only node in V' into which control can flow in P from program points outside V' and v'_{exit} is the only node in V' from which control can leave V'. We denote the entry node and exit nodes of a sub-CFG L by $Entry(L)$ and $Exit(L)$, respectively. We denote the set of sub-CFGs pertaining to the natural loops[2] in the CFG of P by $\textsc{Loops}(P)$.

Our technique is independent of the choice of primitive commands; we only require that they can be handled by the underlying shape analysis. Thus, we leave this set unspecified. In our examples, we use a set of typical commands for heap-manipulating programs: assignments between variables, dynamic allocation and deallocation of objects, (possibly destructive) accesses to fields of objects, and conditionals involving pointer expressions. The meaning of these commands is standard (see, e.g., [23]).

Operational Semantics. The concrete semantics of a program is defined as a *concrete transition relation* $T \subseteq \Sigma \times \Sigma$ over a set of *concrete states* $\sigma \in \Sigma$. We assume an unbounded set $o \in \mathcal{O}$ of *object identifiers* (*objects* for short). We only expect that a concrete state $\sigma = \langle v, A, F, \ldots \rangle$ records the value of the program counter, denoted by $v(\sigma) = v$, the finite, but unbounded, set of *allocated* objects, denoted by $A(\sigma) = A$, and the unbounded set of *free* (unallocated) objects, denoted by $F(\sigma) = F$. Concrete transitions induce a *concrete evolution relation*. That is, a relation that for each $(\sigma, \sigma') \in T$ connects the objects in σ to the corresponding objects in σ'[3]. We place no other restrictions on the form of states or on the transition relation.

[2] Intuitively, a natural loop is the CFG-analogue of a `while` loop: It has a single *header* (entry) node, and every two natural loops are either nested, disjoint, or share the header node. See [1].

[3] Intuitively, this relation is identity, although in reality a garbage collector can relocate objects in memory.

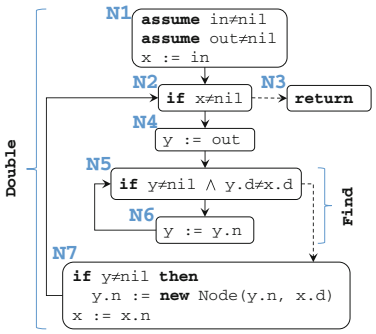

Fig. 1. CFG for `Double` (with extended basic blocks). Dashed edges represent false condition branches.

Running Example. In the rest of the paper, we exemplify our analysis using a simple program in a Java-like language whose control-flow graph (CFG) is shown in Fig. 1. The program operates over two disjoint acyclic singly-linked lists pointed-to by `in` and `out`, respectively. List elements are declared as

```
class Node { Node n; int d; } .
```

The outer loop scans the `in` list and uses the inner loop, shown by the sub-CFG `Find`, to find an identical element in the `out` list. If it finds one, it splices a copy of that element next to it. The sub-CFG of the outer loop is consists of the nodes in the CFG except N1 and N3, and its header is N2. The sub-CFG of the inner loop consists of the nodes N5 and N6, and its header is N5.

3 Termination Analysis

This section defines our termination analysis, which is shown as pseudo-code in Fig. 2.

The analysis is based on the following principles: (i) using a partition-based shape analysis (Sect. 3.1) to approximate the concrete transition relation and to approximate the flow of objects between heap regions by a so-called *evolution relation*; (ii) using the flow of objects between heap regions to define a well-founded relation[4] over heap regions and then lift it to a well-founded relation over the abstract heap descriptors (Sect. 3.2); (iii) using the well-founded relation to discover a *loop cut* for each loop—a set of transitions that strictly decrease the well-founded relation and must be taken in each loop iteration (Sect. 3.3); and (iv) summarizing loops to handle loop nesting (Sect. 3.4).

3.1 Shape Analysis

Our termination analysis is built on top of an underlying shape analysis, which is first applied to the entire program (ObjectFlowTermCheck line 1).

[4] A relation is well-founded if it has no infinite descending chains.

```
ObjectFlowTermCheck(P) {
1   (τ, ev) ⟵ ShapeAnalysis(P)
2   L_1, ..., L_n ⟵ Loops(P)
3   (τ̄[L_1, ..., L_n], ēv[L_1, ..., L_n]) ⟵ Summarize(P, τ, ev)
4   foreach i = 1..n do
5     // Compute well-founded relation over regions, for the loop-specific evolution relation.
6     ev_≺ ⟵ EvolutionToWF(ēv[L_i])
7     τ_↻ ⟵ τ̄[L_i] // Get loop-specific abstract transition relation.
8     // Remove decreasing abstract transitions.
9     foreach (s, s') ∈ τ_↻ do
10      if ∃([(s, ρ)], [(s', ρ')]) ∈ ev_≺(s, s') then
11        τ_↻ ⟵ τ_↻ \ (s, s') //since s ≺ s'
12    NonDecreasingCycles ⟵ {b ∈ SCC(τ_↻)||b| > 1}
13    if NonDecreasingCycles ≠ ∅ then
14      Print "Loop L may not terminate!"
}

EvolutionToWF(ev_L) {
1   {[r_1], ..., [r_m]} ⟵ SCC(May(ev_L))
2   // Merge region ranks possibly containing loop-allocated objects.
3   foreach [r_i] ∈ {[r_1], ..., [r_m]} do
4     if (ρ_free, r_i) ∈ ev_L^* then
5       [ρ_free] ⟵ [ρ_free] ∪ [r_i]
6   // Maintain strictly-decreasing evolution edges.
7   ev_≺ ⟵ ev_L
8   foreach ((s, ρ), (s', ρ')) ∈ ev_≺ do
9     if [(s, ρ)] = [(s', ρ')] ∨ ¬Must((s, ρ), (s', ρ')) then
10      ev_≺ ⟵ ev_≺ \ {((s, ρ), (s', ρ'))}
11  return ev_≺
}
```

Fig. 2. Soundly checking termination. May(ev) and Must(ev) return the subset of evolution may edges and evolution must edges, respectively

$x, z \in \mathsf{Var}$	Variables
$v \in V$	CFG nodes
$E ::= x \mid \mathsf{nil}$	Expressions
$P ::= E = E \mid \neg P$	Simple pure formulae
$Q ::= \mathsf{true} \mid P \mid Q \wedge Q$	Pure formulae
$A ::= (x \mapsto E) \mid \mathsf{ls}(E, E)$	Atomic spatial formulae
$H ::= A \mid \mathsf{emp} \mid H * H$	Spatial formulae
$\langle v, Q \wedge H \rangle$	Symbolic heaps
$\mathsf{ls}(E, F) = E \mapsto F \vee \exists z \,.\, E \mapsto z * \mathsf{ls}(z, F)$	

Fig. 3. Symbolic heaps

The shape analysis must produce two relations—an abstract transition relation and an (abstract) *evolution relation*. The abstract transition relation is a finite graph whose nodes are (abstract) heap descriptors and edges overapproximate the concrete transition relation between the concrete heaps they represent. A heap descriptor consists of a finite set of *heap regions* (or simply *regions*), which partition the set of objects in each concrete heap it represents. The evolution relation is a (finite) graph whose nodes are the regions in the heap descriptors. The edges (over- and under-) approximate the flow of objects between the regions of any two heap descriptors related by an abstract transition. To identify a heap region ρ in a specific heap descriptor s, we write (s, ρ).

To exemplify our analysis, we assume basic familiarity with shape analysis based on separation logic [3] and use the analysis of singly-linked lists [12]. In this analysis, the heap descriptors are represented by *symbolic heaps* and the heap regions are represented by *spatial formulae*, both of which are defined in Fig. 3.

Example 1. Figure 4 shows a concrete transition from state σ to state σ', resulting from executing the statement `y:=y.n`. The objects in each state are partitioned as shown by the spatial formulae annotations. The red edges that connect corresponding objects comprise the so-called concrete evolution relation. Specifically, notice that the transition moves the object e from the region ls(y, nil) to the region ls(out, y).

To approximate the concrete transition relation for a given statement, a separation logic-based analysis applies a sequence of derivation rules. To approximate the concrete evolution relation, each derivation rule must be augmented with two types of edges between the regions of the pre and post symbolic heaps. A may edge means that an object in the source region may exist in the destination region. A must edge means that at least one object in the source region must exist in the destination region. To maintain soundness, may edges can be overapproximated and must edges can be underapproximated.

Figure 5 shows (a simplified subset of) such derivation rules used in analyzing the `Find` loop. For the given rules, the set of evolution may edges and the set of evolution must edges are equal. The double-lined arrows stand for the bijection between the spatial formulae appearing in the pre formula and the corresponding spatial formulae appearing in the post formula.

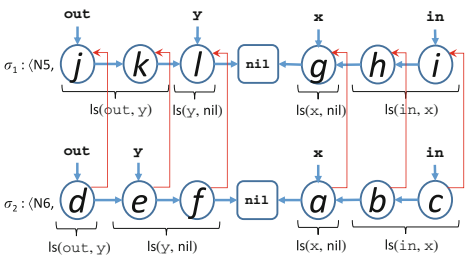

Fig. 4. A concrete transition annotated with evolution edges and spatial formulae

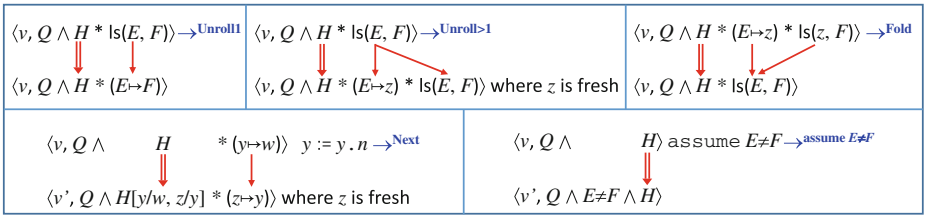

Fig. 5. Derivation rules augmented with abstract evolution edges. For the statement rules, we have that $(v, v') \in E$ and $c(v)$ is the statement at hand

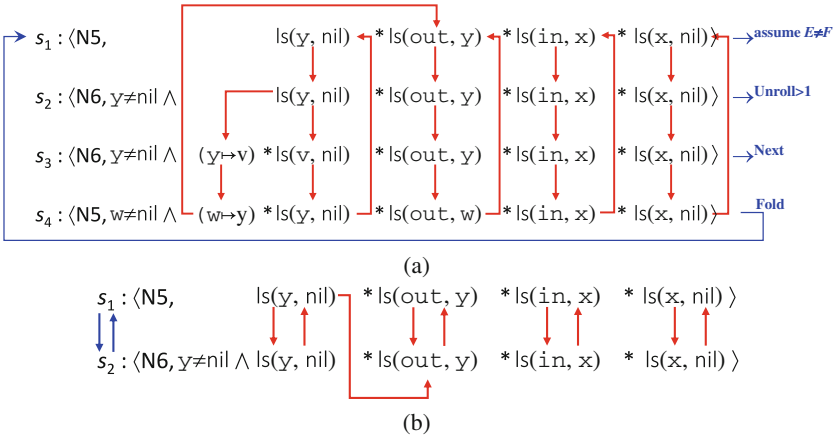

Fig. 6. (a) Symbolic execution augmented with evolution edges, and (b) the resulting transition relation and evolution relation

Figure 6(a) shows how the symbolic heap s_2 is derived from s_1 and how s_1 is derived from s_2 via the intermediate symbolic heaps s_3 and s_4 (redundant variables are removed after each rule application). By composing the relations (using natural join) across the intermediate states of the symbolic execution, we obtain the abstract transitions edges $(s_1, s_2), (s_2, s_1)$ and the evolution relation shown in Fig. 6(b).

The edges $(s_1, s_2), (s_2, s_1)$ form a cycle in the abstract transition relation. *How do we show that every execution represented by this cycle terminates?*

3.2 From Shape Analysis to a Well-Founded Relation

Consider a concrete execution whose states are represented by s_1 and s_2. Every concrete transition represented by the abstract transition (s_2, s_1) shrinks the region $\mathsf{ls}(\mathtt{y}, \mathsf{nil})$ of s_2 by moving an object (e in our example) to the region $\mathsf{ls}(\mathtt{out}, \mathtt{y})$ of s_1 (i.e., from the suffix of the list to its prefix, relative to $\mathtt{y}$). Furthermore, this transition appears infinitely often in every infinite path that contains this cycle. Since no objects flow into this region other than from itself, after

taking this transition a number of times that is bounded by the number of cells in the `out` list, this region becomes empty and the execution must exit the cycle.

In **EvolutionToWF**, line 1, the procedure accepts the evolution relation induced by the transitions between CFG nodes of a loop and uses the may edges to compute its strongly-connected components in linear time [11]. It then produces a so-called *condensation graph* by collapsing the set of regions in each component to a single node and removing self-loops. This results in a finite and acyclic graph, which is therefore a well-founded relation. The *region rank*, or simply *rank*, of a region (s, ρ), denoted $[(s, \rho)]$, is the component in the condensation to which it belongs. We write $[(s, \rho)] \preceq [(s', \rho')]$ if $[(s', \rho')]$ is reachable from $[(s, \rho)]$ in the condensation graph. Figure 7 shows the condensation graph induced by s_1 and s_2.

The well-founded relation $\cdot \preceq \cdot$ induces a well-founded relation over the concrete states represented by the abstract transition relation. Intuitively, we map each object o in a concrete state to the rank of the region representing o, denoted by $rank(o)$. We then lift this relation to pairs of concrete states related by the execution of a statement by comparing the objects in the post-state to the objects in the pre-state.

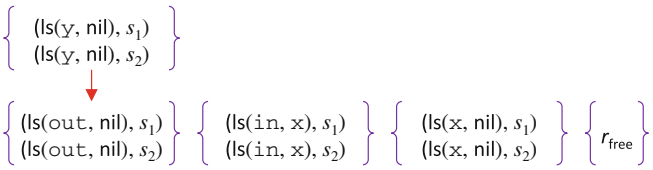

Fig. 7. The condensation graph induced by the evolution edges between s_1 and s_2

Our reasoning depends on depleting regions. To account for this, care must be taken when objects are allocated inside a loop. To account for this, we introduce a special *free region* (not explicitly represented by the symbolic heaps) to represent the (infinite) region of unallocated objects. We merge its region rank with any region rank that may contain loop-allocated objects (lines 2–5). This way, loop-allocated objects never decrease the relation.

By definition (Sect. 5), all concrete transitions preserve the resulting well-founded relation. To show termination, however, we need to show that this relation strictly decreases infinitely often for each cycle. For this purpose, we maintain only evolution must edges that decrease the well-founded relation (lines 6–10).

3.3 Computing Loop Cuts

Let L be a loop and $\tau(L) \subseteq \mathcal{S} \times \mathcal{S}$ be the abstract transition relation computed by the shape analysis for that loop. To show that L terminates, we follow this recipe: (i) find a well-founded relation $s \preceq s'$ over the heap descriptors, which

is preserved by each transition (that is, either the relation stays the same or it strictly decreases); (ii) find a set $Cut(L) \subseteq \tau(L)$ (as large as possible) such that each transition in $Cut(L)$ strictly decreases $\prec$; and (iii) test whether $Cut(L)$ forms a cut. That is whether $\tau(L) \setminus Cut(L)$ does not contain any cycles, in which case all executions represented by $\tau(L)$ must terminate.

The benefit of this approach is that each abstract transition can be tested independently. In our case, we only need to check (lines 8–14) whether the transition includes an edge returned by **EvolutionToWF**.

Contrast this with approaches [5] that synthesize loop-specific rank functions. Those usually choose a priori a set of CFG nodes as loop cutpoints (usually the loop header) and try to establish that transitions leaving these cutpoints decrease the rank. If that fails, e.g., due to the level of abstraction at those points, an alternative set must be searched for. Our approach finds the set of cutpoints at the fine granularity of abstract transitions. Since this set is maximal, all alternative choices have been explored in one shot. This is possible only because we have constructed a well-founded relation that is guaranteed to be preserved by all (concrete and abstract) transitions.

In our example, the evolution edge $((\mathsf{ls}(\mathtt{y}, \mathsf{nil}), s_2), (\mathsf{ls}(\mathtt{out}, \mathtt{y}), s_1))$ decreases the rank of every concrete state represented by s_2 and therefore the abstract transition (s_2, s_1) forms a loop cut.

3.4 Employing Modular Reasoning

To show that a program terminates, we can first infer a sound specification for each loop, e.g., as a Hoare triple, by a program-wide analysis. We can then reason about the termination of each loop separately by: (i) using specifications for its internal loops, and (ii) assuming internal loops terminate. If all loops have been proven to terminate this way, the entire program terminates.

ObjectFlowTermCheck obtains the set of sub-CFGs corresponding to program loops (line 2). Those can be found by linear time loop-reconstruction algorithms based on the dominance. The algorithm then proceeds to produce a loop-specific version of the abstract transition relation and the evolution relation for each loop L. This is achieved by the recursive procedure Summarize (line 3), which works as follows. It computes for each heap descriptor at the loop entry the heap descriptors it reaches at the loop exit. This can be done in linear time by a breadth-first search by following the abstract transition relation. Similarly, the reachability relation can be computed in linear time for the evolution relation. When a heap descriptor stored at the entry of an inner loop L' is encountered, Summarize executes for L' and caches its result—a pair of relations between the entry of L' and its exit. It uses the cached relation of L' to compute the result for L. Finally, Summarize returns for each loop L the pair of relations where inner loops are treated as atomic steps by using their entry-to-exit summarized relations. Finally (lines 12–14), if a loop cannot be determined to terminate, we emit a warning.

Conceptually, the well-founded relations of nested loops yield a lexicographic order. However, summarization lets us avoid reasoning about this lexicographic

order explicitly. Instead, it is constructed incrementally in reverse order to the nesting level of the loops. The resulting verification is loop-modular: The outer loop is verified to terminate under the (proven) assumption that the inner loop terminates. However, loop-modularity comes with a price: When verifying the termination of the outer loop we cannot use must-flow edges of the inner loop, and vice versa.

Using the summary for `Find`, we apply our algorithm to the outer loop of Fig. 1 and establish that it terminates.

The overall running time of our algorithm is linear in the size of the program CFG P, the size of the abstract transition relation τ, and the size of the evolution relation ev.

We note that our reasoning can only be applied to loops whose number of iterations is linearly bounded by the number of objects that exist upon entry (this is because the rank of an object can decrease at most by a number of times equal to the height of the condensation graph, which is a static constant). In practice, this does not seem to be a limitation as the overwhelming majority of programs we have looked at have linear loops and the small number of super-linear loops (such a fixed-point loops) do not seem to be amenable to automation.

4 Partition-Based Shape Analysis

Our algorithm for proving termination is parametric in the underlying shape analysis, which is expected to satisfy three natural properties:

1. The analysis should represent (possibly unbounded) sets of states by *finite* sets of *abstract states*, and provide an over-approximation of the program's transition relation by means of a *finite abstract transition relation* between abstract states.
2. Every abstract state should induce a partitioning of (the unbounded set of) the allocated objects in every state it represents into a finite set of *regions*. Furthermore, the analysis should allow to deduce an *evolution relation* [30] which tracks the flow of objects between regions in every transition.

Abstract Semantic Domains	
$s \in \mathcal{S}$	Abstract states
$\mathrm{S} \in \mathfrak{S} \subseteq 2^{\mathcal{S}}$	Abstract domain
$\rho \in \mathcal{R}$	Region identifiers
$\tau \subseteq \mathcal{S} \times \mathcal{S}$	Abstract transition relation
$\mathsf{ev}, \hat{e} \subseteq (\mathcal{S}, \mathcal{R}) \times (\mathcal{S}, \mathcal{R})$	Evolution relation
Concretization Functions	
$\gamma : \mathfrak{S} \to 2^{\Sigma}$	Concretization function
$\gamma^R : (\mathcal{S} \times \Sigma) \rightharpoonup (\mathcal{R} \to 2^{\mathcal{O}})$	Region concretization function

Fig. 8. Summary of the abstract semantic domains and expected operations

3. Given an abstract transition pertaining to a command that allocates memory, the analysis should be able to determine which regions in the post-state may contain newly allocated objects. Similarly, for a command that deallocates memory, it should be able to determine which regions in the pre-state may contain objects deallocated by the command.

In the rest of the section, we formalize the expected properties of the underlying shape analysis and describe the construction of the *evolution relation*. Figure 8 summarizes the semantic domains and the operations assumed on them.

4.1 Shape Analysis

We assume that the underlying shape analysis uses an abstract domain $\mathfrak{S}$, ranged over by S, whose elements are finite sets of *abstract states*. We denote the (arbitrary but *finite*) set of all possible abstract states by $\mathcal{S}$, and range over it by s. We are not concerned by the way the analysis works, only that it provides a *concretization function* γ which determines which (concrete) states every set $\mathrm{S} \in \mathfrak{S} \subseteq 2^{\mathcal{S}}$ of abstract states represents. We expect that the latter work in a pointwise manner, i.e., $\gamma(\mathrm{S}) = \bigcup_{s \in \mathrm{S}} \gamma(\{s\})$. That is, the set of states represented by S is the union of the states represented by its members. In addition, we expect abstract states to record the value of the program counter, i.e., all the (concrete) states represented by an abstract state s have the same value for the program counter, and all the abstract states in a set $\mathrm{S} \in \mathfrak{S}$ record the same value of the program counter.

Definition 1 (Shape Domain). *A* shape domain $(\mathfrak{S}, \gamma)$ *is comprised of a set of sets of abstract states* $\mathfrak{S} \subseteq 2^{\mathcal{S}}$ *and a concretization function* $\gamma : \mathfrak{S} \to 2^{\Sigma}$ *mapping sets of abstract states to sets of (concrete) states, such that for any* $\mathrm{S} \in \mathfrak{S}$ *the following holds:*

$$\gamma(\mathrm{S}) = \bigcup_{s \in \mathrm{S}} \gamma(\{s\})\,, \textit{ and}$$

$$\forall s_1, s_2, \sigma_1, \sigma_2.\, (\{s_1, s_2\} \subseteq \mathrm{S} \wedge \sigma_1 \in \gamma(\{s_1\}) \wedge \sigma_2 \in \gamma(\{s_2\})) \implies v(\sigma_1) = v(\sigma_2)\,.$$

Recall that the semantics of the program is described using a transition relation between states. We expect that the underlying shape analysis computes an over-approximation of this relation as a relation between abstract states.

Definition 2 (Abstract Transition Relation). *Let* $(\mathfrak{S}, \gamma)$ *be a shape domain. An* abstract transition relation $\tau \subseteq \mathcal{S} \times \mathcal{S}$ *is a binary relation over abstract states.* T conservatively represents *a concrete transition relation* T *if*

$$T \subseteq \{(\sigma_1, \sigma_2) \mid (s_1, s_2) \in \tau, \sigma_1 \in \gamma(\{s_1\}), \sigma_2 \in \gamma(\{s_2\})\}\,.$$

4.2 Partition-Based Shape Analysis

Partition-based shape analyses *utilize abstract states which induce* a finite partitioning of the (the unbounded set of) allocated objects in every state they represent into a finite set of regions. The regions induced by an abstract state often have a semantic meaning, e.g., different regions may correspond to different data structures. However, for our purpose, such a meaning is of no importance. We only care that for a given program, the number of regions induced by any abstract state is finite. We abstract away from the particular details of this kind of partition-based abstractions using the notion of a *region concretization function* γ^R. Formally, we assume to be given a finite set of *regions identifiers* $\mathcal{R}$, ranged over by the metavariable ρ, and use them to identify subsets of the dynamically allocated objects in a state. γ^R maps an abstract state s and a concrete state σ represented by it to a function from region identifiers to (the possibly empty) the set of objects they represent. For technical reasons, we assume that $\mathcal{R}$ includes a specially designated region identifier ρ_{free}, which represents the free (unallocated) memory locations.

Definition 3 (Partition-Based Shape Domains). *A* partition-based shape domain $\mathcal{A} = (\mathfrak{S}, \gamma, \gamma^R)$ *is comprised of a shape domain* $(\mathfrak{S}, \gamma)$ *and a* region concretization function $\gamma^R : (\mathfrak{S} \times \Sigma) \to (\mathcal{R} \to 2^{\mathcal{O}})$, *such that for every abstract state* $s \in \mathfrak{S}$ *and any concrete state* $\sigma \in \gamma(s)$, *the following holds:*

1. *The free region represents the unallocated objects:* $\gamma^R(s, \sigma, \rho_{free}) = F(\sigma)$, *and*
2. *Every object is represented by exactly one region:*

$$\forall o \in \mathcal{O}.\, \exists! \rho \in \mathcal{R}.\, o \in \gamma^R(s, \sigma, \rho).$$

The definition formalizes our intention that an abstract state s induces a partitioning of the objects of every concrete state σ it represents. For example, if a heap containing a single linked list, then all the objects comprising the list may be in one region or in two regions, separating, e.g., the head of the list from its tail. However, it is not possible for an object to appear in two different regions. The partitioning of allocated objects is parametric in the chosen shape domain. By introducing the free region, we ensure that all the objects, and not only the allocated ones, are associated with a region. In our experiments, a region is comprised of a connected set of objects. However, for our purposes, a region may contain an arbitrary set of objects.

The success of a shape analysis algorithm to prove interesting properties often depends on its ability to record properties which are common to all the objects that reside in the same region. For example, in TVLA [26], it is often crucially important to track the reachability relation between heap allocated object, e.g., that all the elements in the list pointed-to by a variable `out` can be reached from it by following a finite number of `n`-pointer fields. To be able to record such information, the partitioning induced by abstract states is dynamic—changes to the state might lead objects to flow from one region to another.

Perhaps the most important property that we require from the underlying shape analysis is that it should be possible to deduce an *evolution relation* [30], which tracks the flow of objects between the regions of abstract states related by an abstract transition. We have shown an example of how to compute this relation for separation logic. TVLA uses a similar approach.

Definition 4 (May Evolve Relation). *Let* $(\mathfrak{S}, \gamma, \gamma^R)$ *be a partition-shape domain. A* may evolution relation $\mathsf{ev} \subseteq (\mathcal{S}, \mathcal{R}) \times (\mathcal{S}, \mathcal{R})$ *is a binary relation over pairs of abstract states and region identifiers. We denote the* abstract transition relation underlying ev *by*

$$\tau(\mathsf{ev}) \stackrel{def}{=} \{(s_1, s_2) \mid ((s_1, \rho_1), (s_2, \rho_2)) \in \mathsf{ev}\} \ .$$

The evolution relation ev *conservatively represents a concrete transition relation* T *if*

1. T *is conservatively represented by* $\tau(\mathsf{ev})$.
2. *There is a transition from* (s_1, ρ_1) *to* (s_2, ρ_2) *if an object in a state represented by* s_1 *in region* ρ_1 flows *to region* ρ_2 *in a state represented by* s_2*:*

$$\forall \sigma_1, \sigma_2, s_1, s_2, \rho_1, \rho_2. \big((\sigma_1, \sigma_2) \in T \wedge \sigma_1 \in \gamma(\{s_1\}) \wedge \sigma_2 \in \gamma(\{s_2\}) \wedge \gamma^R(s_1, \sigma_1)(\rho_1) \cap \gamma^R(s_2, \sigma_2)(\rho_2) \neq \emptyset\big) \implies ((s_1, \rho_1), (s_2, \rho_2)) \in \mathsf{ev}.$$

The evolution relation records the way objects *may* flow between regions. The last requirement we place is the ability to (conservatively) determine that some of the evolution transitions correspond to *must* flow information, i.e., that an object actually moves from one region to another.

Definition 5 (Must Evolve Relation). *Let* $(\mathfrak{S}, \gamma, \gamma^R)$ *be a partition-based shape domain,* $\hat{e}$ *an* evolution relation*, and* T *be a concrete transition relation.* $\hat{e}$ conservatively under-approximates the flow of objects in T *if*

$$\forall \sigma_1, \sigma_2, s_1, s_2, \rho_1, \rho_2. \big(((s_1, \rho_1), (s_2, \rho_2)) \in \hat{e} \wedge (\sigma_1, \sigma_2) \in T \wedge \sigma_1 \in \gamma(\{s_1\}) \wedge \sigma_2 \in \gamma(\{s_2\})\big) \implies \gamma^R(s_1, \sigma_1)(\rho_1) \cap \gamma^R(s_2, \sigma_2)(\rho_2) \neq \emptyset) \ .$$

5 Establishing Well-Founded Relations

Our termination algorithm ensures that a program P terminates by: (i) establishing a well-founded relation over the pre- and post- states of every transition in P's concrete transition relations T, and (ii) attempting to prove the existence of a cutting set of transitions that strictly decrease the relation.

We now describe how to construct this well-founded relation from the abstract transition relation τ and the evolution relation ev, and how to detect decreasing transitions via the must-evolve relation $\hat{e}$, computed using a partition-based shape domain $\mathcal{A} = (\mathfrak{S}, \gamma, \gamma^R)$.

Establishing a Well-Founded Relation Over Regions. Given the (finite) evolution relation ev, we construct the graph induced by its strongly connected components. Formally, every node in this graph corresponds to an equivalence class of pairs $r = (s, \rho)$ of abstract states and region identifiers. We refer to such a pair as an *abstract region* (region for short). Recall that our algorithm verifies the termination of every loop in the program separately. It does so by tracking the flow of objects which were allocated at the entry to the loop. To prevent us from considering objects which are allocated inside the loop in our termination argument, we collapse together all the nodes that contain a (s, ρ) pair which is reachable in the evolution relation from the free region of some input abstract state into a single ρ_{free} region. We refer to the resulting graph as the *condensation graph* and to its nodes as *region ranks*, and denote the region rank corresponding to a region r by $[r]$. We say that a region r is *less than or equal to* a region r', denoted $r \preceq r'$, if $[r']$ is reachable from $[r]$.

Lemma 1. *The relation $r \preceq r'$ is well-founded.*

Establishing a Well-Founded Relation Across Loop Iterations. We define a relation $\preceq$ between states that occur in T by lifting the relation between region rank to objects in a pointwise manner. Note that we do not care about relating states that do not occur in T and recall that a state encodes the value of the program location. Thus, it is sufficient to only define a relation between pairs of states $(\sigma_1, \sigma_2) \in T$ that appear as transitions in T and then take its transitive closure.

Let s be an abstract state and $\sigma \in \gamma(\{s\})$ a state represented by s. We define the *rank map* of σ according to s, denoted by $rank_{s,\sigma}$ to be a mapping from objects to region ranks according to the partitioning induced by s:

$$rank_{s,\sigma}(o) = (s, \rho), \text{ where } o \in \gamma^R(s, \sigma, \rho)\,.$$

Let $(\sigma_1, \sigma_2) \in T$ be a concrete transition of P. We say that an abstract transition (s_1, s_2) *represents* (σ_1, σ_2) if $\sigma_1 \in \gamma(\sigma_1)$ and $\sigma_1 \in \gamma(\sigma_1)$. We say that σ_1 is *less than or equal* to state σ_2, denoted by $\sigma_1 \preceq_1 \sigma_2$, if there exists an abstract transition $(s_1, s_2) \in \tau$ such that for every $o \in \mathcal{O}$, it holds that $rank_{s_1,\sigma_1}(o) \leq rank_{s_2,\sigma_2}(o)$. We write $\sigma_1 \prec_1 \sigma_2$ if $\sigma_1 \preceq_1 \sigma_2$ and $\sigma_2 \not\preceq \sigma_1$. We define the binary relation $\cdot \preceq \cdot$ as the reflexive transitive closure of the *smaller or equal* relation $\cdot \preceq_1 \cdot$ induced by T.

Lemma 2. *The relation $\sigma_1 \preceq \sigma_2$ is well-founded, and if $(\sigma_1, \sigma_2) \in T$ then $\sigma_1 \preceq \sigma_2$.*

Theorem 1. *Let T^+ denote the (irreflexive) transitive closure of T. If the loop contains a cutting set of transitions, then the relation defined as*

$$\sigma_1 \ll \sigma_2 \stackrel{def}{=} \{\sigma_1 \preceq \sigma_2 \mid (\sigma_1, \sigma_2) \in T^+ \wedge v(\sigma_1) = v(\sigma_2)\}$$

(that is, σ_2 is the result of executing at least one loop iteration and reaching the same program location) is well-founded.

6 Interprocedural Analysis

We now briefly discuss the extension of our approach to verify termination of recursive heap-manipulating programs. We forbid mutual-recursion and assume procedures have local variables, but no input parameters nor return values[5]. Syntactically, every procedure call is represented by a *call node* and a *return node* in the program's CFG. Every call node of p has a control-flow edge associated with a *call* command connecting it to the entry node of p's sub-CFG, and the exit node of the latter is connected to each of p's return nodes via a control-flow edge associated with a *return* command. The operational semantics is extended to handle procedures via reduction: we treat the call stack as a heap-allocated list. This allows treating recursive procedures as loop commands that explicitly manipulate the call stack. This kind of reduction has been employed successfully in previous shape analysis algorithms to verify safety properties [24,25]. As far as we are aware, we are the first to employ it to verify termination.

A disadvantage of this approach is that it does not allow us to take advantage of the modular structure of the program— the underlying shape analysis becomes a whole program analysis and has to cope with an additional heap-allocated data structure. Our approach for verifying termination is, however, loop-modular, and thus can still treat procedures in a modular way.

7 Implementation and Experimental Evaluation

We implemented our algorithm on top of the existing shape analysis of TVLA, a parametric framework for shape analysis based on 3-Valued-Logic [17,26], and used it to verify termination for the suite of benchmarks listed in Table 1. The experiments were performed on a machine with a 2.5 GHz *i*7 Intel processor and 16 GB memory. The **Time** column measures the overall time of the shape analysis and the termination analysis combined. The **overhead** column lists the fraction of the time of the analysis spent to prove termination. As expected, the termination analysis is rather efficient.

We extended our programming language to handle bounded parallelism by allowing the program's CFG to have multiple pairs of entry and exit nodes. Every pair corresponds to a thread: The entry node corresponds to the program location of the thread when the program starts. The program terminates when every thread reaches its exit nodes. We assume that the sub-CFG pertaining to different threads are not connected. We record in every state the program location of every thread, and define the operational semantics to be the interleaving of primitive commands executed by different threads.

The suite of benchmarks consists of a variety of programs, concurrent and single-threaded, both recursive and iterative, flat and deeply nested loops. We handle several classic sorting algorithms, trees and graphs algorithms. We implemented and verified 16 out of the 21 benchmarks used to evaluate MUTANT [5],

[5] Mutually recursive procedures can be handled by merging them into a single recursive procedure, and parameter transfer by using specially-designated global variables.

which we had access to. One of the 16 benchmarks has a termination bug in which a lasso is created in a linked list, preventing traversing loop from ever reaching back to the head, and thus making it iterate forever. Our analysis correctly warned against non-termination. Some other interesting examples which were successfully verified for termination include Deutsch-Schorr-Waite tree-traversal algorithm [18] which involves double-reversal of links, the phases of a Mark-and-sweep garbage collection algorithm, the Even-Odd-marking [16] program which marks the elements of a singly-linked list as odd or even regarding their distance from the *end* of the list. We successfully prove termination of some concurrent programs including Treiber stack and CAS-based queue. A faulty implementation of the Treiber stack was correctly warned for non-termination.

Table 1. Benchmarks, analysis times, relative overhead over the original shape analysis times, and loop-nesting depth

Benchmark	Time (sec.)	Overhead	ND
a. Shallow programs manipulating acyclic lists			
Search, Insert, Reverse, Delete-all, Merge-sorted	≤0.7	≤0.1 %	1
b. Deeply nested programs manipulating acyclic lists			
Bubble-sort	5.1	14 %	2
Insertion-sort	0.9	1 %	2
Insertion-sort with malloc	1.1	2.7 %	2
Odd-even-marking	0.2	5.1 %	2
Nested-loop-depth-4	4.4	4.5 %	4
c. Shallow programs manipulating cyclic lists			
Mutant 1–16	0.113	0.1 %	1
Search, Insert	≤0.1	≤0.1 %	1
d. Recursive programs manipulating acyclic lists			
Search, Insert, Append, Reverse, Delete	≤0.3	≤0.3 %	1
Merge	38.9	0.5 %	1
e. Programs manipulating binary trees			
Insert-to-sorted-tree	0.9	1 %	1
Delete-from-sorted-tree	42.1	6 %	1
Deutsch-Schorr-Waite	14.5	4.8 %	1
f. Programs manipulating graphs			
Mark-and-sweep	0.8	1.25 %	1
g. Concurrent programs (lock-freedom)			
CAS Merge two lists (2 threads)	9.21	4 %	1
Treiber stack (2 threads)	1.4	2.5 %	1
Treiber stack (3 threads)	21.2	13 %	1
Faulty Treiber stack (2 threads)	1.6	1.25 %	1

8 Related Work

In this section, we discuss and compare our work with other approaches for verifying termination of *heap-manipulating programs*. This body of work can broadly be classified according to the following dimensions: (i) lattice of heap invariants and supported data structures, and (ii) type of "heap measures" used for reasoning about termination.

In terms of supported program features, we are the only ones to support side-effecting recursive procedures. However, we completely abstract away integers.

Approaches Based on Inductive Heap Invariants. Approaches based on separation logic [4,5,8,9,19] and tree automata [16] are restricted to programs operating on inductively-defined data structures, preventing them from handling programs operating on graphs of unbounded treewidth such as arbitrary graphs. Our algorithm is parametric in the underlying shape analysis and can therefore be used both with separation logic and 3-valued shape analysis. Our implementation uses 3-valued shape analysis as the underlying domain to prove termination of programs manipulating both inductively-defined data structures as well as arbitrary graphs.

Both Berdine et al. [4,5] and Brotherson et al. [8,9] use the number of unfoldings of inductively-defined definitions as a basis for their reasoning. Berdine et al. [4,5] modify a separation logic-based analysis to track changes in the number of unfoldings of inductive predicates and produce a numeric program whose termination implies the termination of the original program. The termination of the numeric program can be checked by termination proving tools for numeric programs. Brotherson et al. [8] develop a proof framework for cyclic proofs where termination is done by directly relating (cyclic) inference trees with inductive predicates. Brotherson and Gorogiannis [9] introduce a method for synthesizing inductive predicates, allowing them to automatically adapt to the data structures appearing in a program.

Magill et al. [19] develop a flexible analysis that simultaneously tracks both shape invariants and related numeric measures such as the sizes of lists and trees. They generate a numeric program whose termination proof implies termination of the original program. Proving termination of the numeric program is done by using tools dedicated for that purpose (ARMC-LIVE).

Tracking numeric properties is both expensive and requires making modifications to the shape analysis. Proving termination of numeric programs is undecidable in general and expensive in practice. Our algorithm avoids these problems by directly processing the result of the shape analysis in linear time. We note that computing the evolution relation is relatively simple and efficient.

Haberhmehl et al. [16] use shape analysis based on tree automata to generate a counter automaton that simulates the original program. They use counterexample-guided abstraction refinement to obtain a high degree of automation (and to prove relative completeness for a subclass of programs). They track measures suitable for termination of tree manipulation such as tree size, height, and distance of pointers from the root. However, they only handle tree rotations and do not handle dynamic allocation.

Approaches Based on $\mathsf{FO}^{\mathsf{TC}}$ Shape Invariants. A family of analyses approximate the shape of the heap via predicates expressible in first-order logic with transitive closure ($\mathsf{FO}^{\mathsf{TC}}$). These enable to indirectly capture some recursive data structures as well as reason about arbitrary data structures in a sound manner.

Podelski et al. [22] present an analysis for inferring necessary preconditions for termination of list-manipulating programs. Their analysis starts with a coarse heap abstraction and proves termination by translation to numeric programs. If an abstract counterexample is found, the heap abstraction is refined. The heap abstraction is converted to a numeric program which tracks the lengths of heap paths between two pointer variables.

Gulwani et al. [14] develop an abstract interpretation framework for tracking numerical relations betweens the sizes of heap regions, and use it to manually obtain ranking functions for bubble-sort and the mark and sweep garbage collector benchmark. One disadvantage of combining shape and size information (other than the expensive cost incurred by tracking numeric properties and performing partial reduction between the shape domain and numeric domain) is the unpredictability brought by widening operations that are usually necessary for the numeric sub-domain. Additionally, the termination of some programs, e.g., Odd-even-marking [16], does not depend on the sizes of heap regions.

Albert et al. [2] and Fausto et al. [27] develop heap abstractions based on sharing, cyclicity, and aliasing and use access path lengths to prove termination. Technically, they translate Java byte code to constraint logic programs (CLP) where the actual termination reasoning is done. Giesl et al. [21] define a shape analysis based on sharing, aliasing, and reachability patterns. This allows them to analyze programs manipulating integers and tree-shaped data structures and prove their termination by reduction to term rewriting systems (TRS). The termination measures available by TRS go beyond region sizes (e.g., flattening a tree). They later extend their technique to handle certain cyclic data structure patterns [6] and recursive procedures without side-effects [7]. Their analysis has been implemented in the AProVE tool [13], which analyzes both Java and C programs.

The main limitation of the last three approaches is the rigidity of the heap abstraction—adapting the abstraction to precisely handle new data structures and new data structure manipulation patterns requires reworking the analysis. In particular, proving termination for programs manipulating arbitrary graphs seems to be out of reach. In contrast, our algorithm is not tied down to any specific shape analysis technique, allowing us to take advantage of advances in the field of shape analysis, e.g., compositional analysis [10], and automatic synthesis of inductive predicates [9,15]. Our implementation on top of TVLA allows us to quickly adapt to new data structures by refining the abstraction with new predicates (soundness is automatically guaranteed). Finally, our treatment of recursion allows us to quite precisely track intricate properties between the heap and the call stack.

Efficiency. As far as we know, our algorithm is the only one that operates in linear time with respect to an underlying shape analysis. This is due to two factors—the properties of the well-founded relation our analysis constructs and our modular treatment of loops and recursive procedures.

Acknowledgments. We thank the anonymous reviewers for their detailed comments. We thank Josh Berdine and Amir Ben-Amram for useful discussions.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles,Techniques and Tools. Addison-Wesley, Reading (1988)
2. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination analysis of Java bytecode. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 2–18. Springer, Heidelberg (2008)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
4. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.W.: Variance analyses from invariance analyses. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 211–224 (2007)
5. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.W.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
6. Brockschmidt, M., Musiol, R., Otto, C., Giesl, J.: Automated termination proofs for Java programs with cyclic data. In: International Conference on Computer Aided Verification, pp. 105–122 (2012)
7. Brockschmidt, M., Otto, C., Giesl, J.: Modular termination proofs of recursive Java bytecode programs by term rewriting. In: International Conference on Rewriting Techniques and Applications, pp. 155–170 (2011)
8. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 101–112 (2008)
9. Brotherston, J., Gorogiannis, N.: Cyclic abduction of inductively defined safety and termination preconditions. In: Müller-Olm, M., Seidl, H. (eds.) Static Analysis. LNCS, vol. 8723, pp. 68–84. Springer, Heidelberg (2014)
10. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26 (2011)
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
12. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
13. Gies, J., et al.: Proving termination of programs automatically with AProVE. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 184–191. Springer, Heidelberg (2014)
14. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 239–251 (2009)

15. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 256–265 (2007)
16. Habermehl, P., Iosif, R., Rogalewicz, A., Vojnar, T.: Proving termination of tree manipulating programs. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 145–161. Springer, Heidelberg (2007)
17. Lev-Ami, T., Sagiv, M.: TVLA: a framework for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–301. Springer, Berlin (2000)
18. Loginov, A., Reps, T., Sagiv, M.: Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 261–279. Springer, Heidelberg (2006)
19. Magill, S., Tsai, M., Lee, P., Tsay, Y.: Automatic numeric abstractions for heap-manipulating programs. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 211–222 (2010)
20. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco (1997)
21. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of Java bytecode by term rewriting. In: International Conference on Rewriting Techniques and Applications, pp. 259–276 (2010)
22. Podelski, A., Rybalchenko, A., Wies, T.: Heap assumptions on demand. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 314–327. Springer, Heidelberg (2008)
23. Rinetzky, N., Ramalingam, G., Sagiv, M., Yahav, E.: On the complexity of partially-flow-sensitive alias analysis. ACM Trans. Program. Lang. Syst. **30**(3), 13:1–13:28 (2008)
24. Rinetzky, N., Sagiv, M.: Interprocedural shape analysis for recursive programs. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 133–149. Springer, Heidelberg (2001)
25. Rival, X., Chang, B.-Y.E.: Calling context abstraction with shapes. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 173–186 (2011)
26. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002)
27. Spoto, F., Mesnard, F., Payet, E.: A termination analyzer for Java bytecode based on path-length. ACM Trans. Program. Lang. Syst. **32**(3), 8:1–8:70 (2010)
28. Toubhans, A., Chang, B.-Y.E., Rival, X.: Reduced product combination of abstract domains for shapes. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 375–395. Springer, Heidelberg (2013)
29. Toubhans, A., Chang, B.-Y.E., Rival, X.: An abstract domain combinator for separately conjoining memory abstractions. In: Müller-Olm, M., Seidl, H. (eds.) Static Analysis. LNCS, vol. 8723, pp. 285–301. Springer, Heidelberg (2014)
30. Yahav, E., Reps, T.W., Sagiv, S., Wilhelm, R.: Verifying temporal heap properties specified via evolution logic. In: European Symposium on Programming, pp. 204–222 (2003)