From Shapes to Amortized Complexity[⋆]

Tomáš Fiedor[1], Lukáš Holík[1], Adam Rogalewicz[1],
Moritz Sinn[3], Tomáš Vojnar[1], and Florian Zuleger[2]

[1] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic
[2] TU Wien, Austria [3] St. Pölten University of Applied Sciences, Austria

Abstract. We propose a new method for the automated resource bound analysis of programs manipulating dynamic data structures built on top of an underlying shape and resource bound analysis. Our approach first constructs an integer abstraction for the input program using information gathered by a shape analyser; then a resource bound analyzer is run on the resulting integer program. The integer abstraction is based on shape norms — numerical measures on dynamic data structures (e.g., the length of a linked list). In comparison to related approaches, we consider a larger class of shape norms which we derive by a lightweight program analysis. The analysis identifies paths through the involved dynamic data structures, and filters the norms which are unlikely to be useful for the later bound analysis. We present a calculus for deriving the numeric changes of the shape norms, thereby generating the integer program. Our calculus encapsulates the minimal information which is required from the shape analysis.
We have implemented our approach on top of the Forester shape analyser and evaluated it on a number of programs manipulating various list and tree structures using the Loopus tool as the underlying bounds analyser. We report on programs with complex data structures and/or using complex algorithms that could not be analysed in a fully automated and precise way before.

1 Introduction

Automated resource bound analysis is an active field of research (for an overview we refer the reader e.g. to [1] and the references therein), which aims at developing tools and analysis techniques that allow developers to understand the performance of their code and to verify the resource consumption of their programs in case that bounding the resource consumption is a crucial correctness requirement.

The research of this paper is partly motivated by the experimental evaluation of our resource bound analysis tool Loopus [1], where we analysed a large number of C programs. Loopus computes resource bounds based on the updates to integer variables, however, has only a limited support for pointers. One of the results of our experiments was that missing pointer/shape analysis is the most frequent reason for the failure of Loopus to compute a resource bound. In [1] we report that we obtained bounds for 753 of the 1659 functions in our benchmark (45%), and that by a simple (but unsound) shape analysis we were able to increase the number of computed bounds to 1185 (71%).

In this paper, we study the automated resource bound analysis of heap-manipulating programs. We focus on the analysis of data structures as they can be found in systems code such as operating system kernels, compilers, or embedded systems. Performance is

[⋆] Supported by the Czech Science Foundation (project 17-12465S), the BUT FIT project FIT-S-17-4014, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602) and the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF).

a major concern in systems code and has led to the use of customised data structures and advanced data structures such as, e.g. red-black trees, priority heaps or lock-free linked lists. These data structures are complex and prone to introducing errors. Thus, automated tool support promises to increase the reliability of systems and can lead to a better user experience. Resource bound analysis of programs with data structures has been addressed only by a few publications [2,3,4,5,6]. In this paper we improve along several dimensions on these earlier results allowing the automated resource bounds analysis of heap-manipulating programs that cannot be handled by existing approaches.

Our approach. Our analysis works in three steps. We first run a shape analysis and annotate the program with the shape invariants. Then based on numeric information about the heap using the results from the shape analysis we create a corresponding integer abstraction of the program. Finally, we perform resource bound analysis purely on the integer program.

The numeric abstraction is based on *shape norms*, which are numerical measures on dynamic data structures (e.g. the length of a linked list). Our first contribution is the definition of a class of shape norms that express the longest distance between two points of interest in a shape graph and are defined in terms of basic concepts from graph theory. Our norms are parameterized by the program under analysis and are extracted in a pre-analysis (with a possibility of extending the initial set during the subsequent analysis); the extracted norms then correspond to selector paths found in the program.

Our second contribution is a calculus for our shape norms that allows to derive how the norms change along a program statement, i.e. if the norm is incremented resp. decremented or reset to some other expression. The calculus consists of two kinds of rules. (1) Rules that allow to directly infer the change of a norm and do not need to take additional information into account. (2) Rules that rely on the preceding shape analysis; the shape information is used there for (a) dealing with pointer aliasing and (b) deriving an upper bound on the value of a norm from the result of the shape analysis (if possible). We point out that the rules (2) encapsulate the points of the analysis where information about the shape is needed, and thus describe the minimal requirements on the preceding shape analysis. We believe that this separation of concern also allows the use of other shape analysers.

When creating the integer abstraction we could use all shape norms that we extracted from the program. However, we have an additional pre-analysis phase that eliminates norms that are not likely to be useful for the later bound analysis. This reduction of norms has the benefit that it keeps the number of variables in the integer abstraction small. The smaller number of additional variables increases the readability of the resulting integer abstraction and simplifies the developing and debugging of subsequent analyses. Additionally, the number of extracted norms can be quadratic in the size of the program; hence adding quadratically many variables can be prohibitively expensive and the pre-analysis is therefore crucial to the success of the later bound analysis.

Finally, we perform resource bound analysis on the created integer abstraction. This design decision has two advantages. First, we can leverage the existing research on bound analysis for integer programs and do not have to develop a new bound analysis. Second, being able to analyse not only shape but also integer changes has the advantage that we can analyse programs which mix integer iterations with data structure iterations; we illustrate this point by analysing the flagship example of [4], which combines iteration over data-structures and integer loops in an intricate way.

Implementation and Experiments. The generation of the integer program is implemented on top of the shape analyser Forester [7]. We use the Loopus tool [8,1] for inferring the computational complexity of the obtained integer abstractions. Our experimental evaluation demonstrates that the combination of these tools can yield a powerful analysis. We report on results for complex heap manipulating programs that could not be handled by previous approaches as witnessed by experimental evaluation against the tools AProVE [9] and COSTA [5]. We remark that our implementation leverages the strengths of both Forester and Loopus. We inherit the capabilities of Forester to analyse complex data structures, and report on analysis results for double-linked lists, trees, 2-level skip-lists, etc. Moreover, our analysis of shape norms is precise enough to leverage the capabilities of Loopus for amortized complexity analysis — we report on the amortized analysis of the flagship example of [4], whose correct linear bound has to the best of our knowledge never been inferred fully automatically.

Related work. The majority of the related approaches derive an integer program from an input heap-manipulating program followed by a dedicated analysis (e.g. termination or resource bounds) for arithmetic programs. The transformation has to be done conservatively, i.e. the derived integer program needs to simulate the original heap-manipulating program such that the results for the integer program hold for the original program. The related approaches differ in the considered numeric measures on the heap, the data structures that can be analysed and the degree of automation.

Several approaches have targeted restricted classes of data structures such as singly linked lists [10,11,12,13,14] or trees [15,16]. It is unclear how to generalise these results to composed or more complex data structures which require different numeric measures or combinations thereof.

A notable precursor to our work is the framework of [17] implemented in the THOR tool [18], which describes a general method for deriving integer abstractions of data structures. The automation of THOR, however, relies on the user for providing the shape predicates of interest (the implementation only comes with list predicates, further predicates have to be added by the user). Further, we found during initial investigations that THOR needlessly tracks shape sizes not required for a later termination or bounds analysis, which can quickly bloat the program under analysis.

A general abstract interpretation-style framework for combining shape and numerical abstract domains is described in [2]. The paper focuses on tracking of partition sizes, i.e. the only considered norm is the number of elements in a data structure. Our framework is orthogonal: we can express different norms, e.g. the height of a tree, which cannot be expressed in [2]; on the other hand, we use numeric information only in the second stage of the analysis which can be less precise than the reduced product construction of [2].

An automated approach to amortized complexity analysis of object-oriented heap-manipulating programs is discussed in [3]. The approach is based on the idea of associating a potential to (refinements of) data structure classes. Typing annotations allow to derive a constraint system which is then solved in order to obtain valid potential annotations. The implementation is currently limited to linear resource bounds and appears to be restricted to list-like data structures.

The idea of using potentials for the analysis of data structures is also investigated in [4]. The author extends separation logic with resource annotations exploiting the idea of separation in order to associate resource units to every memory cell, resulting in an elegant Hoare-logic for resource analysis. The suggested approach is currently only semi-automated requiring the user to provide shape predicates and loop invariants manually.

In [5], the authors propose an automated resource analysis for Java programs, implemented in the COSTA tool. Their technique is based on abstracting arrays into their sizes and linked structures into the length of the longest chain of pointers terminated by `NULL`, followed by the construction and solving of a system of recurrence equations. However, cyclic lists and more complicated data structures such as DLLs, are, to the best of our knowledge, out of the capabilities of this technique as they require more general numeric size measures.

A recent paper investigates the automated resource analysis for Java programs and reports on its implementation in the AProVE tool [6], based on first translating a program to an integer transition system, and then using a bounds analyser to infer the complexity. The technique makes use of a single size-measure which is the number of nodes reachable from the heap node of interest together with the sum of all reachable integer cells. This norm is orthogonal to the norms considered in this paper. On the other hand, the norm of [6] does often not correspond to the size of interest: for example, in case of an iteration over the top-level list of a list of lists, the relevant norm is the length of the top-level list and not the number of total data structure elements; similarly in case of a search in a sorted tree: the relevant size measure is the height of a tree and not the number of elements. Moreover, it is unclear how the norm of [6] deals with cyclic data structures; while the number of reachable elements is well-defined, it is unclear if resp. how the norm changes when a pointer is advanced because the number of reachable nodes does not change.

Contributions. We summarise our contributions in this paper:

1. In comparison with related approaches we consider a larger class of shape norms.
2. We develop a calculus for deriving the numeric changes of the shape norms. The rules of our calculus precisely identify the information that is needed from a shape analyser. We believe that this definition of minimal shape information will be useful for the development of future resource bound analysis tools.
3. Our norms are not fixed in advance but mined from the program: We define a pre-analysis that reduces the number of considered norms. To our experience, this reduction is very useful during development of the resource analysis and for reporting the integer abstraction to the user.
4. We demonstrate in an experimental validation that we obtain a powerful analysis and report on complex data structure iterations that could not be analysed before.

2 A Brief Overview of the Proposed Approach

We are interested in deriving the computational complexity of Algorithm 1, where we understand the computational complexity as the total number of loop iterations. Our analysis infers an upper bound of the computational complexity by inferring a bound for the number of iterations of each loop and summing these loop bounds. The computation of other resource bounds can often be reduced to the computation of loop bounds in a similar way (we refer the reader to the discussion in [1] for more details).

```
1 x = list;
2 y = x;
3 while $x \neq$ NULL do
4     x = x.next;
5     if (*) then
6         while $y \neq x$ do
7             y = y.next;
```

Algorithm 1: A running example with computational complexity $2 \cdot list\langle\text{next}^*\rangle\texttt{NULL} = \mathcal{O}(n)$.
* denotes nondeterministic choice.

```
1 $x\langle\text{next}^*\rangle\texttt{NULL} = list\langle\text{next}^*\rangle\texttt{NULL}$;
2 $y\langle\text{next}^*\rangle x = 0$;
3 while $x\langle next^*\rangle\texttt{NULL} \neq 0$ do
4     $x\langle\text{next}^*\rangle\texttt{NULL}{-}{-}$;
      $y\langle\text{next}^*\rangle x{+}{+}$;
5     if (*) then
6         while $y\langle next^*\rangle x \neq 0$ do
7             $y\langle\text{next}^*\rangle x{-}{-}$;
```

Algorithm 2: A pure integer program corresponding to Algorithm 1

We thus limit the discussion in this paper to complexity bounds. A bound here is a symbolic expression in terms of the program variables. Our implementation computes complexity bounds with concrete constants—e.g., for Algorithm 1, we infer the bound $2 \cdot list\langle\text{next}^*\rangle\texttt{NULL}$ where $list\langle\text{next}^*\rangle\texttt{NULL}$ is a shape norm (we discuss shape norms in more detail below). For ease of understanding and for comparison with related approaches, we also state asymptotic complexity bounds, which we obtain by replacing all shape norms with n, e.g., $2 \cdot list\langle\text{next}^*\rangle\texttt{NULL} = \mathcal{O}(n)$.

We now present a brief overview of our approach on Algorithm 1, a simplified version of list partitioning. The outer loop at *line 3* iterates over the single linked list referenced by the variable `list`, at *line 5* the loop non-deterministically processes the partition of the list accumulated between variables `y` and `x`. We remark that deriving bounds for Algorithm 1 is challenging because (i) we have to reason not only about the lists x and y but also about the distance between these two pointers, and (ii) to infer the precise bound our reasoning must track the distance between x and y precisely rather than overapproximating it by the worst-case (which would lead to a quadratic bound). We sketch the main steps of our analysis below.

Shape analysis. The underlying shape analysis is run first. A successful run annotates each locations of the control-flow graph with a set of shape invariants and provides a guarantee that no safety violations, like e.g., a `NULL` pointer dereference, can occur during the program run. The shape invariants are needed to generate an integer abstraction of the program. Moreover, they can be leveraged to increase the precision of the subsequent bounds analysis, e.g., when the length of a path between variables y and x through the `next` selector is always constant. If the shape analysis fails, we end the analysis, as we will lack the necessary information to generate an integer program.

Deriving the candidate norms. We infer suitable candidate norms from the program control-flow as follows: (1) The loop header conditions define the set of candidate norms of the form $source\langle\text{re}^*\rangle destination$, where $source$ is a pointer variable, destination a distinct point (such as `NULL` or another pointer variable) and re is a placeholder for a regular expression over pointer selectors, which is filled in the next step. E.g., we can drive norm $x\langle\text{re}^*\rangle\texttt{NULL}$ from the condition $x \neq \texttt{NULL}$ in line 3 of Algorithm 1. We then derive the set of possible selector paths that may be traversed during the program run by a lightweight program analysis, in order to build the regular expression re: For our example we infer that at each iteration of the outer cycle variable `x` is moved by the selector `next`. We thus build the regular expression $next^*$ and obtain the complete candidate norm $x\langle\text{next}^*\rangle\texttt{NULL}$.

Analogically, we infer two candidate norms $x\langle\text{next}^*\rangle y$ and $y\langle\text{next}^*\rangle x$ for the inner loop at *line 6*. We thus obtain $\mathcal{N}_c = \{x\langle\text{next}^*\rangle\texttt{NULL}, x\langle\text{next}^*\rangle y, y\langle\text{next}^*\rangle x\}$ as the inital set of candidate norms to be tracked. Note that this set is only an initial set and can be further extended during the generation of the integer program: E.g., when one of the norms $\mu_1 \in \mathcal{N}_c$ is reset to a norm $\mu_2 \notin \mathcal{N}_c$, μ_2 will be added to $\mathcal{N}_c$ and tracked.

Arithmetic program generation. For simplicity we consider only the norms $\mu_1 = x\langle\text{next}^*\rangle\texttt{NULL}$ and $\mu_2 = y\langle\text{next}^*\rangle x$ in our discussion, because these are sufficient to obtain a precise bound. We first translate the pointer conditions to corresponding integer conditions: The condition $x \neq \texttt{NULL}$ in line 3 is translated to the condition $\mu_1 \neq 0$ over norms. Analogically, the condition $x \neq y$ is translated to $\mu_2 \neq 0$.

We then derive norm updates (increases, decreases, resets) for each pointer instruction: *line 1* resets the norm μ_1 to $\mu_3 = list\langle\text{next}^*\rangle\texttt{NULL}$. We thus add μ_3 to the set of tracked norms $\mathcal{N}_c$. The execution of *line 4*, the instruction `x = x→next`, decrements norm μ_1 and increments μ_2. The instruction at *line 7* (in the inner loop) decrements the value of norm μ_2. By preserving the original control flow, but replacing all pointer instructions by the respective changes in norm values they provoke, we finally obtain the integer program depicted in Algorithm 2.

Bounds analysis. Finally, we apply the bounds analyser, Loopus [8], to infer a bound on the number of times that the loops at *line 3* and *line 6* of the integer abstraction Algorithm 2 can be iterated during the program run. In the following we comment on the analysis underlying Loopus (for a detailed description, we refer the reader to [1] or [19]): The norm $x\langle\text{next}^*\rangle\texttt{NULL}$ which decreases on the outer loop is initialized at *line 1* to the norm $list\langle\text{next}^*\rangle\texttt{NULL}$ and never reset. Hence the tool infers the bound $list\langle\text{next}^*\rangle\texttt{NULL}$ for the outer loop. The norm $y\langle\text{next}^*\rangle x$ which decreases in the inner loop of the integer program (*line 6*) is initialized to 0 at *line 2*, and never reset. However, at each execution of *line 4*, $y\langle\text{next}^*\rangle x$ is incremented by one (which models the execution of the statement $x = x \to next$ in the concrete program). Since the number of executions of *line 4* is bounded by the number of executions of the outer loop, the norm $y\langle\text{next}^*\rangle x$ is thus incremented at most $list\langle\text{next}^*\rangle\texttt{NULL}$ (the bound of the outer loop) times and hence the overall number of times the norm $y\langle\text{next}^*\rangle x$ may be decremented in the inner loop is bound by $list\langle\text{next}^*\rangle\texttt{NULL}$. The overall complexity of the example is the sum of both loop bounds, i.e. $2 \cdot list\langle\text{next}^*\rangle\texttt{NULL}$.

3 Preliminaries

This section introduces the basic notions used, the considered programs, as well as our requirements on the underlying shape analysis and on the way it should represent reachable memory configurations and their possible changes.

3.1 Program Model

For the rest of the paper, we will use $\mathbb{V}_p$ to denote the set of *pointer variables*, $\mathbb{V}_i$ the set of *integer variables*, $\mathbb{S}_p$ the set of *pointer selectors* (or fields) of dynamic data structures, and $\mathbb{S}_i$ the set of *integer selectors*. We assume all these sets to be finite and mutually disjoint. Let $\mathbb{V} = \mathbb{V}_p \cup \mathbb{V}_i$ be the set of all program variables and $\mathbb{S} = \mathbb{S}_p \cup \mathbb{S}_i$ be the set of all selectors. Finally, let NULL denote the *null* pointer and assume that $\texttt{NULL} \notin \mathbb{V} \cup \mathbb{S}$.

We consider *pointer manipulating program statements* from the set STMTS_p generated by the following grammar where $x, y \in \mathbb{V}_p$, $z \in \mathbb{V}_p \cup \{\texttt{NULL}\}$ and $sel \in \mathbb{S}_p$:

$$\begin{aligned}\texttt{stmt}_p ::= {} & x = z \mid x = y \rightarrow sel \mid x \rightarrow sel = z \mid \\ & x = \texttt{malloc()} \mid \texttt{free(}x\texttt{)} \mid x == z \mid x \neq z\end{aligned}$$

Further, we consider *integer manipulating program statements* from the set STMTS_i generated by the following grammar where $x \in \mathbb{V}_i$, $y \in \mathbb{V}_p$, $sel \in \mathbb{S}_i$, $c \in \mathbb{Z}$, and f is an integer operation (more complex statements could easily be added too):

$$\begin{aligned}\texttt{stmt}_i ::= {} & x = op \mid x = f(op, op) \mid y \rightarrow sel = op \mid x == op \mid x \neq op \\ op ::= {} & c \mid x \mid y \rightarrow sel\end{aligned}$$

Finally, we let $\text{STMTS} = \text{STMTS}_p \cup \text{STMTS}_i$.

Control-flow graphs. A *control-flow graph* (CFG) is a tuple $G = (\textsc{Loc}, T, l_b, l_e)$ where $\textsc{Loc}$ is a finite set of program *locations*, $T \subseteq \textsc{Loc} \times \text{STMTS} \times \textsc{Loc}$ is a finite set of *transitions* (sometimes also called *edges*), $l_b \in \textsc{Loc}$ is the *initial (starting) location*, and $l_e \in \textsc{Loc}$ is the *final location*.

Let $G = (\textsc{Loc}, T, l_b, l_e)$ be a CFG. A path in G of length $n \geq 0$ is a sequence of transitions $t_0 \ldots t_n = (l_0, st_0, l_1)(l_1, st_1, l_2) \ldots (l_n, st_n, l_{n+1})$ such that $t_i \in T$ for all $0 \leq i \leq n$. We denote the set of all such paths by Φ_G. For a given location l, we denote by Φ_G^l the set of paths where $l_0 = l$. Given locations $l_1, l_2 \in \textsc{Loc}$, we say l_1 dominates l_2 (and denote it by $l_1 \succ l_2$) iff all paths to l_2 in $\Phi_G^{l_b}$ lead through l_1. We call a transition $(l, st, h) \in T$ a *back-edge* iff $h \succ l$. We call the location h a *loop header* and denote the set of its back-edges as T_h. Further, we denote the set of all loop headers as $\text{LH} \subseteq \textsc{Loc}$. Note that, for a loop header h_n of a loop nested in some outer loop with a loop header h_o, we have $h_o \succ h_n$.

Loops. Given a CFG $G = (\textsc{Loc}, T, l_b, l_e)$ with a set of loop headers LH, a loop L with a header $h_L \in \text{LH}$ is the sub-CFG $L' = (\textsc{Loc}', T_{|\textsc{Loc}'}, h_L, h_L)$ where $\textsc{Loc}' = \{l \in \textsc{Loc} \mid \exists n \geq 0\ \exists (l_0, st_0, l_1) \ldots (l_n, st_n, l_{n+1}) \in \Phi_G : l_0 = l_{n+1} = h_L \wedge (\exists 0 \leq i \leq n : l = l_i) \wedge (\forall 1 \leq j \leq n : h_L \succ l_j)\}$, i.e., the set of locations on cyclic paths from h_L (but not crossing the header of any outer loop in which L might be nested), and $T_{|\textsc{Loc}'}$ is the restriction of T to $\textsc{Loc}'$. We will denote the set of all program loops as $\mathcal{L}$.

3.2 Memory Configurations

Let $\mathbb{V}_p$, $\mathbb{V}_i$, $\mathbb{S}_p$, and $\mathbb{S}_i$ be sets of pointer variables, integer variables, pointer selectors, and integer selectors, respectively, as defined in the previous. We view *memory configurations*, i.e., *shapes*, as triples $s = (M, \sigma, \nu)$ where (1) M is a finite set of memory locations, $\texttt{NULL} \notin M$, $M \cap \mathbb{Z} = \emptyset$, (2) $\sigma : (M \times \mathbb{S}_p \rightarrow M \cup \{\texttt{NULL}\}) \cup (M \times \mathbb{S}_i \rightarrow \mathbb{Z})$ is a function defining values of selectors, and (3) $\nu : (\mathbb{V}_p \rightarrow M \cup \{\texttt{NULL}\}) \cup (\mathbb{V}_i \rightarrow \mathbb{Z})$ is a function defining values of program variables. We denote the set of all such shapes by $\mathcal{S}$. Note that a shape is basically an oriented graph, also called a *shape graph*, with nodes from $M \cup \mathbb{Z} \cup \{\texttt{NULL}\}$, edges labelled by selectors, and some of the nodes referred to by the program variables. For simplicity, we do not explicitly deal with undefined values of pointers in what follows. For the purposes of our analysis, they can be considered equal to null values. If the program may crash due to using them, we assume this to be revealed by the shape analysis phase.

We assume that the shape analyser used within our approach works with a set $\mathcal{A}$ of *abstract shape representations* (ASRs), which can be automata, formulae, symbolic graphs, etc. This is, each ASR $A \in \mathcal{A}$ represents a (finite or infinite) set of shapes $[\![A]\!] \subseteq \mathcal{S}$. Allowing for disjunctive abstract representations, we assume that the shape analyser will label each location of the CFG of a program by a set of ASRs overapproximating the set of shapes reachable at that location. Moreover, we assume that the shape analyser introduces a special successor relation between ASRs whenever they label locations linked by a transition s.t. the statement of the transition may be executed between some shapes encoded by the ASRs. This leads to a notion of annotated CFGs defined below.

Annotated CFGs. An *annotated CFG* (ACFG) Γ is a triple $\Gamma = (G, \lambda, \rho)$ where $G = (\textsc{Loc}, T, l_b, l_e)$ is a CFG, $\lambda : \textsc{Loc} \rightarrow 2^{\mathcal{A}}$ is a function mapping locations to sets of ASRs generated by the underlying shape analyser for the particular locations, and $\rho \subseteq (\textsc{Loc} \times \mathcal{A}) \times (\textsc{Loc} \times \mathcal{A})$ is a *successor relation* on pairs of locations and ASRs where $((l_1, A_1), (l_2, A_2)) \in \rho$ iff $A_1 \in \lambda(l_1)$, $A_2 \in \lambda(l_2)$, and there is a transition $(l_1, st, l_2) \in T$ and shapes $s_1 \in [\![A_1]\!]$, $s_2 \in [\![A_2]\!]$ such that st transforms s_1 into s_2.

4 Numerical Measures on Dynamic Data Structures

Our approach uses a notion of *shape norms* based on regular expressions that encode sets of selector paths between some memory locations. Intuitively, we assume that the program needs to traverse these paths and hence their length determines (or at least contributes to) the complexity of the algorithm. Typically, one considers selector paths between two memory locations pointed by some pointer variables or between a location pointed by a variable and `NULL`. However, one can also use paths between a source location pointed by some variable and any location containing some specific data value.

For a concrete memory configuration, the numerical value of a shape norm corresponds to the *supremum* of the lengths of the paths represented by the regular expression of the norm. Indeed, in the worst case, the program may follow the longest (possibly cyclic and hence infinite) path in the memory. However, note that our analysis does usually not work with concrete values of shape norms since we work with ASRs and hence need to reason about the values of a given norm over potentially infinite sets of shapes. Instead, we track relative changes (i.e., increments, decrements) of the norms in a way consistent with all shapes in a given ASR. An exception to this is the case where the value of a norm is equal to a constant for all shapes in the ASR (e.g., after the statement $y = x \rightarrow next$, the distance from x to y via $next$ is always 1).

When analyzing a program, we first infer an initial set of *candidate norms* $\mathcal{N}_c$ (i.e., norms potentially useful for establishing resource bounds of the given program) from the CFG of the program—this set may later be extended if we realize some more norms may be useful. Subsequently, we derive as precisely as possible the effects (i.e., increments, decrements, or resets) that particular program statements have on the values of the candidate norms in shapes represented by the different ASRs obtained from shape analysis. The obtained set of shape norms $\mathcal{N}_c$ together with their relative changes could then be directly used to prove termination and to subsequently derive bounds on the loops by trying to form lexicographic norm vectors. The values of these vectors should be lexicographically ordered and have the property that the value is decreased by each loop iteration. However, we instead use $\mathcal{N}_c$ to generate a *numerical program* simulating the original program, which allows us to leverage the strength of current termination and resource bounds analysers for numerical programs as well as to deal with termination and/or resource bounds arguments combining heap and numerical measures.

Below, we first formalize the notion of shape norms and then describe our approach to generating the initial set of candidate norms.

4.1 Shape Norms

Let $\mathbb{S}_p$ be the set of selectors. In what follows, we will use the set $\mathrm{RE}_{\mathbb{S}}$ of restricted regular expressions $\mathtt{re}$ over $\mathbb{S}_p$ defined as follows:

$$\mathtt{re} ::= \mathtt{ru}^* \qquad \mathtt{ru} ::= sel \mid \mathtt{ru} + \mathtt{ru} \qquad sel \in \mathbb{S}.$$

Below, the $\mathtt{ru}$ sub-expressions are called *regular units*, sometimes distinguishing *selector units* (sel) and *join units* ($\mathtt{ru} + \mathtt{ru}$). For $re \in \mathrm{RE}_{\mathbb{S}}$, we denote the language of selector paths described by re as $\mathcal{L}_{re}$. Intuitively, when we analyse the control-flow graph of a program for traversals through selectors, a join unit corresponds to a branching of the control-flow, and the star expression ($\mathtt{re}^*$) to a loop.

Our notion of selector regular expressions can be extended with *concatenation units* ($\mathtt{ru}.\mathtt{ru}$) and *nested star units* ($\mathtt{ru}^*$), corresponding to sequences of unit traversals and nested loop traversals, respectively. Concatenation units are supported in our tool. However, since their introduction brings in many (quite technical) corner cases, we limit ourselves to the join units to simplify the presentation. On the other hand, extending the techniques below by nested stars seems to be more complicated, and we leave it for future work. Nevertheless, note that we did not find it much useful in our experiments as it would correspond to using the same variable as the iterator of several nested loops (while usually different pointer variables are used as the iterators of the loops).

Let $\mathbb{V}_p$ be a set of pointer variables and $\mathbb{S}_i$ a set of data selectors. We use $\mathcal{P} = \mathbb{V}_p \cup \{\mathtt{NULL}\} \cup \{[.data = k] \mid k \in \mathbb{Z}, data \in \mathbb{S}_i\}$ to refer to locations of memory configurations (shapes) of a program. While $x \in \mathbb{V}_p$ denotes the location that is pointed by the pointer variable x, and NULL denotes the special null location, $[.data = k]$ denotes any memory location whose selector $data$ has the value $k \in \mathbb{Z}$. A numerical measure μ on a memory configuration, i.e., a *shape norm*, is a triple $(x, \mathtt{re}, y) \in \mathbb{V}_p \times \mathrm{RE}_{\mathbb{S}} \times \mathcal{P}$. We will use $\mathcal{N}$ to denote the set of all shape norms, and, further, we will use $x\langle\mathtt{ru}^*\rangle y$ as a shorthand for the triple $(x, \mathtt{ru}^*, y) \in \mathcal{N}$.

As we have already mentioned above, we are interested in evaluating norms over ASRs, not over concrete shapes. Moreover, up to the cases where a norm has the same constant value for all shapes in an ASR, we are not interested in absolute values of the norms at all, and we instead track changes of the values of the norms only. However, in order to be able to soundly speak about such changes, we need to first define the value of a norm for a shape.

We will define the value of norms in terms of graphs. For this, we first define the notion of the height of a pointed graph. Then we describe how to obtain a pointed graph for a pair of a shape graph and a norm.

Pointed graphs. A *pointed graph* $\mathcal{G} = (N, E, n)$ consists of a set of nodes N, a set of directed edges $E \subseteq N \times N$ and, a source node $n \in N$. A path π is a finite sequence of nodes $n_0, \cdots, n_l$ such that $(n_i, n_{i+1}) \in E$ for all $0 \leq i < l$. We call $|\pi| = l$ the *length* of the path. We say π starts in n if $n_0 = n$. We define the *height* of $\mathcal{G}$ by setting $|\mathcal{G}| = \sup\{|\pi| \mid \text{path } \pi \text{ starts in } n\}$ where we set $\sup D = \infty$ for an infinite set $D \subseteq \mathbb{N}$. We note that, for a finite graph $\mathcal{G} = (N, E, n)$, we have $|\mathcal{G}| = \infty$ iff there is a cycle reachable from n.

Pointed graphs associated to shape graphs and null-terminated norms. We first consider norms μ that end in NULL, i.e, $\mu = x\langle \mathtt{ru}^*\rangle\mathtt{NULL}$. For a shape $s = (M, \sigma, \nu) \in \mathcal{S}$, we define the associated pointed graph $\mathcal{G}_s^{x\langle \mathtt{ru}^*\rangle \mathtt{NULL}} = (M \cup \{\mathtt{NULL}\}, E, \nu(x))$ where $E = \{(n_1, n_2) \in (M \cup \{\mathtt{NULL}\}) \times (M \cup \{\mathtt{NULL}\}) \mid$ there is path from n_1 to n_2 in s s.t. the string of selectors along the path matches the regular expression $\mathtt{ru}\}$.

Pointed graphs associated to shape graphs and non-null-terminated norms. We now consider a norm $\mu = x\langle \mathtt{ru}^*\rangle y$ with $y \in \mathcal{P} \setminus \{\mathtt{NULL}\}$. For a shape $s = (M, \sigma, \nu) \in \mathcal{S}$, we set $s(y) = \{\nu(y)\}$ for $y \in \mathbb{V}_p$, and $s(y) = \{m \in M \mid \sigma(m, data) = k\}$ for $y = [.data = k]$. We define the shape $s[y/\mathtt{NULL}] = (M \setminus s(y), \sigma[y/\mathtt{NULL}], \nu[y/\mathtt{NULL}])$ where (1) $\sigma[y/\mathtt{NULL}](m, sel) = \sigma(m, sel)$ if $m \notin s(y)$ and $\sigma[y/\mathtt{NULL}](m, sel) = \mathtt{NULL}$ otherwise, and (2) $\nu[y/\mathtt{NULL}](x) = \nu(x)$ if $\nu(x) \notin s(y)$ and $\nu[y/\mathtt{NULL}](x) = \mathtt{NULL}$ otherwise. We define the associated pointed graph as $\mathcal{G}_s^{x\langle \mathtt{ru}^*\rangle y} = \mathcal{G}_{s[y/\mathtt{NULL}]}^{x\langle \mathtt{ru}^*\rangle \mathtt{NULL}}$.

Values of shape norms. We are now ready to define values of shape norms in shapes. In particular, the value of a norm $\mu \in \mathcal{N}$ in a shape $s \in \mathcal{S}$, denoted $\|\mu\|_s$, is a value from the set $\mathbb{N} \cup \{\infty\}$ defined such that $\|\mu\|_s = |\mathcal{G}_s^\mu|$. This is, the value of the norm μ in the shape s is defined as the height of the associated pointed graph.

The intuition behind the above definition is the following. The pointed graph associated to a norm $\mu = x\langle \mathtt{ru}^*\rangle y$ makes the instances of the regular expression $\mathtt{ru}$ explicit. The height of the pointed graph corresponds to the longest chain of instances of the expression $\mathtt{ru}$ in the given shape graph. The intuition behind replacing the targets of norms with NULL stems from the fact that one either reaches the replaced target (and program will terminate naturally) or reaches the NULL, dereferences it and thus crashes (hence terminating unnaturally). However, since our method uses the results of a preceding shape analysis, we can assume memory safety and exclude termination by crash. In case there exists a cycle in the shape reachable from the source point x, the value of the norm is infinite. In such a case the norm is unusable for the later complexity analysis, hinting at the potential non-termination of the program under analysis.

We now generalize the notion of values of norms from particular shapes to ASRs. The value of a norm $\mu \in \mathcal{N}$ over a set of shapes given by an ASR $A \in \mathcal{A}$, denoted $\|\mu\|_A$, is a value from the set $\mathbb{N} \cup \{\infty, \omega\}$ defined such that $\|\mu\|_A = \sup_\omega \{\|\mu\|_s \mid s \in [\![A]\!]\}$ where (i) $\sup_\omega X = \omega$ iff $\infty \in X$ and (ii) $\sup_\omega(X) = \sup X$ otherwise. Intuitively, we need to distinguish the case when some of the represented shapes contains a cyclic selector path and the case where the ASR represents a set of shapes containing paths of finite but unbounded length (as, e.g., in the case when the ASR represents all acyclic lists of any length). Indeed, in the former case, the program may loop over the cyclic selector path while, in the latter case, it will terminate, but its running time cannot be bounded by a constant (it is bounded, e.g., by the length of the encountered list).

4.2 Deriving the Set of Candidate Shape Norms $\mathcal{N}_c$

We now discuss how we infer a suitable initial set of norm candidates. Note that this set is only an initial set of norm candidates that could be useful for inferring the bounds on the program loops. It is extended when tracking norm changes as discussed in Section 5. For each program loop L we derive a set of norm candidates in the following three steps:

1. We inspect all of the conditions of the loop L which involve pointer variables wrt the program model of Section 3.1 and declare each variable that appears in such a condition as *relevant*. E.g., for our running example Algorithm 1 variables x and y are declared as relevant due to the condition $x \neq y$ in line *line 6*.

2. We iterate over all simple loop paths of L (a loop path is any path which stays inside the loop L, and starts from and returns to the loop header; a loop path is simple if it does not visit any location twice except for the loop header) and derive a set of selectors $S_x \subseteq \mathbb{S}_p$ for each relevant variable x: Given a simple loop path *slp* and a relevant variable x, we perform a symbolic backward execution to compute the effect of *slp* on x, i.e., we derive an assignment $x = exp$ such that exp captures how x is changed when executing *slp*. For example, for our running example Algorithm 1 we infer $x = x \rightarrow next$, $y = y$ for both simple loop paths of the outer loop and $y = y \rightarrow next$, $x = x$ for the single simple loop path of the inner loop. In case exp is of form $x \rightarrow sel$, i.e., the effect of the loop path is dereferencing variable x by some selector $sel \in \mathbb{S}_p$, we add sel to S_x. This basic approach can be easily extended to handle consecutive dereferences of the same pointer over different selectors: We can deal with expressions of the form $exp = x \rightarrow sel_1 \rightarrow sel_2$ by adding $sel_1.sel_2$ to S_x.

3. Finally, we consider all subsets $T \subseteq S_x$ and create norms for each $T = \{sel_1, ..., sel_l\}$ using the regular expression $join(T) = sel_1 + ... + sel_l$. The candidate norms $\mathcal{N}_L$ created for different forms of conditions of the loop L are given in the right column of Fig. 1. For example, for our running example in Sect. 2, we create norms $x\langle \text{next}^* \rangle \texttt{NULL}$ for the outer cycle, and norms $x\langle \text{next}^* \rangle y$ and $y\langle \text{next}^* \rangle x$ for the inner loop.

The overall set of tracked norm candidates $\mathcal{N}_c$ is set to the union of norm candidates over all loops in the program, i.e. $\mathcal{N}_c = \bigcup_{L \in \mathcal{L}} \mathcal{N}_L$. For each norm from $\mathcal{N}_c$ we track its size-changes, as we discuss in Section 5.

Condition of L	**Candidate Norms** $\mathcal{N}_L$
$x \circ y$	$\{\, x\langle join(\text{T})^* \rangle y \mid T \subseteq S_x \}$ $\cup \{\, y\langle join(\text{T})^* \rangle x \mid T \subseteq S_x \}$
$x \circ \texttt{NULL}$	$\{\, x\langle join(\text{T})^* \rangle \texttt{NULL} \mid T \subseteq S_x \}$
$x \rightarrow d \circ k$	$\{\, x\langle join(\text{T})^* \rangle [.data = k] \mid T \subseteq S_x \}$

Fig. 1: Norm candidates $\mathcal{N}_L$ for a loop L, $\circ \in \{=, \neq\}$

Note that we can optimize the size of $\mathcal{N}_c$ by pruning irrelevant norms, e.g. those that never decrease; the concrete heuristics are described in Section 6.2.

5 From Shapes to Norm Changes

In the previous section, we have shown how to derive an initial set of candidate norms $\mathcal{N}_c$ that are likely to be useful for deriving bounds on the number of executions of the different program loops. This section describes how to derive numerical changes of the values of these norms, allowing us to derive a numeric program simulating the original program from the point of view of its runtime complexity. During this process, new norms may be found as potentially useful, which leads to an extension of $\mathcal{N}_c$ and to a re-generation of the numeric program such that the newly added norms are also tracked.

In the numeric program, we introduce a *numeric variable* for each candidate norm. By a slight abuse of the notation, we use the norms themselves to denote the corresponding numeric variables, so, e.g., we will write $x\langle \text{u}^* \rangle \texttt{NULL} == 0$ to denote that the value of the numeric variable representing the norm $x\langle \text{u}^* \rangle \texttt{NULL}$ is zero. The values these variables may get are from the set $\mathbb{N} \cup \{\omega\}$ with omega representing an infinite distance (due to a loop in a shape). In what follows, we assume that any increment/decrement of ω yields ω again and that ω is larger than any natural number. Note that we do not need a special value to represent ∞ for describing a finite distance without an explicit bound. For that, we will simply introduce a fresh variable constrained to be smaller than ω.

The *numeric program* is constructed using the ACFG $\Gamma = (G, \lambda, \rho)$ built on top of the CFG $G = (\textsc{Loc}, T, l_b, l_e)$ of the original program. The original control flow is preserved except that each location $l \in \textsc{Loc}$ is replaced by a separate copy for each ASR labelling it, i.e., it is replaced by locations (l, A) for each $A \in \lambda(l)$. Transitions between the new locations are obtained by copying the original transitions between those pairs of locations and ASRs that are related by the successor relation, i.e., a transition (l_1, st, l_2) is lifted to $((l_1, A_1), st, (l_2, A_2))$ whenever $((l_1, A_1), (l_2, A_2)) \in \rho$. Subsequently, each pointer-dependent condition labelling some edge in the extended CFG is translated to a condition on the numeric variables corresponding to the shape norms from $\mathcal{N}_c$. Likewise, each edge originally labeled by a pointer-manipulating statement is relabeled by numerical updates of the values of the concerned norm variables. Integer conditions and statements are left untouched.

Soundness of the abstraction. The translation of the pointer statements described below is done such that, for any path π in the CFG of a program and the shape s resulting from executing π, the values of the numeric norm variables obtained by executing the corresponding path in the numeric program conservatively *over-approximate* the values of the norms over s. This is, if the numeric variable corresponding to some norm μ can reach a value $n \in \mathbb{N} \cup \{\omega\}$ through the path π with pointer statements replaced as described below, then $\|\mu\|_s \leq n$. *As a consequence, we get that every bound obtained for the integer abstraction is a bound of the original program.*

Given the above, the translation of *pointer conditions* is easy. We translate each condition `x == NULL` to a disjunction of tests $x\langle \text{u}^* \rangle \texttt{NULL} == 0$ over all regular units u such that $x\langle \text{u}^* \rangle \texttt{NULL} \in \mathcal{N}_c$. Likewise, every condition `x == y` is translated to a disjunction of conditions of the form $x\langle \text{u}^* \rangle y == 0$ over all regular units u such that $x\langle \text{u}^* \rangle y \in \mathcal{N}_c$. Pointer inequalities are then translated to a negation of the conditions formed as above, leading to a conjunction of inequalities on numeric norm variables.

Handling *data-related pointer tests* of the form $\texttt{x} \rightarrow \texttt{data} == \texttt{y}$ is more complex. Consider such a test on an edge starting from a location-ASR pair (l, A). Currently, we can handle the test in a non-trivial way only if y evaluates to the same constant value in all shapes represented by A, i.e., if there is some $k \in \mathbb{N}$ such that $\nu(y) = k$ for all shapes $(M, \sigma, \nu) \in [\![A]\!]$. In this case, the test is translated to a disjunction of conditions of the form $x\langle \text{u}^* \rangle [.data = k] == 0$ over all regular units u such that $x\langle \text{u}^* \rangle [.data = k] \in \mathcal{N}_c$. Otherwise, the test is left out—a better solution is an interesting issue for future work, possibly requiring more advanced shape analysis and a tighter integration with it. Data-related pointer non-equalities can then again be treated by negation of the equality test (provided y evaluates to a constant value).

Finally, after a successful equality test (of any of the above kinds), all numeric norm variables that appeared in the disjunctive condition used are set to zero. All other variables (and all variables in general for an inequality test) keep their original value.

Next, we describe how we translate non-destructive, destructive, and data-related pointer statements other than tests. The translation can lead to decrements, resets, or increments of the numeric norm variables corresponding to the norms in $\mathcal{N}_c$. In case, we realize that we need some norm $\mu' \notin \mathcal{N}_c$ to describe the value of some current candidate norm $\mu \in \mathcal{N}_c$, we add μ' into $\mathcal{N}_c$ and restart the translation process (in practice, of course, the results of the previously performed translation steps can be reused). Such a situation can happen, e.g., when $\mathcal{N}_c = \{x\langle\text{next}^*\rangle\texttt{NULL}\}$ and we encounter an instruction `x = list`, which generates a reset of the norm $x\langle\text{next}^*\rangle\texttt{NULL}$ to the value of the norm $list\langle\text{next}^*\rangle\texttt{NULL}$. The latter norm is then added into $\mathcal{N}_c$.[1]

The rules for translating non-destructive, destructive, and data-related pointer updates to the corresponding updates on numeric norm variables are given in Fig. 2, 3, and 4, respectively. Before commenting on them in more detail, we first make several general notes. First, values of norms of the form $x\langle\text{u}^*\rangle x$ are always zero, and hence we do not consider them in the rules. Next, let $u = sel_1 + \ldots + sel_n$, $n \geq 1$, be a regular join unit. We will write $sel \in u$ iff $sel = sel_i$ for some $1 \leq i \leq n$. We denote new values of norms using an overline, and the old values without an overline. The norms that are not mentioned in a given rule keep implicitly the same value.

Finally, in rules describing how the value of a norm variable μ is changed by firing some statement between ASRs A_1 and A_2, we often use constructions of the form $\overline{\mu} \stackrel{\circ}{=} expr$ where $expr$ is an expression on norm variables. This construction constrains the new value of μ using the current values of norm variables or using directly the ASRs encountered, depending on what of this is more precise. First, if μ has the same natural value in all shapes in $[\![A_2]\!]$, i.e., if $\|\mu\|_{A_2} \in \mathbb{N}$, then we let $\overline{\mu} = \|\mu\|_{A_2}$. Otherwise, if the value of μ is infinite in A_1 and unbounded but finite in A_2, i.e., if $\|\mu\|_{A_1} = \omega$ and $\|\mu\|_{A_2} = \infty$, we constrain the new value of μ by the constraint $\overline{\mu} = v \wedge v < \infty$ where v is a fresh numeric variable.[2] The same constraint with a fresh variable is used when $\|\mu\|_{A_2} = \infty$ and $expr = \omega$. Otherwise, we let $\overline{\mu} = expr$.

The described translation allows for sound resource bounds analysis. Indeed, for each run of the original pointer program, there will exist one run in the derived numeric program where the norms get exact/overapproximated values. Provided that the underlying bounds analyser is sound in that it returns worst case bounds, the bounds obtained for the numeric program will not be smaller than the bounds of the original program.

5.1 Non-Destructive Pointer Updates

We now comment more on the less obvious parts of the rules for non-destructive pointer updates from Fig. 2. Concerning the rule for $x = \texttt{NULL}$, Case 1 reflects the fact that we always consider all paths from x limited by either the designated target w or, implicitly, `NULL`. Hence, after $x = \texttt{NULL}$, the distance is always 0. Likewise, in Case 1 of $x = malloc()$, the distance is always 1 as we assume all fields of the newly allocated cell to be nullified, and so the paths consist of the newly allocated cell only. Case 2 of $x = malloc()$ is based on that we assume the newly allocated cell to be unreachable

[1] Alternatively, one could use a more complex initial static analysis that would cover, although may be less precisely, even such dependencies among norms.

[2] Intuitively, this case is used, e.g., when $\mu = x\langle\text{n}^*\rangle\texttt{NULL}$, and the encountered pointer statement cuts an ASR representing cyclic lists of any length pointed by x to an ASR representing acyclic `NULL`-terminated lists pointed by x. Naturally, when one subsequently starts a traversal of the list, it will terminate though in an unknown number of steps.

$$\dfrac{\texttt{[x = NULL]} \quad \forall w \in \mathcal{P},\ \forall z \in \mathbb{V}_p \setminus \{x\}}{\begin{array}{rcll} \overline{x\langle \mathrm{u}^* \rangle w} & = & 0 & (1) \\ \overline{z\langle \mathrm{u}^* \rangle x} & \stackrel{\circ}{=} & z\langle \mathrm{u}^* \rangle \texttt{NULL} & (2) \end{array}}
\qquad
\dfrac{\texttt{[x = malloc()]} \quad \forall w \in \mathcal{P} \setminus \{x\},\ \forall z \in \mathbb{V}_p}{\begin{array}{rcll} \overline{x\langle \mathrm{u}^* \rangle w} & = & 1 & (1) \\ \overline{z\langle \mathrm{u}^* \rangle x} & \stackrel{\circ}{=} & z\langle \mathrm{u}^* \rangle \texttt{NULL} & (2) \end{array}}$$

$$\dfrac{\texttt{[free(x)]} \quad \forall z \in \mathbb{V}_p,\ \forall w \in \mathcal{P}}{\overline{z\langle \mathrm{u}^* \rangle w} \stackrel{\circ}{=} \begin{cases} z\langle \mathrm{u}^* \rangle x & \texttt{AllPathsThr}(A_1, u, z, w, x)) \\ z\langle \mathrm{u}^* \rangle w & \text{otherwise} \end{cases}}$$

$$\dfrac{\begin{array}{c}\texttt{[x = y}\rightarrow\texttt{n (alias)]} \\ \exists v \in \texttt{AliasNext}(A_1, y, \mathtt{n}) \\ \forall w \in \mathcal{P}\ \forall z \in \mathbb{V}_p\end{array}}{\begin{array}{rcll} \overline{x\langle \mathrm{u}^* \rangle w} & \stackrel{\circ}{=} & v\langle \mathrm{u}^* \rangle w & (1) \\ \overline{z\langle \mathrm{u}^* \rangle x} & \stackrel{\circ}{=} & z\langle \mathrm{u}^* \rangle v & (2) \end{array}}
\qquad
\dfrac{\begin{array}{c}\texttt{[x = y}\rightarrow\texttt{n (non-unit)]} \\ \mathtt{n} \notin u,\ \forall w \in \mathcal{P},\ \forall z \in \mathbb{V}_p\end{array}}{\begin{array}{rcll} \overline{x\langle \mathrm{u}^* \rangle w} & \stackrel{\circ}{=} & \omega & (1) \\ \overline{z\langle \mathrm{u}^* \rangle x} & \stackrel{\circ}{=} & \omega & (2) \end{array}}$$

$$\dfrac{\begin{array}{c}\texttt{[x = y}\rightarrow\texttt{n (unit)]} \\ \mathtt{n} \in u,\ x \neq y,\ \forall t \in \texttt{Alias}(A_1, y) \\ \forall s \in \texttt{MayAlias}(A_1, y) \\ \forall w \in \mathcal{P} \setminus \texttt{MayAlias}(A_1, y) \\ \forall z \in \mathbb{V}_p \setminus \texttt{Alias}(A_1, y)\end{array}}{\begin{array}{rcll} \overline{t\langle \mathrm{u}^* \rangle x} & \stackrel{\circ}{=} & t\langle \mathrm{u}^* \rangle \texttt{NULL} & (1) \\ \overline{x\langle \mathrm{u}^* \rangle s} & \stackrel{\circ}{=} & y\langle \mathrm{u}^* \rangle \texttt{NULL} - 1 & (2) \\ \overline{x\langle \mathrm{u}^* \rangle w} & \stackrel{\circ}{=} & y\langle \mathrm{u}^* \rangle w - 1 & (3) \\ \overline{z\langle \mathrm{u}^* \rangle x} & \stackrel{\circ}{=} & z\langle \mathrm{u}^* \rangle y + \overline{y\langle \mathrm{u}^* \rangle x} & (4) \end{array}}$$

$$\dfrac{\texttt{[x = y]} \quad \forall z \in \mathbb{V}_p\ \forall w \in \mathcal{P}}{\overline{x\langle \mathrm{u}^* \rangle w} \stackrel{\circ}{=} y\langle \mathrm{u}^* \rangle w \ (1) \qquad \overline{z\langle \mathrm{u}^* \rangle x} \stackrel{\circ}{=} z\langle \mathrm{u}^* \rangle y \ (2)}$$

Fig. 2: Translation rules for *non-destructive pointer updates*. The rules are assumed to be applied between location-ASR pairs (l_1, A_1) and (l_2, A_2) linked by an edge labelled by a non-destructive pointer update with $x, y \in \mathbb{V}_p$, $n \in \mathbb{S}_p$. For all rules with left-hand side of form $\overline{a\langle \mathrm{u}^* \rangle b}$, u ranges over all regular units such that $a\langle \mathrm{u}^* \rangle b \in \mathcal{N}_c$. If the norms used on the right-hand side of any of the applied rules is not in $\mathcal{N}_c$, it is added into $\mathcal{N}_c$, and the analysis is re-run with the new $\mathcal{N}_c$.

from other memory locations, and so any path taken from another memory location towards x will implicitly be bounded by NULL.[3]

Concerning the rules for $free(x)$, the predicate $\texttt{AllPathsThr}(A, u, z, w, x)$ holds iff all paths over selector sequences matching u^* between the location z and the location w go through x in all shapes in $[\![A]\!]$. In this case, clearly, all paths from z to w are shrunk to paths to x by $free(x)$ as x becomes undefined (which we take as equal to NULL for our purposes). Otherwise, we take the old value of the norm since it either stays the same or perhaps gets shorter but in some shapes only.

Concerning the rules for $x = y \rightarrow n$, we first note that, if applicable, the "alias" rule has priority. It is applied when the n-successor of y is pointed by some variable v in all shapes in $[\![A_1]\!]$. Formally, $v \in \texttt{AliasNext}(A, y, \mathtt{n})$ iff $\forall(M, \sigma, \nu) \in [\![A]\!] : \sigma(\nu(y), n) = \nu(v)$. Such an alias can be used to define norms based on x by copying those based on v. Of course, the distance from x to v after the update should be zero, which is assured by the $\stackrel{\circ}{=}$ operator. If there is no such v, and n does not match u, we can only limit the new value of the norm based on the ASR, which is again taken care by the $\stackrel{\circ}{=}$ operator (otherwise we take the worst possibility, i.e., ω).

The most complex rule is that for $x = y \rightarrow n$ when there is no alias for the n-successor of y, and n matches u. First, note that the rule is provided for the case of x being a different variable than y only. We assume statements $x = x \rightarrow n$ to be transformed to a sequence $y = x; x = y \rightarrow n$; for a fresh pointer variable y. In the rules, we then use the following must- and may-alias sets: $\texttt{Alias}(A, y) = \{v \in \mathbb{V}_p \mid \forall(M, \sigma, \nu) \in [\![A]\!] : \nu(y) = \nu(v)\}$ and $\texttt{MayAlias}(A, y) = \{v \in \mathbb{V}_p \mid \exists(M, \sigma, \nu) \in [\![A]\!] : \nu(y) = \nu(v)\}$.

Concerning Case 1, note that u can be a join unit and u^* can match several paths from y that need not go to the new position of x at all (and hence can stop only when reaching NULL), or they can go there, but as there is no variable pointing already to the

[3] We assume that the preceding shape analysis will discover potential problems with a location being freed and re-allocated with some dangling pointers still pointing to it (the ABA problem).

$$\frac{\texttt{[x}\rightarrow\texttt{n = NULL (unit)]} \\ \texttt{n} \in u,\ \forall z \in \mathbb{V}_p,\ \forall w \in \mathcal{P}}{\overline{z\langle \mathrm{u}^* \rangle w} \doteq \begin{cases} z\langle \mathrm{u}^* \rangle x + 1 & \texttt{AllPathsThrFld}(A_1, u, z, w, x, n) \\ z\langle \mathrm{u}^* \rangle w & \text{otherwise} \end{cases}}$$

$$\frac{\begin{array}{c}\texttt{[x}\rightarrow\texttt{n = y (unit)]} \\ \texttt{n} \in u,\ \forall s_1 \in \texttt{Alias}(A_1, x),\ \forall s_2 \in \texttt{MayAlias}(A_1, x) \\ \forall t_1 \in \texttt{Alias}(A_1, y),\ \forall t_2 \in \texttt{MayAlias}(A_1, y) \\ \forall w \in \mathcal{P} \setminus (\texttt{Alias}(A_1, x) \cup \texttt{Alias}(A_1, y)) \\ \forall z \in \mathbb{V}_p \setminus (\texttt{MayAlias}(A_1, x) \cup \texttt{MayAlias}(A_1, y))\end{array}}{\begin{array}{rll} \overline{s_1\langle \mathrm{u}^* \rangle t_1} \doteq & s_1\langle \mathrm{u}^* \rangle t_1 & (1) \\ \overline{s_2\langle \mathrm{u}^* \rangle w} \doteq & \begin{cases} \omega & \texttt{BadLoopClosed}(A_2, u, y, x, w) \\ \overline{s_2\langle \mathrm{u}^* \rangle y} + y\langle \mathrm{u}^* \rangle w & \text{otherwise} \end{cases} & (2) \\ \overline{t_2\langle \mathrm{u}^* \rangle w} \doteq & \begin{cases} \omega & \texttt{BadLoopClosed}(A_2, u, y, x, w) \\ t_2\langle \mathrm{u}^* \rangle w & \text{otherwise} \end{cases} & (3) \\ \overline{z\langle \mathrm{u}^* \rangle w} \doteq & \begin{cases} z\langle \mathrm{u}^* \rangle x + \overline{x\langle \mathrm{u}^* \rangle w} & \texttt{AllPathsThr}(A_2, u, z, w, x)) \\ \max(z\langle \mathrm{u}^* \rangle x + \overline{x\langle \mathrm{u}^* \rangle w}, z\langle \mathrm{u}^* \rangle w) & \texttt{SomePathsThr}(A_2, u, z, w, x)) \\ z\langle \mathrm{u}^* \rangle w & \text{otherwise} \end{cases} & (4) \end{array}}$$

Fig. 3: Translation rules for *destructive pointer updates*. The rules are assumed to be applied between location-ASR pairs (l_1, A_1) and (l_2, A_2) linked by an edge labelled by a destructive pointer update with $x, y \in \mathbb{V}_p$, $n \in \mathbb{S}_p$. The treatment of the regular units u is the same as in Fig. 2.

new position of x, we anyway have to approximate such paths by extending them up to NULL. The case when the only path to x is via n will then be solved by the $\doteq$ operator. The must aliases of y can naturally be treated in an equal way as y in the above.

In Case 2, start by considering paths from x to y. Since we have no alias of the n-successor of y that could help us define the value of the norm, we have to approximate the distance from x to y by extending the paths from x up until NULL. Further note that such paths are a subset of those from y to NULL (since the new position of x is a successor of y). We can thus use $y\langle \mathrm{u}^* \rangle\mathtt{NULL}$ to approximate $\overline{x\langle \mathrm{u}^* \rangle\mathtt{NULL}}$. However, we can constrain the latter distance to be smaller by one. Indeed, if the longest path from y to NULL does not go through the new position of x, the distance from x to NULL is at least by one smaller. On the other hand, if the longest path goes through the new position of x, then we save the step from y to the new position of x. The same reasoning then applies for any variable that may alias y—for those that cannot alias it, one can do better as expressed in the next case.

In Case 3, one can use a similar reasoning as in Case 2 as the paths from y to w include those from the new position of x to w. Note, however, that this reasoning cannot be applied when y may alias with w. In such a case, their distance may be zero, and the distance from the new position of x to w can be bigger, not smaller. Finally, to see correctness of Case 4, note that should there be a longer path over u from z to the n-successor of y than going through y, this longer path will be included into the value of the norm for getting from z to y too since the norm takes into account all u paths either going to y or missing it and then going up until NULL (or looping).

The intuition behind the rule for $x = y$ is similar to the other statements.

5.2 Destructive Pointer Updates

We now proceed to the rules for destructive pointer statements shown in Fig. 3. We start with the translation for the statement $x \rightarrow n = \mathtt{NULL}$, considering the case of n being a unit, i.e., $n \in u$. After this statement, the distance from any source memory location z to any target memory location w either stays the same or decreases. The latter happens

$$\texttt{[x}\!\rightarrow\!\texttt{d = y (data-const)]}$$

$$\begin{array}{c}
\exists k \in \mathbb{Z} : \texttt{ValIsConst}(A_1, y, k),\ \forall z \in \mathbb{V}_p \setminus \{x\},\ \forall l \in \mathbb{Z} \setminus \{k\} \\
\hline
\begin{array}{rcll}
\overline{x\langle \mathrm{u}^* \rangle[.d = k]} & = & 0 & (1) \\
\overline{x\langle \mathrm{u}^* \rangle[.d = l]} & \stackrel{\circ}{=} & \begin{cases} x\langle \mathrm{u}^* \rangle\texttt{NULL} & \texttt{ValMayBe}(A_1, x, d, l) \\ x\langle \mathrm{u}^* \rangle[.d = l] & \text{otherwise} \end{cases} & (2) \\
\overline{z\langle \mathrm{u}^* \rangle[.d = k]} & \stackrel{\circ}{=} & \begin{cases} z\langle \mathrm{u}^* \rangle x & \texttt{AllPathsThr}(A_1, u, z, [.d = k], x) \\ z\langle \mathrm{u}^* \rangle[.d = k] & \text{otherwise} \end{cases} & (3) \\
\overline{z\langle \mathrm{u}^* \rangle[.d = l]} & \stackrel{\circ}{=} & \begin{cases} z\langle \mathrm{u}^* \rangle\texttt{NULL} & \texttt{ValMayBe}(A_1, x, d, l) \\ z\langle \mathrm{u}^* \rangle[.d = l] & \text{otherwise} \end{cases} & (4)
\end{array}
\end{array}$$

$$\texttt{[x}\!\rightarrow\!\texttt{d = y (data-unknown)]}$$

$$\begin{array}{c}
(\neg\exists k \in \mathbb{Z} : \texttt{ValIsConst}(A_1, y, k)),\ \forall z \in \mathbb{V}_p,\ \forall l \in \mathbb{Z} \\
\hline
\overline{z\langle \mathrm{u}^* \rangle[.d = l]} \stackrel{\circ}{=} \begin{cases} z\langle \mathrm{u}^* \rangle[.d = l] & z\langle \mathrm{u}^* \rangle[.d = l] < z\langle \mathrm{u}^* \rangle x \\ z\langle \mathrm{u}^* \rangle[.d = l] & \neg\texttt{ValMayBe}(A_1, x, d, l) \\ z\langle \mathrm{u}^* \rangle\texttt{NULL} & \end{cases}
\end{array}$$

Fig. 4: Translation rules for *data-related pointer updates*. The rules are assumed to be applied between location-ASR pairs (l_1, A_1) and (l_2, A_2) linked by an edge labelled by a data-related pointer update with $x \in \mathbb{V}_p$, $y \in \mathbb{V}_i$, $d \in \mathbb{S}_i$. The treatment of the regular units u is the same as in Fig. 2.

when the changed n-selector of x influences the longest previously existing path from z to w. Identifying this case in general is difficult, but one can reasonably recognise it in common ASRs at least in the situation when all paths between z and w whose selector sequences match u^* go through the n-selector of the memory location marked by x in all shapes represented by the ASR A_1, i.e., $[\![A_1]\!]$. We denote this fact by the predicate $\texttt{AllPathsThrFld}(A_1, u, z, w, x, n)$. In this case, the new distance between z and w clearly corresponds to the old distance between z and x plus one (for the step from x to NULL). In all other cases, we conservatively keep the old value of the distance (up to it can be reduced by the $\stackrel{\circ}{=}$ operator as usual).

Concerning the statement $x \to n = y$, the distance between x and y (and their aliases) can stay the same or get shortened. In Case 1 of the rule for this statement, the latter is reflected in the use of the $\stackrel{\circ}{=}$ operator. In Case 2, we use the predicate $\texttt{BadLoopClosed}(A_2, u, y, x, w)$ to denote a situation when the statement $x \to n = y$ closes a loop (over the u selectors) in at least some shape represented by A_2 such that w does not appear in between of y and x in the loop. Naturally, in such a case, the distance between x (or any of its may-aliases) and w is set to ω. Note that the may-alias is needed in this case since it is enough that this problematic situation arises even in one of the concerned shapes. As for correctness of the other variant of Case 2, note that if there are paths over u^* from x to w not passing through y, they will be covered by $\overline{x\langle \mathrm{u}^* \rangle y}$, which will consider such paths extended up until NULL. In Case 3, note that if the loop is not closed, then the paths from y to w are not influenced.

In Case 4, if all paths from z to w in the shapes represented by A_2 go through x, we can take the original distance of z and x, which does not change between A_1 and A_2 as the change happens after x, and then add the new distance from x to w. If no path from z to w passes x, the distance is not influenced by the statement. If some but not all of the paths pass x, we have to take the maximum of the two previous cases.

As for non-unit cases of the above two statements, i.e., the case when $n \notin u$, the norms do not change since the paths over u^* do not pass the changed selector.

5.3 Data-Related Pointer Updates

Our rules for translating data-related pointer updates are given in Fig. 4. The first of them applies in case the value being written into the data field d of the memory location pointed by x is constant over all shapes represented by the ASR A_1, i.e., if there is some constant $k \in \mathbb{Z}$ such that $\forall(M, \sigma, \nu) \in [\![A]\!] : \nu(y) = k$. This fact is expressed by the $\texttt{ValIsConst}(A_1, y, k)$ predicate. In this case, after the statement $x \to d = y$, the distance from x to a data value k becomes clearly zero. Case 2 captures the fact that if the d-field of x may be l in at least one shape represented by A_1, i.e., if $\exists(M, \sigma, \nu) \in [\![A]\!] : \nu(y) = l$, which is expressed by the $\texttt{ValMayBe}(A_1, x, d, l)$ predicate, the new distance of x to a data value l is approximated by its distance to `NULL`. The reason is that the old data value is re-written, and one cannot say whether another data field with the value l may be reached before one gets to `NULL`. Otherwise, the norm keeps its original value. Case 3 covers the distance from a location z other than x to a data value k. This distance clearly stays the same or can get shorter after the statement. We are able to safely detect the second scenario when all paths from z to a data value k lead through x. In that case, the distance from z to a data value k shrinks to that from z to x. Otherwise, we conservatively keep the norm value unchanged. Finally, Case 4 is an analogy of Case 2.

In case the value being written through a data selector is not constant, which is covered by the second rule of Fig. 4, our approach is currently rather conservative. We keep the original value of the norms between z and a data value l if either this data value is always reached from z before x is reached (the norm takes into account the first occurrence of the data value) or if the re-written value of the data field d of x is not l in any of the shapes represented by A_1 (and hence the original value of the norm is not based on the distance to this particular field). In such a case, the distance between z and the data value l does surely not change. Otherwise, we conservatively approximate the new distance between z and the data value l by the distance over paths matching u^* from z up until `NULL`.

The stress on handling constant values of data may seem quite restricted, but it may still allow one to verify a lot of interesting programs. The reason is that often the programs use various important constants (like 0) to steer their control flow. Moreover, due to data-independence, it is often enough to let programs work with just a few constant values—c.f., e.g., [20,21,22] where just a few data values ("colors") are used when checking various advanced properties of dynamic data structures. Still a better support of data is an interesting issue for future work.

6 Implementation and Experiments

We have implemented our method in a prototype tool called RANGER. The implementation is based on the *Forester* shape analyser [23,22], which represents sets of memory shapes using so-called *forest automata* (FAs). As a back-end *bounds analyser* for the generated numeric programs, we use the *Loopus* tool [8]. We evaluated RANGER on a set of benchmarks including programs manipulating various complex data structures and requiring amortized reasoning for inferring precise bounds. The experimental results we obtained are quite encouraging and show that we were able to leverage both the precise shape analysis of complex data structure provided by Forester as well as the amortized analysis of loop bounds provided by Loopus and for the first time fully-automatically and precisely analyse some challenging programs.

In the rest of the section, we first briefly introduce the Forester tool in some more detail and discuss how we implemented our approach on top of it. Next, we mention various further optimizations we included into the implementation. Then, we present the experiments we performed and their results.

6.1 Implementation on Top of Forester

The Forester shape analyser represents particular shapes by decomposing them into *tuples* of *tree components*, and hence *forests*. In particular, each memory location that is `NULL`, pointed by a pointer variable, or that has multiple incoming pointers becomes a so-called *cut-point*. Shape graphs are cut into tree components at the cut-points, and each cut-point becomes the root of one of the tree components. Leaves of the tree components may then refer back to the roots, which can be used to represent both loops in the shapes as well as multiple paths leading to the same location. Of course, Forester does not work with particular shapes but with sets of shapes. This leads to a need of dealing with tuples of sets of tree components, which are finitely represented using finite *tree automata* (TAs). A tuple of TAs then forms an FA, which we use as the ASR in our implementation.

Hence, we need to be able to implement all the operations used on ASRs in the previous section on FAs. Fortunately, it turns out that this is not at all difficult.[4] In particular, we can implement the various operations by searching through the particular TAs of an FA, following the TA transitions that match the relevant unit expressions.[5] We can then, e.g., easily see whether the distance between some memory locations is constant, finite but unbounded, or infinite. It is constant if the given memory locations are linked by paths in the structure of the involved automata that are of the given constant length. It is finite but unbounded if there is a loop in the TA structure in between the concerned locations (allowing the TA to accept a sequence of any finite length). Finally, the distance is infinite if some path from the source location leads—while not passing through the target location—to some of the roots, which is then in turn referenced back from some leaf node reachable from it. Likewise, one can easily implement checks whether all paths go (or at least some path goes) through some location, whether some variables are aliased (in Forester, this simply corresponds to the variables being associated with the same root), or whether a loop is closed by some destructive update (which must create a reference from a leaf back to a loop).

6.2 Optimizations of the Basic Approach

In RANGER, we use several heuristic optimizations to reduce the size of the generated numeric program. First, we do not translate each pointer statement in isolation as described in Section 5. Instead, we perform the translation per *basic blocks*. Basically, we take the blocks written in the static single assignment form, translate the statements in the blocks as described in Section 5, and then perform various standard simplifications

[4] Based on our experience with other representations of sets of shapes, such as separation logic or symbolic memory graphs, we believe that it would not be difficult with other shape representations either.

[5] In RANGER, we support even concatenation units to some degree, which requires us to look at sequences of TA transitions to match a single unit.

of the generated numeric constraint (evaluation of constant expressions, copy propagation, elimination of variables) using the SMT solver Z3 [24]. In our experience, the size of the generated numeric program can be significantly reduced this way.

Our second optimization aims at reducing the *number of tracked norms*. For that, we use a simple heuristic exploiting the underlying shape analysis and the principle of *variable seeding* [14]. Basically, for each pointer variable x used as a source/target of some norm in $\mathcal{N}_c$, we create a shadow variable x', and remember the position of x at the beginning of a loop by injecting a statement $x' = x$ before the loop. We then use our shape analyser on the extended code to see whether the given variable indeed moves towards the appropriate target location when the loop body is fired once. If we can clearly see that this is not the case due to, e.g., the variable stays at the same location, we remove it from $\mathcal{N}_c$. For illustration, in our example from Section 2, we generate two norms $y\langle \text{next}^* \rangle x$ and $x\langle \text{next}^* \rangle y$ for the loop at *line 6*. Using the above approach, we can see that x is never moved, $x\langle \text{next}^* \rangle y$ is never decreased, and so we can discard it. Moreover, we check which norms decrease at which loop branches (or, more precisely, that cannot be excluded to decrease) and prune away norms that decrease only when some other norm is decreased—we say that such a norm is *subsumed*.

Finally, we reduce the size of the resulting numeric program by taking into account only those changes (resets, increments, and decrements) of the norms whose effect can reach the loop for whose analysis the norm is relevant. For that, we use a slight adaptation of the reset graphs introduced in [1].

6.3 Experimental Evaluation

Our experiments were performed on a machine with an Intel Core i7-2600@3.4 GHz processor and 32 GiB RAM running Debian GNU/Linux. We compared our prototype RANGER with two other tools: APROVE and COSTA. These two tools are, to the best of our knowledge, the closest to RANGER and represent the most recent advancements in bounds analysis of heap-manipulating programs. However, note that both of the tools work over the Java bytecode, and thus we had to translate our benchmarks to Java. For our tool, we report three times — the running times of the shape analysis of Forester (**SA**), generation of the integer program (**IG**), and bounds analysis in Loopus (**BA**). For the other tools, we report times as reported by their web interface[6].

Further, from the outputs of the tools, we extracted the reported complexity of the main program loop, and, if needed, simplified the bounds to the big $\mathcal{O}$ notation. We remark that COSTA uses path-based norms (i.e. a subset of our norms), so it is directly comparable with RANGER. APROVE, however, uses norms based on counting all reachable elements, and is therefore orthogonal to us. But, their norm is always bigger than our norms, thus if it reports an equal or bigger computational complexity we can meaningfully compare the results.

The results are summarized in Table 1. We use TIMEOUT(60S) if a time-out of 60 seconds was hit, ERROR if the tool failed to run the example[7], and UNKNOWN if the tool could not bound the main loop of the example. We divided our benchmarks to three distinct categories. The BASIC category consists of simple list structures — Single-Linked Lists (SLL), Circular Single-Linked Lists (CSLL). In the ADVANCED

[6] We could not directly compare the tools on the same machine due to the tool availability issues.

[7] However, we verified that all our examples are syntactically correct.

Table 1: Experimental results.

Benchmark	Short description	Real bounds	RANGER				APROVE		COSTA	
			Bound	SA	IG	BA	Bound	Time(web)	Bound	Time(web)
BASIC										
SLL-CST	Constant-length SLL Traversal	$\mathcal{O}(1)$	$\mathcal{O}(1)$	0.002s	0.023s	0.011s	$\mathcal{O}(1)$	3.664s	$\mathcal{O}(n)$	0.251s
SLL	SLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.012s	0.087s	0.040s	$\mathcal{O}(n)$	6.434s	$\mathcal{O}(n)$	0.441s
SLL-NESTED	SLL with non-reset nested traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.027s	0.256s	0.057s	$\mathcal{O}(n)$	6.361s	$\mathcal{O}(n^2)$	1.582s
SLL-INT	SLL Traversal with int combination	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.037s	0.275s	0.057s	$\mathcal{O}(n)$	8.945s	$\mathcal{O}(n)$	0.921s
CSLL	CSLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.013s	0.086s	0.032s	ERROR		UNKNOWN	0.383s
CSLL-NT	Non-terminating CSLL Traversal	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.003s	0.001s	0.011s	ERROR		UNKNOWN	0.843s
ADVANCED STRUCTURES										
DLL-NEXT	Forward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.034s	0.518s	0.036s	$\mathcal{O}(n)$	5.954s	UNKNOWN	0.657s
DLL-PREV	Backward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.031s	0.181s	0.044s	$\mathcal{O}(n)$	6.459s	UNKNOWN	0.712s
DLL-NT	Non-terminating DLL Traversal	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.011s	0.004s	0.024s	ERROR		UNKNOWN	0.684s
DLL-INT	Forward DLL Traversal with int combination	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.044s	0.654s	0.044s	$\mathcal{O}(n)$	5.723s	UNKNOWN	0.946s
DLL-PAR	Parallel Forward and Backward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.058s	0.510s	0.069s	ERROR		UNKNOWN	0.668s
BUTTERFLY	Terminating Butterfly Loop	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.005s	0.054s	0.024s	$\mathcal{O}(n)$	7.389s	$\mathcal{O}(n)$	0.883s
BUTTERFLY-INT	Terminating Butterfly Loop with int combination	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	0.026s	0.198s	0.059s	$\mathcal{O}(n)^*$	3.513s	UNKNOWN	0.899s
BUTTERFLY-NT	Non-terminating Butterfly Loop	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.005s	0.090s	0.015s	$\mathcal{O}(n)^*$	7.768s	UNKNOWN	1.701s
BST-DOUBLE	Leftmost BST Traversal with nested Rightmost	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	25.147s	12.523s	0.203s	$\mathcal{O}(n^2)^{**}$	14.547s	UNKNOWN	3.004s
BST-LEFT	Leftmost BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	2.947s	7.321s	0.171s	$\mathcal{O}(n)^{**}$	13.335s	UNKNOWN	2.476s
BST-RIGHT	Rightmost BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	2.895s	5.779s	0.168s	$\mathcal{O}(n)^{**}$	13.007s	UNKNOWN	2.457s
BST-LR	Random BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	3.331s	7.010s	0.188s	$\mathcal{O}(n)^{**}$	14.488s	UNKNOWN	2.619s
2-LVL SL-L1	2-lvl Skip-list Traversal via lvl1	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.309s	0.837s	0.036s	ERROR		UNKNOWN	1.449s
2-LVL SL-L2	2-lvl Skip-list Traversal via lvl2	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.096s	0.526s	0.042s	ERROR		UNKNOWN	1.442s
ADVANCED ALGORITHMS										
FUNCQUEUE	Queue implemented by two SLLs	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.046s	0.519s	0.136s	$\mathcal{O}(n)$	8.222s	UNKNOWN	4.808s
PARTITIONS	SLL Partitioning (from Sec 2)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.094s	0.729s	0.059s	$\mathcal{O}(n^2)$	8.526s	$\mathcal{O}(n^2)$	7.047s
INSERTSORT	Insert Sort on SLL	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	0.041s	0.288s	0.051s	$\mathcal{O}(n^2)$	6.453s	$\mathcal{O}(n^2)$	0.904s
MERGEINNER	Showcase example of Atkey [4]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	3.589s	14.080s	1.502s	$\mathcal{O}(n^2)$	57.935s	TIMEOUT(60s)	

STRUCTURES category, we infer bounds for programs on more complex structures — Binary Trees (BST), Double-Linked Lists (DLL), and even 2-level skip-lists (2-LVL SL). The last category ADVANCED ALGORITHMS includes experiments with various more advanced algorithms, including show cases taken from related work.

In benchmarks marked with (*), APROVE returned an incorrect bound in our experiments. Further, in benchmarks marked with (**), we obtained different bounds from different runs of APROVE even though it was run in exactly the same way. In both cases, we were unable to find the reason. Moreover, we remark that while the measured times show that RANGER is mostly faster, the measured times of APROVE and COSTA may be biased by using different target machines and implementations of the benchmarks (C vs Java).

The results confirm that our approach, conceived as highly parametric in the underlying shape and bounds analyses, allowed us to successfully combine an advanced shape analysis with a state-of-the-art implementation of amortized resource bounds analysis. Due to this, we were able to fully automatically derive tight complexity bounds even over data structures such as 2-level skip-lists, which are challenging even for safety analysis, and to get more precise and tight bounds for algorithms like PARTITIONS or FUNCQUEUE, which require amortized reasoning to get the precise bound. The most encouraging result is the fully automatically computed precise linear bound for the `mergeInner` method [4]. While APROVE was able to process the example, it was still not able to infer the precise interplay between the traversals of the involved SLL partitions and numeric values needed to compute the precise linear bound.

Of course, our path-based norms do have their limitations too. They are, e.g., not sufficient to verify algorithms like the Deutsch-Schorr-Waite tree traversal algorithm or tree destruction algorithms, which could probably be verified using size-based norms, based on counting all memory locations reachable from a given location. We thus see an approach combining such norms (perhaps with suitably bounded scope) with our norms as an interesting direction of future research along with a better support of norms based on data stored in dynamic data structures.

References

1. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. J. Autom. Reasoning **59**(1) (2017) 3–45
2. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: Proc. of POPL'09. (2009) 239–251
3. Hofmann, M., Rodriguez, D.: Automatic type inference for amortised heap-space analysis. In: Proc. of ESOP'13. Number 7792 in LNCS, Springer (2013)
4. Atkey, R.: Amortised resource analysis with separation logic. Logical Methods in Computer Science **7**(2) (2011)
5. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Automatic Inference of Resource Consumption Bounds. In: Proc. of LPAR-18. Volume 7180 of LNCS., Springer
6. Frohn, F., Giesl, J.: Complexity analysis for java with aprove. In: Proc of IFM'17. (2017) 85–101
7. Holik, L., Hruska, M., Lengal, O., Rogalewicz, A., Simacek, J., Vojnar, T.: Forester: Shape analysis using tree automata (competition contribution). In: Proc. of TACAS'15. Volume 9035 of LNCS., Springer (2015)
8. Sinn, M., Zuleger, F.: Loopus - a tool for computing loop bounds for c programs. In: Proc. of WING@ETAPS/IJCAR. (2010)
9. Ströder, T., Aschermann, C., Frohn, F., Hensel, J., Giesl, J.: Aprove: Termination and memory safety of c programs (competition contribution). In: Proc. of TACAS'15. LNCS, Springer
10. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with Lists are Counter Automata. Formal Methods in System Design **38**("2) (2011)
11. Distefano, D., Berdine, J., Cook, B., O'Hearn, P.: Automatic Termination Proofs for Programs with Shape-shifting Heaps. In: Proc. of CAV'06. Volume 4144 of LNCS., Springer
12. Lahiri, S., Qadeer, S.: Verifying Properties of Well-Founded Linked Lists. In: Proc. of POPL'06, ACM Press (2006)
13. Yavuz-Kahveci, T., Bultan, T.: Automated verification of concurrent linked lists with counters. In: Proc. of SAS'02. Volume 2477 of LNCS., Springer 69–84
14. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.: Variance analyses from invariance analyses. In: Proc. of POPL'07, ACM
15. Rugina, R.: Shape analysis quantitative shape analysis. In: Proc. of SAS'04. Volume 3148 of LNCS., Springer (2004)
16. Habermehl, P., Iosif, R., Rogalewicz, A., Vojnar, T.: Proving Termination of Tree Manipulating Programs. In: Proc. of ATVA'07. Volume 4762 of LNCS., Springer
17. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: Proc. of POPL'10
18. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Thor: A tool for reasoning about shape and arithmetic. In: Proc. of CAV'08. Volume 5123 of LNCS., Springer
19. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. (2014) 745–761
20. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract Regular Model Checking. In: Proc. of CAV'04. Volume 3114 of LNCS., Springer (2004)
21. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: Proc. of TACAS'13, Springer
22. Holik, L., Hruska, M., Lengal, O., Rogalewicz, A., Vojnar, T.: Counterexample Validation and Interpolation-Based Refinement for Forest Automata. In: Proc. of VMCAI'17, Springer
23. Holik, L., Simacek, J., Rogalewicz, A., Vojnar, T.: Fully Automated Shape Analysis Based on Forest Automata. In: Proc. of CAV'13. Volume 8044 of LNCS., Springer (2013)
24. Bjørner, N.: The Z3 Theorem Prover
URL: `https://github.com/Z3Prover/z3/`.

A Overview of our approach on `mergeInner` method [4]

We discuss how our approach fully automatically infers the linear complexity of the `mergeInner` function in Fig. 5, the inner loop of an in-place linked-list merge sort implementation, taken from [4]).

```
1  Function void mergeInner(Node* list, int k)
2      Node* p = list;
3      Node* tail = NULL;
4      Node* list = NULL;
5      while $p \neq \texttt{NULL} \wedge k > 0$ do
6          Node* q = p;
7          int j = k;
8          while $j > 0 \wedge q \neq \texttt{NULL}$ do
9              q = q→next;
10             j−−;
11         Node* pstop = q;
12         qsize = k;
13         while
14         $(p \neq pstop) \vee (qsize > 0 \wedge q \neq \texttt{NULL})$ do
               Node* e;
15             if $(p == pstop)$ then
16                 e = q;
17                 q = q→next;
18                 qsize−−;
19             else if $(qsize == 0 \vee q == \texttt{NULL})$ then
20                 e = p;
21                 p = p→next;
22             else if $(p \rightarrow data \leq q \rightarrow data)$ then
23                 e = p;
24                 p = p→next;
25             else
26                 e = q;
27                 q = q→next;
28                 qsize−−;
29             if $(tail \neq \texttt{NULL})$ then
30                 tail→next = e;
31             else
32                 list = e;
33             tail = e;
34         p = q;
35     if $(tail == \texttt{NULL})$ then
36         return;
37     else
38         tail→next = NULL;
```

Fig. 5

We first shortly describe the function and discuss the challenges it poses to an automatic analysis. The outer loop at *line 5* iterates over the single linked list $list$ in steps of size $2k$. The inner loop at *line 8* progresses k steps through the `next`-selector from pointer q. At *line 11* we remember the original position of q in variable $pstop$ and so we have prepared two partitions of list $list$: (1) the partition of size k between variables p and $pstop = q$, we refer to this partition as p-chunk, (2) a partition which starts at q, we refer to this partition as q-chunk.

The second inner loop at *line 13* implements the ordered (ascending order) merging of the elements of both chunks. It compares the current element of the p-chunk with the current element of the q-chunk. In *line 26* an element `p→data` of chunk p is swapped with an element `q→data` of chunk q, if `p→data` is greater than `q→data`. Assuming that both chunks were already sorted, their merged chunk of size 2k will be sorted when reaching the *line 34*, which concludes one iteration of the outer loop.

The resulting list of merged chunks of size $2k$ is constructed using the variables $tail$ (pointer to the tail of the list), e (pointer to current front element of either p-chunk or q-chunk) and $list$ (pointer to the head of the list) on $lines$ $28 - 32$. At *line 36* we `NULL` terminate the list of merged chunks. At this point, n/k chunks of length k are merged.

Given that the outer loop iterates over the list in steps of size $2k$ while the first inner loop partitions the list into k-chunks and the second inner loop iterates over these chunks, the overall complexity of the example is linear in the size of the $list$. The example implements merge sort if we assume that `mergeInner` is consecutively called with $k = 1, 2, 4, \ldots, \frac{n}{2}$, where n is the size of $list$. `mergeInner` is thus called $\log_2(n) - 1$ times, and we obtain the well-known merge sort complexity of $n \log(n)$.

This example poses two main challenges to an automated bound analysis since (1) it requires precise shape analysis to track the interplay between the list chunks pointed to by variables p and q, as well as handle the in-place modification of the traversed list and (2) given an input single linked list of size n, in the worst-case, k may be n and the inner loops at *line 8* and *line 13* may thus iterate n-times on a single execution of the outer loop, which itself can be iterated n-times for the case $k = 1$, nevertheless, the overall complexity is $\mathcal{O}(n)$.

The tools APROVE and COSTA could not infer the linear complexity: COSTA times out after 5 minutes of computation and APROVE infers a quadratic complexity for the example. In the following we describe how our approach infers the linear complexity of the `mergeInner` function, as discussed in the paper:

Deriving of suitable norms. We derive the following norms: For the outer loop at *line 5*, we derive the single norm $p\langle \text{next}^* \rangle$NULL, for the first inner loop at *line 8*, we derive the norm $q\langle \text{next}^* \rangle$NULL and for the last inner loop at *line 13*, we derive the norms $q\langle \text{next}^* \rangle$NULL, $p\langle \text{next}^* \rangle pstop$, $pstop\langle \text{next}^* \rangle p$

Generation of arithmetic program. We generate a numerical program using the initial set of tracked norms $\mathcal{N}_c$: During generation of the numerical program, the initial set of norm candidates $\mathcal{N}_c$ is extended: As an example, consider the instruction $pstop = q$ in line 11. To model this instruction we add the norm $p\langle \text{next}^* \rangle q$, derived from the norm $p\langle \text{next}^* \rangle pstop \in \mathcal{N}_c$ to $\mathcal{N}_c$, which is then tracked as well. The resulting numerical instructions are shown in Alg. 3, highlighted in red. We include the original program statements for better understanding, our approach runs the bound analyser only on the integer program which results from removing all pointer instructions from Alg. 3.

Bounds Analysis. The numerical program is analysed by the underlying bounds analyser (Loopus), which computes the bounds on program loops. Due to the nature of numerical program construction, an upper bound on the numerical program is also an upper bound on the original program. We give a brief sketch on how Loopus infers a bound for our example (we refer the reader to [1] for details on the used algorithm).

For the main loop at *line 5* we can use the norm $p\langle \text{next}^* \rangle$NULL as the bound. The norm is initialized to the value of norm $list\langle \text{next}^* \rangle$NULL and never increases (despite the swapping operations between p and q on $lines$ 6 and 34) and it is decremented at least once at every iteration — since we assume that $k > 0$, then either in the inner loop at *line 8* the instruction at *line 9* is executed at least once (and hence $p\langle \text{next}^* \rangle$NULL is decremented after execution of *line 34*) or the loop is not executed at all which implies that $q\langle \text{next}^* \rangle$NULL $== 0$ and hence $p\langle \text{next}^* \rangle$NULL $== 0$ will hold at *line 34*. Hence, the outer loop has linear complexity in size of the $list\langle \text{next}^* \rangle$NULL. Now a similar reasoning can be applied for the loop at *line 8*, for which we can use the norm $q\langle \text{next}^* \rangle$NULL as the bound. $q\langle \text{next}^* \rangle$NULL is initialized to the size of $p\langle \text{next}^* \rangle$NULL, hence the bound is linear in size of $list\langle \text{next}^* \rangle$NULL as well.

However, for the inner loop at *line 13*, we have to use the combination of norms $p\langle \text{next}^* \rangle pstop$ and $q\langle \text{next}^* \rangle$NULL in order to compute bounds. The size of $q\langle \text{next}^* \rangle$NULL is bounded by the size of $list\langle \text{next}^* \rangle$NULL and norm $p\langle \text{next}^* \rangle pstop$ is always set to 0 at *line 6* and is incremented on each execution of the instruction at *line 9*. This increment can be executed in maximum the same number of times as the loop at *line 8*. Thus, the number of increments by one of $p\langle \text{next}^* \rangle pstop$ can be bounded by the size of $list\langle\text{next}^*\rangle\texttt{NULL}$ and the overall bound on $p\langle\text{next}^*\rangle pstop$ is $0 + list\langle\text{next}^*\rangle\texttt{NULL} \cdot 1$, i.e. it is bounded by $list\langle\text{next}^*\rangle\texttt{NULL}$ as well. Thus, in worst case the loop at *line 13* is executed $(p\langle\text{next}^*\rangle pstop + q\langle\text{next}^*\rangle\texttt{NULL})$-times, hence the bound of the loop is $2 \cdot list\langle\text{next}^*\rangle\texttt{NULL}$. Since, all the loops are linearly bounded in the length of the list $list$, the overall complexity of `mergeInner` is linear in the length of $list$.

```
1  Function void mergeInner(Node* list, int k)
2      Node* p = list;
       $p\langle\text{next}^*\rangle\texttt{NULL} = list\langle\text{next}^*\rangle\texttt{NULL}$;
3      Node* tail = NULL;
4      Node* list = NULL;
5      while $p\langle\textit{next}^*\rangle\texttt{NULL} \neq 0 \wedge p \neq \texttt{NULL} \wedge k > 0$ do
6          Node* q = p;
           $q\langle\text{next}^*\rangle\texttt{NULL} = p\langle\text{next}^*\rangle\texttt{NULL}$;
           $p\langle\text{next}^*\rangle q = 0$;
7          int j = k;
8          while $j > 0 \wedge\ q\langle\textit{next}^*\rangle\texttt{NULL} \neq 0 \wedge q \neq \texttt{NULL}$ do
9              q = q→next;
               $q\langle\text{next}^*\rangle\texttt{NULL} - -$;
               $p\langle\text{next}^*\rangle q + +$;
10             j−−;
11         Node* pstop = q;
           $p\langle\text{next}^*\rangle pstop = p\langle\text{next}^*\rangle q$;
12         qsize = k;
13         while $(p\langle\textit{next}^*\rangle pstop \neq 0 \wedge p \neq pstop) \vee \big(qsize > 0 \wedge q\langle\textit{next}^*\rangle\texttt{NULL} \neq 0 \wedge q \neq \texttt{NULL}\big)$ do
14             Node* e;
15             if $(p\langle\textit{next}^*\rangle pstop = 0 \vee p == pstop)$ then
16                 e = q;
17                 q = q→next;
                   $q\langle\text{next}^*\rangle\texttt{NULL} - -$;
                   $p\langle\text{next}^*\rangle q + +$;
18                 qsize−−;
19             else if $(qsize == 0 \vee q\langle\textit{next}^*\rangle\texttt{NULL} = 0 \vee q == \texttt{NULL})$ then
20                 e = p;
21                 p = p→next;
                   $p\langle\text{next}^*\rangle\texttt{NULL} - -$;
                   $p\langle\text{next}^*\rangle pstop - -$;
                   $p\langle\text{next}^*\rangle q - -$;
22             else if $(*)$ then
23                 e = p;
24                 p = p→next;
                   $p\langle\text{next}^*\rangle\texttt{NULL} - -$;
                   $p\langle\text{next}^*\rangle pstop - -$;
                   $p\langle\text{next}^*\rangle q - -$;
25             else
26                 e = q;
27                 q = q→next;
                   $q\langle\text{next}^*\rangle\texttt{NULL} - -$;
                   $p\langle\text{next}^*\rangle q + +$;
28                 qsize−−;
29             if $(tail \neq \texttt{NULL})$ then
30                 tail→next = e;
31             else
32                 list = e;
33             tail = e;
34         p = q;
           $p\langle\text{next}^*\rangle\texttt{NULL} = q\langle\text{next}^*\rangle\texttt{NULL}$;
           $p\langle\text{next}^*\rangle pstop = q\langle\text{next}^*\rangle\texttt{NULL}$;
           $p\langle\text{next}^*\rangle q = 0$;
35     if $(tail == \texttt{NULL})$ then
36         return;
37     else
38         tail→next = NULL;
```

Algorithm 3: Numerical abstraction of the showcase example from Algorithm 5.