Arithmetizing Shape Analysis

Sebastian Wolff[1⋆], Ekanshdeep Gupta[1], Zafer Esen[2], Hossein Hojjat[3], Philipp Rümmer[4,2], and Thomas Wies[1]

[1] New York University, USA
[2] Uppsala University, Sweden
[3] TeIAS, Khatam University, Iran
[4] University of Regensburg, Germany

Abstract Memory safety is a fundamental correctness property of software. For programs that manipulate linked, heap-allocated data structures, ensuring memory safety requires analyzing their possible shapes. Despite significant advances in shape analysis, existing techniques rely on handcrafted domains tailored to specific data structures, making them difficult to generalize and extend. This paper presents a novel approach that reduces memory-safety proofs to the verification of heap-less imperative programs, enabling the use of off-the-shelf software verification tools. We achieve this reduction through two complementary program instrumentation techniques: space invariants, which enable symbolic reasoning about unbounded heaps, and flow abstraction, which encodes global heap properties as local flow equations. The approach effectively verifies memory safety across a broad range of programs, including concurrent lists and trees that lie beyond the reach of existing shape analysis tools.

1 Introduction

One of the most severe and common types of flaws in software systems are memory safety violations. Memory safety violations typically happen in unsafe languages such as C and C++, for instance when the program tries to use a pointer to a memory location that has already been freed or that is out of bounds.

In this paper, we focus on *automatic* methods to prove the memory safety of programs operating on *linked mutable data structures*. We explicitly ignore issues related to the use of low-level patterns such as pointer arithmetic, union types, and casts as these are of orthogonal concern.

The key difficulty to verifying memory safety in this context is to determine the expected *shape* of the data structures, a challenge that has led to the field of *shape analysis* [37,13] and a plethora of different methods; for instance, based on three-valued logic [59], automata techniques [31], separation logic [16,5,15,17], bi-abduction [12,33], and other tailor-made abstract domains [57,14,2,10,18,34,62,22]. Today, the best tools competing in the *MemSafety* category of the software verification competition SV-COMP are based on an intricate combination of shape analysis techniques, including abstract interpretation, symbolic execution, and model checking.

⋆ This work was completed prior to the author's employment at Amazon.

The mentioned approaches have in common that they generally need to be carefully tuned for a particular class of linked data structures to obtain good performance. As a result, practical shape analyses often make trade-offs such as targeting only specific data structures, e.g., linked lists. The engineering effort involved in developing and maintaining these sophisticated analyses makes an extension to other classes of data-structures (e.g., to trees), support for concurrency in programs [49,64], or an integration with techniques that target properties unrelated to memory safety (e.g., reasoning about data [10]) highly non-trivial. At SV-COMP, shape analysis-based tools largely form a class of their own, whereas software model checkers that aim at general safety properties provide only very limited support for reasoning about pointers. State-of-the-art verifiers, even though they might perform well on general verification tasks, can fail even on simple memory safety benchmarks involving singly-linked lists.

The goal of this paper is to provide a new avenue for adding shape reasoning to general-purpose verification tools, bypassing the need to develop and integrate sophisticated shape analysis domains. To this end, we propose a reduction-based approach that can be implemented in a preprocessing step for static analyzers targeting heap-free programs. The reduction builds on recent advances on local reasoning techniques for heap-manipulating programs. Specifically, we combine ideas from the *flow framework* [42,43,50], an approach based on separation logic for node-local reasoning about inductive properties of general heap graphs, with *space invariants* [38], which can summarize heap properties using node-local invariants. We formalize the approach in terms of rewrite rules that translate heap-manipulating programs to heap-less programs, and thus effectively *arithmetize* shape analysis for memory safety verification.

Compared to bespoke shape analysis methods, our reduction-based approach has several advantages. (i) Our approach is not tailored to any specified class of data-structures, but can handle, e.g., lists, nested lists, and trees out of the box. By plugging in different flow domains, the precision of the analysis can be controlled and increased on demand. (ii) Since the final analysis is carried out by an off-the-shelf verification tool, our approach inherits all capabilities for reasoning about data from the back-end tool; similarly, the approach is agnostic of the control structure (e.g., recursion) present in programs. (iii) Our approach is easy to extend to concurrent programs, which can be analyzed in a thread-modular way. We show that the reduction method can handle yield points and locks, and thus analyze challenging concurrent programs fully automatically. (iv) Our approach is simple to implement, since the required symbolic reasoning is offloaded to the off-the-shelf verification tool used as back-end.

For evaluation, we have implemented our approach in a prototype verification tool TRICERATOPS, utilizing the off-the-shelf software model checkers SEAHORN [23] and TRICERA [20] as back-ends, and evaluate using a set of benchmarks from the SV-COMP [7], as well as implementations of standard data structures written by us. The benchmarks include both sequential and concurrent programs and cover a variety of different shapes, including singly-linked lists, doubly-linked lists, and trees. We find that TRICERATOPS is able to verify

```
1  Node* List = new_dummy();
2  void pop() {
3    lock(List);
4    Node* item = List->next;
5    if (item != NULL) {
6      lock(item);
7      List->next = item->next;
8      unlock(item);
9      free(item); }
10   unlock(List);
11 }
```

```
12 void insert() {
13   Node* curr = List; lock(curr);
14   while (havoc && curr->next != NULL) {
15     Node* next = curr->next;
16     lock(next); unlock(curr);
17     curr = next; }
18   Node* item = malloc;
19   item->next = curr->next;
20   curr->next = item;
21   unlock(curr);
22 }
```

Figure 1. A `NULL`-terminated singly-linked list implementation that pops elements from the front and inserts elements at a non-deterministically chosen position. Adding the colored lines turns the sequential implementation into a concurrent one that supports arbitrarily many concurrent pops and inserts.

memory safety effectively on such a diverse range of problems. In comparison with PREDATOR-HP [56,30], the 2024 SV-COMP gold medalist in the memory safety category [7], we observe that TRICERATOPS tends to exhibit longer runtimes, but is able to cover a wider range of problems than the more specialized tool PREDATOR-HP.

The *main contributions* of our paper are (i) a new reduction-based approach to shape analysis that combines space invariants (§3) with flow reasoning (§4) and extends to concurrent programs (§5); (ii) an *implementation* of our approach, resulting in the TRICERATOPS tool; and (iii) an *empirical evaluation* of the approach using a diverse set of sequential and concurrent programs (§6). The paper presents the overarching ideas of our new shape analysis and takes an operational view that lends itself to an implementation via program transformation. For a formalization of the analysis, we direct the reader to [68].

2 Motivating Example

We address the challenge of proving *memory safety* for heap-manipulating programs. Our goal is to fully automate the verification of the following properties:

- **(M1)** Absence of unsafe accesses: heap reads and writes happen only through valid pointers, i.e., pointers that reference memory addresses that are allocated and have not been freed.
- **(M2)** Absence of double frees: no memory is deallocated more than once.
- **(M3)** Absence of memory leaks: all allocated memory is eventually deallocated.

Ensuring memory safety is a foundational aspect of program correctness but is often challenging due to the inherent complexity of understanding and capturing the shape of heap-allocated structures.

Running Example. We use the singly-linked list implementation from Fig. 1 as the running example (ignore the colored lines for a moment). The implementation includes a shared pointer `List` that references a dummy node whose successor is the head of the list. Initially, the list is empty and `List`'s next field is `NULL`. This initialization is performed by `new_dummy()` on Line 1, which we elide.

New elements are added using the `insert` function. The insertion location is denoted by pointer `curr`, which is chosen by traversing the list for a non-deterministic number of steps, Lines 14–17. A new node `item` is allocated using `malloc` and its `next` field is set to `curr`'s current successor, `curr->next`, on Line 19. Then, the new `item` is inserted after `curr` on Line 20 by updating `curr`'s `next` field to `item`.

The `pop` function removes the head of the list. It begins by reading the current head into the pointer `item`, Line 4. If `item` is non-null, its successor becomes the new head, Line 7, and `item` is deallocated using `free`, Line 9.

Figure 2 illustrates a possible heap graph after inserting three nodes and popping one. Concretely, node a_1 represents the dummy node referenced by `List`. Nodes a_2, a_3, and a_4 have been inserted into the list in that order. Node a_2 has been unlinked (Line 7) and freed (Line 9) while pointer `item` from `pop` is still referencing it.

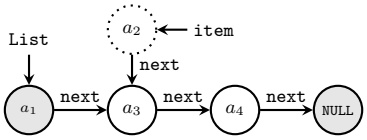

Figure 2. A possible heap resulting from three `insert`s and one `pop`.

We are working in the regime of whole-program analysis. When we refer to the *program* in Fig. 1 below, we mean the most general client of the given functions.

Although simple, verifying the program automatically turns out to be surprisingly challenging. To establish memory safety, we have to identify the following invariant that the program maintains: the allocated heap objects that have not yet been deallocated are precisely those reachable from `List` by traversing `next` links. Deriving this invariant automatically is non-trivial, as it requires global reasoning about the structure of the heap graph. While existing shape analysis techniques can derive such invariants, they rely on carefully designed shape domains to achieve efficiency and are typically precise only for specific shapes they are tailored to. Additionally, these techniques are often difficult to combine with other static analysis methods, such as those needed to reason about data [10,22]. Consequently, many software model checkers and automatic verification tools are unable to verify memory safety of our running example.

Our Approach. We present a source-to-source code transformation (*instrumentation*) that carries out the required shape analysis in a *completely local* way, making it both easy to automate and easy to integrate with other automatic verification methods. After instrumentation, we obtain a heap-less program that can be verified using off-the-shelf verification tools, such as the software model checker SEAHORN [23]. To achieve heap-lessness, our approach is based on two main reasoning principles: *space invariants* [38] and *flow abstraction* [42,43]. While both space invariants and flow abstraction have been developed in previous works, we are the first to combine them in order to obtain a fully automatic approach for

verifying memory safety of programs like the one in Fig. 1. We stress that neither of the two approaches, by itself, is able to (automatically) verify such programs.

Space invariants over-approximate the heap: they capture all possible values that the fields of the object at a given address may have. This means that space invariants are per-object invariants. Materializing them for the addresses referenced by the pointers in the program allows us to mirror in the stack the (finitely many) heap locations that are accessible at any given program location. Hence, reads and writes targeting the heap can be mimicked by equivalent reads and writes targeting the stack. Since space invariants can be encoded using uninterpreted predicates [20,66], they can be automatically inferred by verification tools.

Flow abstraction, intuitively, performs a data-flow analysis on the heap graph. That is, the *flow* at a node is computed by propagating some initial flow value from a set of root nodes along the links in the heap. Hence, the flow captures global properties of the heap graph in a node-local way. We can use this abstraction, for instance, to capture the number of paths starting in a set of roots that a given node lies on. Flows like this path count can reveal shape information such as reachability (path count ≥ 1) and tree-ness (path count at most 1 at all nodes). To access the flow in our instrumentation, we treat it as ghost field flow on every object that is updated whenever there are writes to the heap.

Using the above abstractions, our instrumentation is capable of checking for memory safety. For **(M1)**, we add assertions prior to dereferencing a pointer `x` to ensure that `x` is neither `NULL` nor freed. For the latter, we extend all objects with a special free flag that is raised by `free`. That is, `x` is not freed if `x->free` is not raised. For **(M2)**, we do the same prior to freeing a pointer `x`. For **(M3)**, we rely on the path counting flow and let our instrumentation assert that the space invariant is strong enough to imply that every object is either reachable or has been freed.

Figure 3 gives the instrumentation of function `pop` from Fig. 1. The accessible heap locations are mirrored using stack variables, one for each field of each object. For example, field `next` of pointer `List` is mirrored in `List_next`, and similarly for all other pointers. Dereferences like `List->next` on Line 4 in Fig. 1 are then replaced with the corresponding stack variable `List_next` on Line 26. The memory safety **(M1)** of this dereference is ensured by the assertion on Line 25. Similarly, an instance for checking **(M2)** is on Line 39. Checking **(M3)** is wrapped within function `sync` on Line 43, detailed in Fig. 6. In the following sections we elaborate on the ingredients of our instrumentation and discuss how to obtain the one in Fig. 3.

3 Core Instrumentation

We present an idealized version of our code instrumentation for rewriting the input program into a new program that does not use dynamically allocated heap memory. Our instrumentation conservatively overapproximates the behavior of the original program by overapproximating the actual values that it may read from or write to the heap. Later, we incorporate orthogonal techniques for the instrumentation to handle flow reasoning (§4) and concurrency (§5).

```
23 void pop() {
24   // Line 4: Node* item = List->next;
25   assert(List != 0 && List_free == 0);   avail(List)
26   int item = List_next;                                  Fmaterialize(item)
27   int item_next = havoc, item_free = havoc, item_flow = havoc;
28   assume(Inv(item, item_next, item_free, item_flow));    materialize(item)
29   int item_0_next = item_next, item_0_flow = item_flow;
30   assume(item_flow >= 1);   Fprop(List,item)
31   // Line 5: if (item != NULL) { ... }
32   if (item != 0) {
33     // Line 7: List->next = item->next;
34     assert(List != 0 && List_free == 0);   avail(List)
35     assert(item != 0 && item_free == 0);   avail(item)
36     List_next = item_next;
37     if (List == item) { item_next = List_next; }   dyn_up(List,next)
38     // Line 9: free(item);
39     assert(item != 0 && item_free == 0);   avail(item)
40     item_free = 1;
41     if (item == List) { List_free = 1; }   dyn_up(List,next)
42     // combined push site for Lines 7 and 9   Fpush(List), Fpush(item)
43     sync(List, item);
44     assert(Inv(List, List_next, List_free, List_flow));   push(List),
45     assert(Inv(item, item_next, item_free, item_flow));   push(item)
46     List_0_next = List_next; List_0_flow = List_flow;
47     item_0_next = item_next; item_0_flow = item_flow;
48 } }
```

Figure 3. Instrumentation of function `pop` from Fig. 1. The instrumentation is heap-less, mirroring the accessed heap locations in the stack by materializing the space invariant Inv. Moreover, the instrumentation ensures memory safety by inserting appropriate assertions. Helper `sync` on Line 43 performs flow reasoning and is discussed in detail in §4.

3.1 Overview

The instrumentation closely follows the structure of the original program, retaining control structures such as loops and branching. Primitive commands and expressions are rewritten to *not* use the heap. To make this precise, we assume that programs adhere to the following EBNF:

$$
\begin{aligned}
st &::= \texttt{while (havoc) \{ } st \texttt{ \}} \mid \texttt{if (havoc) \{ } st \texttt{ \} else \{ } st \texttt{ \}} \mid st^* \mid com\texttt{;} \\
com &::= \texttt{int x} \mid \texttt{T* x} \mid \texttt{x = y} \mid \texttt{x = y->f} \mid \texttt{x->f = y} \mid \texttt{x = malloc} \\
&\quad \mid \texttt{free(x)} \mid \texttt{assert(x} \oplus \texttt{y)} \mid \texttt{assume(x} \oplus \texttt{y)}
\end{aligned}
$$

Both `while` and `if` statements are non-deterministic, as indicated by their conditions being `havoc`. This way, only primitive commands require rewriting. The

Rule	Input *com*	Instrumentation *com* $\rightsquigarrow \ldots$
Intro	`T* x;`	`int x; int x_f`$_1$`, ..., x_f`$_n$`, x_free, x_flow;`
Intro2	`int x;`	`int x;`
Assign	`x = y;`	`x = y; x_f`$_1$ `= y_f`$_1$`; ...; x_f`$_n$ `= y_f`$_n$`;` `x_free = y_free; x_flow = y_flow;`
Read	`x = y->f;`	`avail(y); x = y_f; materialize(x);`
Write	`x->f = y;`	`avail(x); x_f = y; dyn_up(x, f); push(x);`
Free	`free(x);`	`avail(x); x_free = 1; dyn_up(x, f); push(x);`
Malloc	`x = malloc;`	`x = ++ALLOC; x_f`$_1$ `= havoc; ...; x_f`$_n$ `= havoc;` `x_free = 0; x_flow = 0; no_alias(x); push(x);`
Assume	`assume(x` $\oplus$ `y);`	`prf(`$\oplus$`, x, y); assume(x` $\oplus$ `y);`
Assert	`assert(x` $\oplus$ `y);`	`prf(`$\oplus$`, x, y); assert(x` $\oplus$ `y);`

```
avail(x)       ≡ assert(x != 0 && x_free == 0);
materialize(x) ≡ if (x is not a pointer) { /* no-op */ } else {
    x_f_1 = havoc; ...; x_f_n = havoc; x_free = havoc; x_flow = havoc;
    assume(Inv(x, x_f_1, ..., x_f_n, x_free, x_flow)); }
push(x)        ≡ assert(Inv(x, x_f_1, ..., x_f_n, x_free,  x_flow));
dyn_up(x, f)   ≡ for i in 1..k: if (x == y_i) { y_i_f = x_f; }
no_alias(x)    ≡ for i in 1..k: assume(x != y_i && ⋀_{j=1}^{m} x != y_i_ptr_j);
prf(⊕, x, y)   ≡ if (⊕ is other than ==) { /* no-op */ } else {
    assert(x == 0 || y_free == 0); assert(y == 0 || x_free == 0); }
```

Figure 4. Instrumentation rules (top) and abbreviations (bottom). For a streamlined presentation, we assume that pointers are of type `T*` and have fields $\mathtt{f}_1, \ldots, \mathtt{f}_n$, of which $\mathtt{ptr}_1, \ldots, \mathtt{ptr}_m$ are those that are pointers themselves. We also assume that $\mathtt{y}_1, \ldots, \mathtt{y}_k$ are all pointers in scope other than `x`. We use `for` loops to syntactically repeat code.

primitive commands include variable declarations for integers and pointers `T`, variable assignments, reads from the heap, writes to the heap, heap memory allocation and deallocation, as well as assertions and assumptions that compare variables using an operator $\oplus$. Note that assumptions enable this language to model standard conditional loops and branching.

Our instrumentation is a relation $st_o \rightsquigarrow st_i$ among statements, where st_o is from the original program and st_i is the instrumented version of st_o. As mentioned earlier, statements themselves need no rewrite, only the primitive commands do. An overview of the rewriting rules is given in Fig. 4 and further detailed in the remainder of this section.

3.2 Heap Abstraction

Our instrumentation abstracts from the actual heap by overapproximating it with a *space invariant*. The space invariant is a family of uninterpreted predicates $\mathsf{Inv}_T(a, f_1, \ldots, f_n, \mathsf{free}, \mathsf{flow})$, one for each pointer type T in the program, indicating whether or not the fields of the object at address a may have the given

values. The invariant takes all fields $f_1, \ldots, f_n$ of type T as well as the special fields free and flow.

A crucial aspect of our approach is the fact that the space invariant is uninterpreted. This allows us to pose requirements to it without explicitly specifying it, relying on the back-end solver to synthesize an appropriate invariant when verifying our instrumented program. While uninterpreted predicates are no standard feature, they are supported by tools like SEAHORN [23,66] and TRICERA [20].

3.3 Reading from the Heap

The original program accesses the heap with dereferences, such as `x->f` to access field `f` of the address a that `x` points to. These accesses are replaced to refer to the stack. To that end, our instrumentation mirrors the heap portion referenced by `x` by introducing a program variable `x_f`, for each field `f` of `x`. This is implemented by rule INTRO. The dereference `x->f` is then simply replaced with `x_f`.

To ensure the memory safety of dereferences **(M1)**, our instrumentation inserts assertions prior to every dereference `x->f` that ensure that `x` is neither `NULL` nor freed. This check is implemented by `avail(x)` from Fig. 4.

For the newly introduced stack variables `x_f` to appropriately mirror the actual heap portions they correspond to, we materialize their values from the space invariant. Hence, we refer to those variables as materialization variables. Concretely, when a pointer `x` receives a new address (is assigned to) from a heap read, rule READ non-deterministically chooses new values for all materialization variables associated with `x` and then constrains them to satisfy the space invariant, implemented by `materialize(x)` from Fig. 4. This guarantees that subsequent accesses to the fields of `x`, including the special fields flow and free, overapproximate the values that the actual heap may hold.

Example 1. Consider the assignment `Node* item = List->next;` on Line 4 of our running example from Fig. 1. The instrumentation of this assignment is on Lines 25–28 from Fig. 3 and proceeds as follows. First, Line 25 ensures that dereferencing `List` is safe (`avail(List)`). Second, Line 26 updates `item` to its new value as read from `List`'s materialization variable `List_next`. Last, Line 27 havocs the materialization variables for `item` and Line 28 materializes them from the space invariant Inv (`materialize(List)`).

Note that Line 27 deviates from `materialize(List)` in that it declares the materialization variables for `item`. This is because our running example does not separate the declaration of `item` from its assignment, which is why we combine rules INTRO and READ. □

3.4 Writing to the Heap

In the instrumentation, updates to the heap are reflected by updates to the materialization variables, rule WRITE. As expected, we replace writes to the heap, say to `x->f`, with writes to the corresponding materialization variables, `x_f`. Since we store materialization variables for every pointer, simply updating `x_f` may

result in an inconsistent state where the materialization variable `y_f` of an alias `y` of `x` does not reflect the update. To overcome this, we perform a *semantic alias analysis* and mirror the update on aliasing pointer's materialization variables. The alias analysis is semantic in the sense that we encode it into the instrumentation such that it is performed by the back-end solver. Concretely, we use `dyn_up(x,f)` from Fig. 4 which adds conditionals that, for every pointer `y`, check if it aliases `x` and, if so, update `y_f` to `x_f`.[5]

Besides updating the materialization variables, our instrumentation has to reconcile heap updates with the space invariant, ensuring that it conservatively overapproximates all possible heap values. To that end, we simply assert the space invariant for the updated pointer and its materialization variables, cf. `push(x)` from Fig. 4. We refer to the program locations where this happens as *push sites*.

In practice, it is beneficial not to push each update on its own, but collect multiple updates and push them together. Coarse-grained pushes lead to more *sensible* space invariants as they avoid capturing intermediate states.

Technically, we require a push site for an update to `x->f` after updating the corresponding materialization variable `x_f` and before `x_f` becomes inaccessible. Materialization variable `x_f` may become inaccessible if it goes out of scope, i.e., if `x` goes out of scope, or if `x` is assigned to, i.e., `x_f` represents a potentially different memory portion. Due to space constraints, we do not detail this optimization.

Example 2. Consider the heap update `List->next = item->next;` on Line 7. The instrumentation in Fig. 3 proceeds as follows. First, Lines 34 and 35 ensure that it is safe to dereference pointers `List` (`avail(List)`) and `item` (`avail(item)`), respectively. Second, Line 36 updates the materialization variable `List_next` corresponding to `List->next` in order to reflect its new value, `item_next`. Third, Line 37 mirrors the update on aliases of `List` (`dyn_up(List, next)`). In the example, `item` is the only potential aliasing pointer in scope. Last, Line 44 asserts the invariant for `List` with its updated field values (`push(List)`).

In Fig. 3, the push site for `List` is deferred until the subsequent free of `item` from Line 9, the instrumentation of which is detailed below. Combining the push sites for these two commands alleviates the space invariant from capturing the intermediate state where `item` has been unlinked but not yet freed, improving the precision of our analysis. □

3.5 Frees

The instrumentation of memory reclamation `free(x)` by rule Free is straightforward: we treat it as if it was an ordinary heap update to the special field free, i.e.,

[5] The attentive reader readily realizes that reading from the heap may result in pointers being aliases of the same object, which our materialization does not detect. Since this does not affect soundness, we omit an improved Read rule. Extending Read with a semantic alias analysis such as the one discussed here follows naturally.

we treat it as `x->free = 1`. Consequently, the instrumentation of frees ensures that `x` is neither `NULL` nor freed. This, in turn, ensures that `x` is not freed twice **(M2)**.[6]

It is worth pointing out that in order to check for double frees, the instrumentation does not need to consider the free field of pointers `y` that alias `x`. This is because heap updates to `y` keep the materialization variables of `x` consistent, i.e., freeing `y` sets the free materialization variables associated with both `x` and `y`.

Example 3. The instrumentation in Fig. 3 of `free(item)` on Line 9 proceeds as expected: Line 39 ensures memory safety (`avail(item)`), Line 40 performs the deletion by setting the free flag to `1`, and Line 41 frees `List` should it alias `item`. The push site for the deletion is on Line 45. □

3.6 Allocation

Instrumenting allocations, rule Malloc, is similar to materializing from the space invariant: we assign a new value to the receiving pointer `x` and havoc its materialization variables, except for the special field free which is set to 0 (not freed). We do not assume the space invariant because allocations do not initialize fields but leave them unspecified. For the address that the allocation returns we use a global allocation counter `ALLOC` to produce a new address for every allocation. Moreover, we use `no_alias(x)` to add assumptions that guarantee that the returned address is distinct from all pointers and all materialization variables of pointer type in scope. This resembles a garbage-collected semantics and not a standard `C/C++` semantics where previously freed memory can be reused. For this to be sound, we require that the program cannot distinguish whether or not memory is actually reclaimed [25,32,51,52]. This requirement boils down to proving that the program satisfies memory safety properties **(M1)** and **(M2)** and, additionally, that pointers referencing freed memory are not compared to other pointers (except `NULL`). These assertions are implemented by the helper `prf` from Fig. 4 used in rules Assert and Assume. Note that this means that our instrumentation inlines an analysis akin to [25,32].

Similar to heap updates, pushing new allocations to the space invariant immediately will pollute the space invariant with states of uninitialized objects. In practice, we thus follow standard verification practice [63,65] and treat new allocations as *owned*, deferring their push until they are published into the shared heap so that the space invariant captures only *shared* objects. Due to space constraints and the fact that this optimization is not relevant for soundness, we refrain from making it explicit here.

Example 4. Consider the allocation of a `Node` into pointer `item` on Line 18. The instrumentation for `item = malloc` is given in Fig. 5. Line 58 assigns the next available address to `item` and increases the allocation counter `ALLOC`, which is declared along the shared pointer `List`. Line 59 havocs the fields of `item` and initializes the special fields free and flow. Lines 60 and 61 ensure that the allocation for `item` is fresh, i.e., distinct from `List`, `curr`, and any of their pointer fields. □

[6] This is stricter than necessary as it prevents freeing `NULL`, which is allowed and simply does nothing in `C/C++`. We ignore this technicality here.

```
58 int item = ++ALLOC;
59 int item_next = havoc, item_free = 0, item_flow = 0;
60 assume(item != List && item != List->next);
61 assume(item != curr && item != curr->next);
62 // assert(Inv(item, item_next, item_free, item_flow)); // push deferred
```

Figure 5. Instrumentation of `malloc`, Line 18. The allocation returns a *fresh* address. It is treated as owned memory and not pushed immediately.

4 Flows Instrumentation

We extend our instrumentation with flows. Flows provide light-weight shape information so that we can detect memory leaks, **(M3)**. Together with the assertions for **(M1)** and **(M2)** discussed in the previous section, this completes our memory safety analysis. Additionally, the shape information provided by flows increases the overall precision of our technique.

The changes required to integrate flow reasoning into our instrumentation are summarized in Fig. 6. We elaborate on the key components.

Flow Abstraction. The flow framework [42,43,50] associates a flow value with each node in the heap graph. Similar to forward data flow analyses, the flow values are computed by solving a fixed-point equation: each node is assigned an initial value and propagates updated values along its outgoing edges (pointers) based on its current flow value. This enables the flow framework to express inductive, global properties such as reachability in a local way, through the flow values of individual nodes.

To illustrate the reasoning carried out by our approach, we associate a *path count* with each node, a natural number that counts the number of paths starting in a root node and leading to the respective node on the heap. In terms of the running example from Fig. 2, we associate numbers $f_1, \ldots, f_4 \in \mathbb{N}$ with the nodes $a_1, \ldots, a_4$, respectively. The fixed-point equations for each node are derived by summing the path counts of the incoming `next` links:

$$f_1 = 1, \qquad f_2 = 0, \qquad f_3 = f_1 + f_2, \qquad f_4 = f_3.$$

The unique solution to these equations is $f_1 = f_3 = f_4 = 1$ and $f_2 = 0$, which enables us to differentiate between nodes that are still reachable (path count > 0) and nodes that are not part of the list (path count 0). Note that a_1 receives $f_1 = 1$ despite having no predecessors because it is the root of the list.

Towards detecting memory leaks **(M3)**, observe that the path-counting flow characterizes reachability: nodes with a path count of 0 are no longer reachable from the roots. Consequently, property **(M3)** is satisfied if all nodes with a path count of 0 have been reclaimed when the program under scrutiny terminates.

To perform this flow reasoning within our instrumentation, we extend the heap by adding a field flow to each object. In our example, field a_i->flow represents value f_i. The flow fields are updated dynamically by the instrumentation whenever heap objects or the heap graph change.

Localizing Flow Updates. The challenge in incorporating flows into the instrumentation arises because they are defined as fixed points over the entire heap graph, making a straightforward recomputation intractable.

We refer to the set of nodes whose flow values are affected by a heap update as the *footprint* of the update. A key observation that helps to address the issue of recomputing flow values is that the footprint is often localized to a small bounded region in the vicinity of the update. It then suffices to recompute the flow only in this bounded region, avoiding a recomputation of the full fixed point.

The key idea for this localization of the flow update is thus to (i) guess a bounded set of nodes (the footprint) for which the flow values are changed by the update, (ii) compute new flow values, but just within the footprint, (iii) verify that the update is really local to the chosen footprint.

Identifying a footprint that is guaranteed to localize the recomputation (i.e., step (i) above) is algorithmically challenging and oftentimes not possible to do statically. Instead, we use a heuristic for choosing the footprint: we include objects whose fields are updated by the program and neighboring objects with a statically fixed distance to the updated objects. In our implementation, we include only the immediate successors of updated objects.

We explain step (ii) below. Step (iii) guarantees the soundness of the localization to a given footprint. It requires our instrumentation to ensure that the flow at the boundary of the footprint is not changed by the update [42,43,50]. To be precise, for all objects `y` outside the footprint, we have to ensure that the total flow they receive from the footprint is exactly the same before and after the update. If so, the flow update is indeed local to the footprint and does not change outside of the footprint. Otherwise, verification fails.

Revisited Instrumentation. In order to compare the footprint before and after the update, we have to duplicate the materialization variables of all pointers before any update is performed. Our instrumentation creates such duplicates whenever a pointer is declared, rule FIntro from Fig. 6. For pointer `x` and field `f`, the duplicate is `x_0_f`. (We do not duplicate the free field, it is not needed.) Whenever `x` is materialized (`Fmaterialize`), the duplicates `x_0_f` are set to the corresponding original `x_f` to reflect that no updates have been performed so far.

The main change for integrating flows concerns pushing updates as part of rules Write and Free. Our revised instrumentation from Fig. 6 employs `Fpush` to do this. To present it in a way that does not rely on the actual flow that is being used, we use the following placeholders: $I(\mathtt{x})$ produces the initial flow value for object `x`, $E_{\mathsf{pre}}(\mathtt{x},\mathtt{y})$ produces the flow that `y` receives from `x` before the update (using the duplicates `x_0_f`), and $E_{\mathsf{post}}(\mathtt{x},\mathtt{y})$ produces the flow that `y` receives from `x` after the update (using the originals `x_f`). Note that E_{pre} and E_{post} produce the sum of all edges between the nodes, should there be multiple. For an instantiation of these placeholders to the path-counting flow, refer to Example 5 below.

After choosing some footprint $\mathtt{x}_1, \ldots, \mathtt{x}_k$, e.g., according to the above heuristic, the most interesting part of `Fpush` is `sync`. It computes the new flow in the footprint (step (ii) above) and checks that the updates did not change the flow at the boundary of the footprint (step (iii)). Intuitively, to compute the new flow, we

Rule	Input *com*	Flow-aware instrumentation *com* $\rightsquigarrow \ldots$
FIntro	`T* x;`	`int x; int x_f`$_1$`, ..., x_f`$_n$`, x_free, x_flow;` `int x_0_f`$_1$`, ..., x_0_f`$_n$`, x_0_flow;`

FAssign	`x = y;`	`x = y; x_f`$_1$` = x_f`$_1$`; ...; x_f`$_n$` = x_f`$_n$`;`
		`x_0_f`$_1$` = y_0_f`$_1$`; ...; x_0_f`$_n$` = y_0_f`$_n$`;`
		`x_free = y_free; x_flow = y_flow; x_0_flow = y_0_flow;`

FRead	`x = y->f;`	`avail(y); x = y_f; Fmaterialize(x); Fprop(y, x);`

FMalloc	`x = malloc;`	`x = ++ALLOC; x_f`$_1$` = havoc; ...; x_f`$_n$` = havoc;`
		`x_free = 0; x_flow = 0; no_alias(x);`
		`x_0_f`$_1$` = x_f`$_1$`; ...; x_0_f`$_n$` = x_f`$_n$`;`
		`x_0_flow = x_flow; push(x);`

```
63 Fprop(y, x) ≡ if (x is a pointer) { assume(x_flow >= E_post(y, x)); }
64 Fmaterialize(x) ≡ materialize(x); if (x is a pointer) {
65    x_0_f1 = x_f1; ...; x_0_fn = x_0_fn; x_0_flow = x_flow; }
66 Fpush(x) ≡ let x1, ..., xk = footprint of x; sync(x1, ..., xk);
67    for i in 1..k, j in 1..n: push(xi); xi_0_fj = xi_fj;
68 sync(x1, ..., xk) ≡
69    for i in 1..k: int xi_in = havoc; xi_flow = havoc;
70    for i in 1..k: assume(xi_0_flow == xi_in + I(xi) + ∑_{j=1}^{k} E_pre(xi,xj));
71    for i in 1..k: assume(xi_flow == xi_in + I(xi) + ∑_{j=1}^{k} E_post(xi,xj));
72    int out = havoc; assume(out > 0 && ⋀_{i=1}^{k} out != xi);
73    assert(∑_{i=1}^{k} E_pre(xi, out) == ∑_{1≤i≤k} E_post(xi, out));
```

Figure 6. Revision of the core instrumentation from Fig. 4 to support flow reasoning. Changes to the rules (top) are colored. The `F`-prefixed abbreviations (bottom) replace the ones from Fig. 4 in all remaining and otherwise unchanged rules. In `sync`, functions $E_{\mathsf{pre}}(\mathtt{x},\mathtt{y})$ and $E_{\mathsf{post}}(\mathtt{x},\mathtt{y})$ encode the flow-specific values that are sent from `x` to `y` before and after the update, respectively.

havoc the new flow values and constrain them to guarantee that they are a fixed point (lines 69-71). We explain the relevant steps in more detail.

Since we compute the flow fixed point only for the footprint, we have to account for the flow that the footprint receives from objects outside the footprint. To that end, `sync` introduces $\mathtt{x}_i\mathtt{\_in}$, for each $\mathtt{x}_i$, which reflects that flow $\mathtt{x}_i$ receives from the outside. The value of $\mathtt{x}_i\mathtt{\_in}$ is chosen non-deterministically on Line 69, and then constrained by the flow from before the update on Line 70. Intuitively, Line 70 performs one step of a Kleene fixed point iteration and assumes that the result equals the expected flow. Technically, the Kleene iteration for $\mathtt{x}_i$ sums up the initial value $I(\mathtt{x}_i)$, the flow $\mathtt{x}_i\mathtt{\_in}$ that $\mathtt{x}_i$ receives from outside the footprint, and the flow $\mathtt{x}_i$ receives from within the footprint (via E_{pre}).

Next, `sync` computes the flow after the update on Line 71. While we could explicitly implement a standard least fixed point computation to do so, we found it to be more efficient to non-deterministically choose new flow values and enforce on Line 71 that they are *a* fixed point, analogous to the Kleene iteration above.

Step (iii) is implemented on lines 72-73. Similar to the previous step, we found it to be more efficient to perform this final check for a non-deterministically chosen object outside the footprint, rather than iterating over all the links that the objects in the footprint have to the outside. Concretely, Line 72 first havocs a node outside the footprint (this is variable `out`). Then Line 73 asserts that the flow that this node receives from within the footprint has remained unchanged, i.e., the flow it received before the update is the same as after the update.

Example 5. Consider the updates `pop` performs, namely unlinking and freeing `item` on Lines 7 and 9, respectively. The instrumentation of `pop` in Fig. 3 implements the flow reasoning presented so far. The interesting part is the combined push site for both updates on Lines 43–47. First, it invokes `sync` to recompute the flow after the update and check that choosing $\{\texttt{List}, \texttt{item}\}$ as the footprint is a sound, local update. Then, it asserts the invariant for each node participating in the footprint, Lines 44 and 45. Finally, it resets the duplicated materialization variables, Lines 46 and 47.

The implementation for `sync` follows the blueprint from Fig. 6. We instantiate the flow-specific helpers according to our intuition for the path-counting flow, initializing the path count (flow) at root nodes and forwarding it along `next` links:

$$\begin{aligned} I(\texttt{x}) &\equiv \texttt{x == List ? 1 : 0} \\ E_{\mathsf{pre}}(\texttt{x, y}) &\equiv \texttt{x\_0\_next == y ? x\_0\_flow : 0} \\ E_{\mathsf{post}}(\texttt{x, y}) &\equiv \texttt{x\_next == y ? x\_flow : 0} \end{aligned}$$

Note that the above definitions need to be copied verbatim into `sync`, because the instrumentation has to *dynamically* compute the flow. □

Lastly, it is worth noting that we also change rule FRead. When reading a pointer field `y->f` into `x`, then flow may propagate along the `f` edge if `y` receives flow. We capture this propagation using `Fprop` from Fig. 6, which assumes that the flow in `x` is at least what it receives from `y`, $E_{\mathsf{post}}(\texttt{y}, \texttt{x})$. In the running example, the read on Line 4 leads to the flow propagation on Line 30 in the instrumentation, where `item` is guaranteed to receive at least path count 1 from `List` because the shared `List` pointer is the root of the structure.

Memory Leaks. With the flow instrumentation in place, we are ready to check for memory leaks. To that end, we rely on the space invariant: we require that all objects, as captured by the space invariant, are either deleted or have a flow other than 0. The universal quantifier in this requirement is resolved by non-deterministically choosing some object with some field values, assuming the object satisfies the space invariant, and then asserting the above requirement. The following code implements this check for a type with fields $\texttt{f}_1, \ldots, \texttt{f}_n$:

```
int x = havoc; int x_f1, ..., x_fn, x_free, x_flow; materialize(x);
assume(x != NULL); assert(x_flow > 0 || x_free == 1);
```

It does not matter where exactly this code is inserted because it only poses a requirement to the invariant. If the back-end solver cannot satisfy this requirement, verification fails. Our prototype tool inserts the check at the end of `main`.

Rule	Input *com*	Flow-aware instrumentation *com* ⇝ ...
YIELD	`yield(x);`	`if (x_lock != 1) { Fmaterialize(x); assume(x_lock != 1); }`
LOCK	`lock(x);`	`avail(x); assume(x_lock == 0); x_lock = 1;`
UNLOCK	`release(x);`	`avail(x); assert(x_lock == 1); x_lock = 0;`
CWRITE	`x->f = y;`	`avail(x); assert(x_lock == 1); x_f = y; dyn_up(x, f); push(x);`

Figure 7. Instrumentation for verifying concurrent programs using a thread-modular abstraction. New rules and changes to existing ones are colored. The instrumentation uses locks to avoid unnecessary materializations when no interference is possible.

5 Concurrency

Our instrumentation is readily extended to perform a thread-modular abstraction [6,21,36,55], allowing us to verify the program as if it was executed concurrently by an arbitrary number of threads. To that end, our instrumentation verifies the program from the point of view of an isolated thread that is subject to *interference*, i.e., heap updates, from other threads. The interference is applied whenever the isolated thread may be preempted, at so-called *yield points*. The possible heap updates from interfering threads are dictated by the space invariant: we continue to enforce that all atomic updates maintain the space invariant, so we can rely on it to always hold even in the presence of concurrency. For our instrumentation, summarized in Fig. 7, this simply means that we re-materialize all pointers at every yield point.

To make this precise, we assume that the yield points of the program under scrutiny are annotated with commands `yield x`, for every pointer `x` in scope. Annotating yield points in a given program is straightforward as they are required after every atomic command. Note that this approach easily supports applying standard moverness arguments [45,19,24,40] to reduce the number of yields, thereby increasing both precision and performance. Our instrumentation then replaces these `yield x` with `Fmaterialize(x)`.

To improve the precision of our instrumentation, we make it lock-aware, with the goal of reducing unnecessary materializations. To that end, we equip all objects `x` with a lock `x->lock`, which can be acquired and released using commands `lock(x)` and `unlock(x)`. To distinguish which thread is holding a lock, our instrumentation uses value `0` to indicate that the lock is available, value `1` to indicate that the isolated thread holds the lock, and any other value to indicate that an interferer is holding the lock. Hence, the instrumentation for acquiring `x->lock` sets its value to `1` if it is currently available (rule LOCK), and releasing `x->lock` reverts it back to `0` if it is currently held (rule UNLOCK).

Using locks, we can now elide materializing `x` at a yield point if the isolated thread holds `x->lock`, rule YIELD. Since interferers cannot acquire the lock on behalf of the isolated thread, it is safe for the rule to include an assumption stating that the newly materialized lock of `x` is distinct from `1`.

Table 1. Each cell gives the number of solved benchmarks and average time over 10 runs. TRICERA and SEAHORN use instrumented benchmarks, while PREDATOR uses uninstrumented ones. "Broom" and "Shape" denote `memsafety-broom` and standard shape analysis benchmarks, respectively.

Properties	Benchmark Set	TRICERA	SEAHORN	PREDATOR	Total
sequential, safe	Broom	20 (9.4s)	21 (9.2s)	30 (0.8s)	31
sequential, safe	Shape1	9 (10.1s)	9 (4.3s)	8 (0.9s)	9
sequential, unsafe	Shape2	4 (4.8s)	4 (3.2s)	4 (0.9s)	4
concurrent, safe	Concurrent	4 (17.4s)	4 (39.6s)	0 (–)	5

To ensure that eliding materializations is sound, we add assertions to updates of `x` to guarantee that its lock is held during the update, rule CWRITE. This enforces that all threads adhere to this policy, so that holding `x->lock` indeed prevents interferers from updating `x`. More involved locking strategies are straightforward to integrate, but are beyond the scope of this paper.

Example 6. Consider the `pop` function from Fig. 1, including the concurrent extension in pink color. Line 4 reads `item` from the locked `List`'s next field. Between this line and the subsequent locking of `item` on Line 6, there is a yield point. At this yield point, `item` is re-materialized because it is not locked. However, `List` is not re-materialized because it is locked, Line 3. In particular, this allows our instrumentation to *remember* that `item` equals `List->next` on Line 7, which is crucial to derive the fact that `item` is being unlinked. □

Our instrumentation enabled us to verify complex concurrent structures, such as lock-based linked lists and trees (cf. §6), substantiating the merit of our approach. Notably, it generalizes far more easily to various settings compared to traditional, hand-crafted shape analyses.

6 Evaluation

We have implemented our proposed approach in a tool called TRICERATOPS. TRICERATOPS accepts programs in a C-like language, performs the instrumentation from the previous sections, and analyzes the instrumented program with an off-the-shelf solver to verify the assertions added through instrumentation, thereby proving memory safety. The supported solver toolchains are SEAHORN [23,66] with Z3/SPACER [39] as well as TRICERA [20] with ELDARICA [29,28]. Our tool currently requires manual preparation of input programs to match its simplified input language, such as separating chained pointer dereferences into individual ones. While automating this manual preprocessing step is conceptually straightforward, we did not prioritize it in the prototype.

Our instrumentation relies on several constructs and operators not native to `C` but widely supported by verification tools: (i) `havoc` for non-deterministically choosing values, (ii) `assert` for raising a runtime error if a given condition fails,

and (iii) `assume` for blocking the execution if the given condition is not satisfied. Additionally, our encoding of space invariants Inv depends on (iv) uninterpreted predicates [20,66], which are less commonly supported. Our back-end solvers, SEAHORN and TRICERA, support all of these features. While we are not aware of a general method to encode uninterpreted predicates in any verification tool, we believe it is straightforward to do so in Horn clause-based tools.

Limitations of the current implementation are that we rely on a simplified acyclicity requirement for the localized flow update instrumentation instead of running a fixed point computation, for performance reasons. This precludes handling of data structures like circular lists. Currently, we also only support the path-counting flow domain. Neither of these are inherent limitations of the technique; our implementation can be extended to support different flow domains, but we decided to focus on the path-counting flow domain since it suffices to handle a large variety of data structures most relevant in practice, and to demonstrate the usefulness of our technique. As mentioned previously, our analysis is a whole-program analysis—if the input program has no `main` function, TRICERATOPS inserts one that implements a most general client by invoking the remaining program functions in a nondeterministic order and analyzes it.

We evaluated TRICERATOPS on a set of benchmarks that covers a variety of shapes, including singly-linked, doubly-linked, and nested lists, as well as trees, to substantiate that our approach is capable of verifying programs where this task inherently relies on shape information. Our benchmark set is divided into three parts: (1) 31 sequential `memsafety-broom` [1] examples from SV-COMP [7,8], featuring singly-linked, doubly-linked, and nested lists. Amongst these, four examples feature circular queues and three more examples feature pointer arithmetic and are thus incompatible with TRICERATOPS. We limit our evaluation to the `memsafety-broom` subset of the `MemSafety-LinkedLists` sub-category of SV-COMP due to the manual preprocessing of benchmarks required by TRICERATOPS. (2) 13 standard sequential shape analysis examples with singly- and doubly-linked lists and binary trees, including four with memory-safety violations, to stress-test TRICERATOPS. (3) Five memory-safe examples of concurrent singly-linked lists and binary trees, designed for memory safety verification; they non-deterministically generate, traverse (and possibly modify), and free dynamic structures in the heap.

We compare TRICERATOPS with the state-of-the-art shape analysis tool PREDATOR-HP [56,30], the 2024 SV-COMP gold medalist in the memory safety category [7]. Table 1 summarizes the number of sequential and concurrent benchmarks solved by TRICERATOPS using TRICERA and SEAHORN, and the number solved by PREDATOR-HP. Detailed results for the concurrent benchmarks are in Table 2. All experiments were conducted on an Apple M1 Pro.

The experiments show that our approach is capable of verifying intricate memory safety tasks. As expected, the runtimes are slower than those of a fine-tuned shape analysis tool like PREDATOR-HP. PREDATOR-HP is able to handle a wider variety of linked list examples than TRICERATOPS. It is able to correctly solve 30 of the 31 examples from the `memsaftey-broom` benchmark with runtimes

Table 2. Concurrent benchmarks with runtimes averaged over 10 runs. The TRICERATOPS column shows instrumentation time. TRICERA and SEAHORN columns show solving times using those backends. LoC = lines of code; LoI = lines after instrumentation. Symbols ✓ and ✗ indicate if the tool produced a correct verification verdict.

Benchmark	LoC	LoI	TRICERATOPS	TRICERA	SEAHORN
Running Example (Fig. 1)	37	303	0.77s	15.73s ✓	0.99s ✗
Coarse Stack	38	260	0.42s	8.37s ✓	4.39s ✓
Fine List [26, §9.5]	60	392	2.25s	30.56s ✓	92.19s ✓
Internal BST (no maintenance)	41	281	1.73s	14.77s ✓	5.78s ✓
Internal BST (simple removal)	78	441	6.04s	122.62s ✗	55.93s ✓

of less than one second, whereas TRICERATOPS can only handle 21. Seven of the examples from this set are incompatible with TRICERATOPS because they involve circular lists or pointer arithmetic. Another three examples fail due to unbounded flow updates. We note that swapping the order of two operations in each of these three benchmarks makes the resulting flow updates bounded, without changing the semantics. Such transformations could be implemented with a simple heuristic. Overall, the better performance of PREDATOR-HP on these benchmarks is expected since it is a highly specialized and mature tool.

However, our adaptable approach verifies sequential and concurrent tree benchmarks that are beyond PREDATOR-HP's capabilities. We believe that extending PREDATOR-HP to handle trees or concurrency would be non-trivial and labor intensive. This substantiates the usefulness of arithmetizing shape analysis to leverage a broader range of verification tools.

Notably, runtimes for sequential and concurrent benchmarks are comparable, likely due to materializing the space invariant, yielding similarly precise shape information in both cases, and the re-materialization required for concurrent instrumentation (cf. rule YIELD) can be dealt with efficiently by the solvers.

Finally, our over-approximate analysis may produce spurious counter-examples. However, these often allow reconstructing *why* a failing assertion was added, revealing actual bugs in the input program. Although not automated, this approach was effective during the development of TRICERATOPS and our experiments.

7 Related Work

Shape Analysis. We briefly survey existing approaches in shape analysis that our work is related to; for a more in-depth overview, we refer the reader to [13].

A parametric framework based on three-valued logic was introduced in [59] and later reformulated in terms of predicate abstraction in [57]. The effectiveness of this approach depends on manually chosen predicates tailored to each program.

The tool PREDATOR [18] uses an abstract shape domain for lists. PREDATOR can report spurious counterexamples, which is addressed in the version PREDATOR-HP [53] by running additional instances of PREDATOR without heap abstraction

in parallel. PREDATOR only provides limited support for non-pointer data, and cannot handle programs with trees, skip-lists, or concurrency in a sound way. The approach of PREDATOR is also used by CPACHECKER [3] in combination with symbolic execution [9] for the purpose of checking unbounded memory safety. MEMCAD [62,34,22] combines sequence and shape abstractions and supports lists and trees. The sequence abstraction enables the tracking of data constraints on such data structures, like value ranges and lengths of sequences.

Several studies employ expensive global reasoning about heap memory, but deliberately stay within decidable logics in order to enable automation [35,44,46,67].

More generally, many points-to domains have been proposed to reason abstractly about pointers and linked data-structures stored on the heap, e.g., [61,27]. Such domains are less precise, but can be implemented more efficiently than shape analysis domains. Space invariants resemble points-to analysis in that they summarize the possible states of objects using a symbolic invariant; the inference of space invariants is carried out symbolically using a model checker, however, and not with the help of abstract interpretation. Refined versions of space invariants [38] are related to recency-abstraction [4], which distinguishes objects based on allocation sites and summarizes object states separately for the most recent allocation and all earlier allocations.

Separation Logic and the Flow Framework. Separation logic (SL) [54,58] is widely used for reasoning about memory safety and functional properties. It enables compositional reasoning by partitioning the heap into smaller regions, allowing properties for different heap regions to be expressed locally, such that modifications to one region do not invalidate properties about other regions.

The flow framework [42,43,50], which we build on in this paper, endows the (heap) graph with a flow that allows global properties to be specified in terms of node-local invariants by referring to that flow. Flows augment the nodes in the heap with additional ghost information, and are computed inductively over the graph structure using data-flow equations. The flow framework has been used in the verification of sophisticated algorithms that are difficult to handle by other techniques, such as the Priority Inheritance Protocol, object-oriented design patterns, and complex concurrent data structures [41].

A general proof technique for reasoning about global graph properties using the flow framework is presented in [43]. The framework automatically checks that each modified heap region preserves invariants, and is implemented using the tool VIPER. In this setting, invariants must be manually provided but can then be verified automatically. Similarly, [48,49,47] devise (semi-)automatic flow framework-based techniques for fine-grained concurrent data structures.

Bi-abduction-based Shape Analysis. Bi-abduction [12] is an SL-based shape analysis technique that simultaneously identifies both the missing preconditions required for safe code execution and the portions of memory that remain unchanged after execution. This enables a compositional analysis strategy that analyzes parts of a program independently and later combines the results, leading to improved scalability, as demonstrated by the success of the INFER tool [11].

The BROOM tool implements a bi-abduction analysis for low-level C code [33]. Classical bi-abduction often yields imprecise invariants for loops. The authors of [60] address this limitation by introducing "biabductive loop acceleration", which directly constructs and verifies candidate loop invariants, complemented by a shape extrapolation heuristic that leverages locality in list-like data structures.

Our technique has strengths and weaknesses orthogonal to those of bi-abduction-based approaches. On the one hand, our reduction-based approach enables easy integration into existing software verification tool chains. The approach also promises to be more easily extensible, e.g., by instantiating the flow abstraction with different flow domains to track different kinds of shape properties. In particular, we can use the same flow domain to prove memory safety of both concurrent and sequential programs manipulating lists and trees.

On the other hand, our approach reduces the verification problem to a whole program analysis that is not immediately amenable to the more efficient compositional reasoning that underlies biabduction techniques. Exploring a combination of the two approaches is an interesting direction for future work.

8 Conclusions

We have presented a new automatic shape analysis method based on two reasoning principles: *flow abstraction,* which reduces global properties of the heap graph to local flow equations that are required to hold for every object on the heap, and *space invariants,* for representing an unbounded number of heap objects using symbolic invariants. As our approach is implemented through a source-to-source transformation, it can be used in conjunction with different verification back-ends, and is able to leverage all data types and language features supported by the back-end tool. Our experiments show that the analysis approach covers a wide range of shapes and can even be extended to concurrent programs.

Several avenues for future work exist. At the moment, concurrency support in TRICERATOPS is only experimental, more research is needed to work out the details of how to analyze concurrent programs operating on linked data structures using our approach. We also plan to investigate the use of other flow domains, beyond path counting, to obtain more precise shape analysis. Lastly, the details of how to combine shape analysis with data analysis (e.g., sortedness of lists or well-formedness of search trees) remain to be investigated.

Acknowledgements. This work is funded in parts, by the National Science Foundation under grant GS1000000012304758 and by the Swedish Research Council (VR) under the grant 2021-06327. The first author was supported by a Junior Fellowship from the Simons Foundation (855328, SW).

Disclaimers. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. c/memsafety-broom benchmark suite of SV-COMP 2024. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp24-final/c/memsafety-broom, [Accessed 04/08/2025]
2. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Ji, R., Rezine, A.: Shape analysis via monotonic abstraction. In: Beyond the Finite: New Challenges in Verification and Semistructured Data. Dagstuhl Seminar Proceedings, vol. 08171. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2008)
3. Baier, D., Beyer, D., Chien, P., Jankola, M., Kettl, M., Lee, N., Lemberger, T., Rosenfeld, M.L., Spiessl, M., Wachowitz, H., Wendler, P.: Cpachecker 2.3 with strategy selection - (competition contribution). In: TACAS (3). Lecture Notes in Computer Science, vol. 14572, pp. 359–364. Springer (2024)
4. Balakrishnan, G., Reps, T.W.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4134, pp. 221–239. Springer (2006). https://doi.org/10.1007/11823230_15, https://doi.org/10.1007/11823230_15
5. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: CAV. Lecture Notes in Computer Science, vol. 4590, pp. 178–192. Springer (2007)
6. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, S.: Thread quantification for concurrent shape analysis. In: CAV. Lecture Notes in Computer Science, vol. 5123, pp. 399–413. Springer (2008)
7. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: TACAS (3). Lecture Notes in Computer Science, vol. 14572, pp. 299–329. Springer (2024)
8. Beyer, D.: Sv-benchmarks: Benchmark set for software verification (SV-COMP 2024) (version svcomp24). https://doi.org/10.5281/zenodo.10669723 (Mar 2024). https://doi.org/10.5281/ZENODO.10669723, https://doi.org/10.5281/zenodo.10669723, accessed on YYYY-MM-DD.
9. Beyer, D., Lemberger, T.: Cpa-symexec: efficient symbolic execution in cpachecker. In: ASE. pp. 900–903. ACM (2018)
10. Bouajjani, A., Dragoi, C., Enea, C., Rezine, A., Sighireanu, M.: Invariant synthesis for programs manipulating lists with unbounded data. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 72–88. Springer (2010)
11. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: NASA Formal Methods (NFM). Lecture Notes in Computer Science, vol. 6617, pp. 459–465. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_33, https://doi.org/10.1007/978-3-642-20398-5_33
12. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL. pp. 289–300. ACM (2009)
13. Chang, B.E., Dragoi, C., Manevich, R., Rinetzky, N., Rival, X.: Shape analysis. Found. Trends Program. Lang. **6**(1-2), 1–158 (2020)
14. Chang, B.E., Rival, X.: Relational inductive shape analysis. In: POPL. pp. 247–260. ACM (2008)
15. Chang, B.E., Rival, X., Necula, G.C.: Shape analysis with structural invariant checkers. In: SAS. Lecture Notes in Computer Science, vol. 4634, pp. 384–401. Springer (2007)

16. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: TACAS. Lecture Notes in Computer Science, vol. 3920, pp. 287–302. Springer (2006)
17. Dudka, K., Peringer, P., Vojnar, T.: Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 372–378. Springer (2011)
18. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: SAS. Lecture Notes in Computer Science, vol. 7935, pp. 215–237. Springer (2013)
19. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL. pp. 2–15. ACM (2009)
20. Esen, Z., Rümmer, P.: Tricera: Verifying C programs using the theory of heaps. In: FMCAD. pp. 380–391. IEEE (2022)
21. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: SPIN. Lecture Notes in Computer Science, vol. 2648, pp. 213–224. Springer (2003)
22. Giet, J., Ridoux, F., Rival, X.: A product of shape and sequence abstractions. In: SAS. Lecture Notes in Computer Science, vol. 14284, pp. 310–342. Springer (2023)
23. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: CAV (1). Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015)
24. Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S.: Automated and modular refinement reasoning for concurrent programs. In: CAV (2). Lecture Notes in Computer Science, vol. 9207, pp. 449–465. Springer (2015)
25. Haziza, F., Holík, L., Meyer, R., Wolff, S.: Pointer race freedom. In: VMCAI. Lecture Notes in Computer Science, vol. 9583, pp. 393–412. Springer (2016)
26. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
27. Hind, M., Pioli, A.: Assessing the effects of flow-sensitivity on pointer alias analyses. In: Levi, G. (ed.) Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1503, pp. 57–81. Springer (1998). https://doi.org/10.1007/3-540-49727-7_4, https://doi.org/10.1007/3-540-49727-7_4
28. Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems - tool paper. In: FM. Lecture Notes in Computer Science, vol. 7436, pp. 247–251. Springer (2012)
29. Hojjat, H., Rümmer, P.: The ELDARICA horn solver. In: FMCAD. pp. 1–7. IEEE (2018)
30. Holík, L., Kotoun, M., Peringer, P., Soková, V., Trtík, M., Vojnar, T.: Predator shape analysis tool suite. In: Haifa Verification Conference. Lecture Notes in Computer Science, vol. 10028, pp. 202–209 (2016)
31. Holík, L., Lengál, O., Rogalewicz, A., Simácek, J., Vojnar, T.: Fully automated shape analysis based on forest automata. In: CAV. Lecture Notes in Computer Science, vol. 8044, pp. 740–755. Springer (2013)
32. Holík, L., Meyer, R., Vojnar, T., Wolff, S.: Effect summaries for thread-modular analysis. CoRR **abs/1705.03701** (2017)
33. Holík, L., Peringer, P., Rogalewicz, A., Soková, V., Vojnar, T., Zuleger, F.: Low-level bi-abduction. In: ECOOP. LIPIcs, vol. 222, pp. 19:1–19:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
34. Illous, H., Lemerre, M., Rival, X.: A relational shape abstract domain. Formal Methods Syst. Des. **57**(3), 343–400 (2021)

35. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: CAV. Lecture Notes in Computer Science, vol. 8044, pp. 756–772. Springer (2013)
36. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
37. Jones, N.D., Muchnick, S.S.: Flow analysis and optimization of lisp-like structures. In: POPL. pp. 244–256. ACM Press (1979)
38. Kahsai, T., Kersten, R., Rümmer, P., Schäf, M.: Quantified heap invariants for object-oriented programs. In: LPAR. EPiC Series in Computing, vol. 46, pp. 368–384. EasyChair (2017)
39. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in smt-based unbounded software model checking. In: CAV. Lecture Notes in Computer Science, vol. 8044, pp. 846–862. Springer (2013)
40. Kragl, B., Qadeer, S.: Layered concurrent programs. In: CAV (1). Lecture Notes in Computer Science, vol. 10981, pp. 79–102. Springer (2018)
41. Krishna, S., Patel, N., Shasha, D.E., Wies, T.: Verifying concurrent search structure templates. In: PLDI. pp. 181–196. ACM (2020)
42. Krishna, S., Shasha, D.E., Wies, T.: Go with the flow: compositional abstractions for concurrent data structures. Proc. ACM Program. Lang. **2**(POPL), 37:1–37:31 (2018)
43. Krishna, S., Summers, A.J., Wies, T.: Local reasoning for global graph properties. In: ESOP. Lecture Notes in Computer Science, vol. 12075, pp. 308–335. Springer (2020)
44. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: POPL. pp. 171–182. ACM (2008)
45. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)
46. Madhusudan, P., Qiu, X., Stefanescu, A.: Recursive proofs for inductive tree data-structures. In: POPL. pp. 123–136. ACM (2012)
47. Meyer, R., Opaterny, A., Wies, T., Wolff, S.: nekton: A linearizability proof checker. In: CAV (1). Lecture Notes in Computer Science, vol. 13964, pp. 170–183. Springer (2023)
48. Meyer, R., Wies, T., Wolff, S.: A concurrent program logic with a future and history. Proc. ACM Program. Lang. **6**(OOPSLA2), 1378–1407 (2022)
49. Meyer, R., Wies, T., Wolff, S.: Embedding hindsight reasoning in separation logic. Proc. ACM Program. Lang. **7**(PLDI), 1848–1871 (2023)
50. Meyer, R., Wies, T., Wolff, S.: Make flows small again: Revisiting the flow framework. In: TACAS (1). Lecture Notes in Computer Science, vol. 13993, pp. 628–646. Springer (2023)
51. Meyer, R., Wolff, S.: Decoupling lock-free data structures from memory reclamation for static analysis. Proc. ACM Program. Lang. **3**(POPL), 58:1–58:31 (2019)
52. Meyer, R., Wolff, S.: Pointer life cycle types for lock-free data structures with memory reclamation. Proc. ACM Program. Lang. **4**(POPL), 68:1–68:36 (2020)
53. Müller, P., Peringer, P., Vojnar, T.: Predator hunting party (competition contribution). In: TACAS. Lecture Notes in Computer Science, vol. 9035, pp. 443–446. Springer (2015)
54. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: CSL. Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer (2001)
55. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976)

56. Peringer, P., Soková, V., Vojnar, T.: Predatorhp revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: TACAS (2). Lecture Notes in Computer Science, vol. 12079, pp. 408–412. Springer (2020)
57. Podelski, A., Wies, T.: Boolean heaps. In: SAS. Lecture Notes in Computer Science, vol. 3672, pp. 268–283. Springer (2005)
58. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)
59. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002)
60. Sextl, F., Rogalewicz, A., Vojnar, T., Zuleger, F.: Compositional shape analysis with shared abduction and biabductive loop acceleration. In: Programming Languages and Systems - 34th European Symposium on Programming (ESOP). Lecture Notes in Computer Science, Springer (2025), to appear
61. Steensgaard, B.: Points-to analysis in almost linear time. In: Boehm, H., Jr., G.L.S. (eds.) Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996. pp. 32–41. ACM Press (1996). https://doi.org/10.1145/237721.237727, https://doi.org/10.1145/237721.237727
62. Toubhans, A., Chang, B.E., Rival, X.: Reduced product combination of abstract domains for shapes. In: VMCAI. Lecture Notes in Computer Science, vol. 7737, pp. 375–395. Springer (2013)
63. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge, UK (2008)
64. Vafeiadis, V.: Automatically proving linearizability. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 450–464. Springer (2010)
65. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR. Lecture Notes in Computer Science, vol. 4703, pp. 256–271. Springer (2007)
66. Wesley, S., Christakis, M., Navas, J.A., Trefler, R.J., Wüstholz, V., Gurfinkel, A.: Inductive predicate synthesis modulo programs. In: ECOOP. LIPIcs, vol. 313, pp. 43:1–43:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024)
67. Wies, T., Muñiz, M., Kuncak, V.: An efficient decision procedure for imperative tree data structures. In: CADE. Lecture Notes in Computer Science, vol. 6803, pp. 476–491. Springer (2011)
68. Wolff, S., Gupta, E., Esen, Z., Hojjat, H., Rümmer, P., Wies, T.: Arithmetizing shape analysis. CoRR **abs/2408.09037** (2024). https://doi.org/10.48550/ARXIV.2408.09037, https://doi.org/10.48550/arXiv.2408.09037