Compositional Shape Analysis with Shared Abduction and Biabductive Loop Acceleration

Florian Sextl[1](✉), Adam Rogalewicz[2], Tomáš Vojnar[2,3], and Florian Zuleger[1]

[1] TU Wien, Faculty of Informatics, Institute of Logic and Computation, Vienna, Austria, florian.sextl@tuwien.ac.at
[2] Brno University of Technology, Faculty of Information Technology, Brno, Czechia
[3] Masaryk University, Faculty of Informatics, Brno, Czechia

Abstract. Biabduction-based shape analysis is a compositional verification and analysis technique that can prove memory safety in the presence of complex, linked data structures. Despite its usefulness, several open problems persist for this kind of analysis; two of which we address in this paper. On the one hand, the original analysis is path-sensitive but cannot combine safety requirements for related branches. This causes the analysis to require additional soundness checks and decreases the analysis' precision. We extend the underlying symbolic execution and propose a framework for *shared abduction* where a common precondition is maintained for related computation branches. On the other hand, prior implementations lift loop acceleration methods from forward analysis to biabduction analysis by applying them separately on the pre- and post-condition, which can lead to imprecise or even unsound acceleration results that do not form a loop invariant. In contrast, we propose *biabductive loop acceleration*, which explicitly constructs and checks candidate loop invariants. For this, we also introduce a novel heuristic called *shape extrapolation*. This heuristic takes advantage of locality in the handling of list-like data structures (which are the most common data structures found in low-level code) and jointly accelerates pre- and post-conditions by extrapolating the related shapes. In addition to making the analysis more precise, our techniques also make biabductive analysis more efficient since they are sound in just one analysis phase. In contrast, prior techniques always require two phases (as the first phase can produce contracts that are unsound and must hence be verified). We experimentally confirm that our techniques improve on prior techniques; both in terms of precision and runtime of the analysis.

Keywords: Shape Analysis · Biabduction

1 Introduction

Over the last two decades, *shape analysis* has proven to be one of the most useful techniques for ensuring memory safety in programs. This kind of analysis focuses on verifying memory-safe handling of linked data structures by representing them as abstract memory *shapes*. Thereby, memory safety can often be verified with fully automatic reasoning for a wide range of data structures. Examples of successful shape analyzers

F. Sextl—Main author, other authors in alphabetical order of their last names

V. Vafeiadis (Ed.): ESOP 2025, LNCS 15695, pp.230–257, 2025.
https://doi.org/10.1007/978-3-031-91121-7_10

include the tools Predator [10], which has won a number of medals at the well-known SV-COMP competition (see [2,3]), and Infer [4], which has been used for several years to check large code bases at Meta (formerly Facebook).

Among the reasoning principles underlying shape analysis, *biabduction* has the unique ability to enable compositional analysis for open programs (i.e., program fragments) by synthesizing invariants and function contracts consisting of separation logic formulas [25]. The ability to synthesize contracts, i.e., pairs of pre- and post-conditions, allows for a highly modular inter-procedural analysis. In addition, the compositional analysis with biabduction enjoys what Calcagno et al. [5,6] have called "graceful imprecision", i.e., the analysis will find useful results for some parts of a program, even if it introduced imprecisions for other parts. In contrast, closed program analyses are likely to build up such imprecisions and fail, even if they could handle further parts of the program otherwise. Due to this and due to not requiring programmers to develop verification harnesses, biabduction-based shape analysis is often considered to be more useful for large-scale verification compared to techniques for closed programs, which are advantageous for smaller, self-contained programs. However, this advantage comes at the cost of more complex computation principles as well as generally less expressive abstract shapes. Moreover, existing biabduction-based shape analyses can compute unsound results and, thus, require a second analysis phase to filter out these results.

The highly path-sensitive analysis proposed by [5,6] does not work well with branching if the branch to be taken cannot be determined purely from the pre-condition of the analyzed function (e.g., because it depends on user input or because the used logical fragment cannot express the dependency sufficiently), see [6, section 4.3]. We call these cases *non-determinable branching*. The problem with these cases arises since maintaining different pre-conditions for each program path is, in general, insufficient because the only sound precondition might consist of a combination of these. Thus, Calcagno et al. suggested a heuristic procedure for merging pre-conditions, but their approach may fail to compute any valid pre-conditions (see Section 2.1) at all. In contrast, we present a novel technique of *shared abduction*, which allows for sound pre-condition computation across program branches. The technique extends biabductive symbolic execution by tracking which program locations share which pre-condition requirements. Due to this, shared abduction circumvents the need for a verification phase for programs with arbitrary branching, and, at the same time, can infer non-trivial contracts in more cases than the traditional analysis (since it does not give up on the cases where some sharing of information between branches is necessary).

Symbolic analyses generally require loop acceleration techniques to allow the analyses to reach a fixed point. In shape analysis, this acceleration replaces concrete pointers with more abstract, typically inductive shape predicates such as list segments. The analysis by [5] lifts such abstraction to the setting of biabductive analysis by applying abstraction separately to the pre- and the post-condition. However, such a direct lifting is not guaranteed to result in a sound loop invariant (see [6, section 4.3] or [29, Appendix F.1]). In contrast, we introduce a novel *biabductive loop acceleration* scheme that constructs candidate loop invariants after analyzing the loop body once. This step allows us to verify the soundness of the candidate invariant explicitly through another symbolic execution of the loop body. To construct the candidate invariant, we intro-

duce a novel *shape extrapolation* heuristic, which exploits the locality of typical data structure traversals to find fitting abstract shape predicates.

Even though the two-phase approach is easy to implement as the two phases primarily differ in which biabduction rules are used, and even though most related works rely on the second phase (see [13,12]), it is quite natural to wonder whether the overhead of the two phases can be reduced. Overcoming this overhead for the broader family of biabduction-based shape analyses is exactly the goal of this paper. Thereby, our shared abduction technique avoids the unsoundness problem for non-determinable branching and circumvents the need for the second phase (at the same time, producing more non-trivial contracts than the previous approaches). In addition, our loop acceleration approach only requires us to check the extrapolated invariant for soundness, but this check is much more local and less costly than the second analysis phase.[4] Our approach to constructing the invariant is heuristic and, hence, not always applicable, but the underlying invariant check is still guaranteed to assert soundness. In addition, our experiments demonstrate that our approach can significantly improve the efficiency, i.e., reduce the needed number of loop iterations and lower the runtime compared to the two-phase biabduction architecture in practice. We conjecture that this is because programmers do commonly write loops in a way compatible with our approach.

Main Contributions. The main contributions of this work in the context of analyzing sequential, non-recursive programs are the following:

- A novel sound analysis for loop-free code based on *shared abduction* (Section 4).
- A novel sound way to construct and check loop invariants as part of biabductive shape analysis via *biabductive loop acceleration*. It uses a novel heuristic to exploit locality via *shape extrapolation* on list-manipulating programs (Section 5).
- Formal proofs of shared abduction and biabductive loop acceleration being sound (Theorems 1 and 2 with proofs in [29, Appendix D]).
- An experimental evaluation based on a proof-of-concept implementation applied to a number of small-scale but rather challenging programs, including real-life library functions, that show the superiority of our approach with regard to runtime and completeness compared to established analyzers (Section 6).

General Limitations. Our acceleration approach is currently limited to programs manipulating various kinds of lists (singly or doubly linked, possibly circular, nested, and intrusive). While this restriction coincides with prior work [6,13], we hope that exploiting locality for loop acceleration will apply to further data structures (such as trees), but we must leave this for future work. Furthermore, we only focus on non-recursive programs, following most previous biabductive shape analysis approaches. Qin et al. [22] introduced an extension to handle recursion via a fixed point computation of the function contract, but this is orthogonal to our work. Moreover, our prototype tool is focused on low-level C code, which rarely contains recursion anyway.

An extended version of this paper with the full appendix can be found as [29].

[4] The traditional approaches to biabduction, such as [13,12], will analyze each loop at least twice (to get to a fixpoint) in each of the phases, i.e., each loop is analyzed *at least four times*, but often even more (see table 2). On the other hand, our approach may soundly find a loop invariant within one analysis phase, which analyses the loop in general only twice.

true for $\texttt{read} \leq 0 \vee \texttt{in\_mode} \geq 2 \quad | \quad hd \mapsto \ell_1 * out \mapsto \ell_4$ for $\texttt{in\_mode} = 0$
$curr \neq hd : curr \mapsto \ell_3 * out \mapsto \ell_4$ and $curr = hd : lst \mapsto \ell_2 * out \mapsto \ell_4$ for $\texttt{in\_mode} = 1$

Fig. 1: Insufficient candidate pre-conditions for `user_choice`

2 Motivation

2.1 Cross-Branch Abduction Sharing

```
int user_choice(node *hd, node *lst,
    node *curr, node *out) {
  int in_mode = 0;
  int read = scanf("%d", &in_mode);

  if (read <= 0) {
    return -1;
  } else if (in_mode == 0) {
    memcpy(out, hd,...);
  } else if (in_mode == 1) {
    if (curr != hd) {
      memcpy(out, curr,...);
    } else {
      memcpy(out, lst,...);
    }
  }
  return in_mode;
}
```

Listing 1: Non-determinable branching

Listing 1 shows a program fragment that works on a data node based on user input read from the command line via `scanf`. Based on this, `user_choice` takes the user input and calls `memcpy` with the corresponding arguments. The exact invocation depends on the user input,[5] which is modeled by a non-deterministic choice in the analysis, and, in the case of $\texttt{in\_mode} = 1$, also depends on whether $\texttt{curr} = \texttt{hd}$. We call these kinds of branching *non-determinable*, since the branch taken at runtime can't be determined from the function parameters alone.

Problem. Non-determinable branching is difficult to handle for a path-sensitive biabduction-based shape analysis as proposed in [6,5]. This is because such an analysis will generate one precondition per program branch in Figure 1, expressed with standard separation logic connectives.[6]

We note that the preconditions can take into account branching conditions that depend on the function's arguments, e.g., the preconditions in Figure 1 contain the predicates $curr = hd$ and $curr \neq hd$. However, non-determinable branching, such as for $in\_mode = 0$, cannot be modeled in terms of the function's arguments, and hence such conditions can never be part of a precondition. Then the problem arises that the different branches require different memory locations to be allocated (note the different pointers arguments to `memcpy`), e.g. $hd \mapsto \ell_1 * out \mapsto \ell_4$ for the branch with $\texttt{in\_mode} = 0$. However, due to the non-deterministic input, none of the required allocations for one branch guarantee a memory-safe execution for all user inputs.

Previous Solutions. This problem has already been noticed in the original work [6,5] and was partially addressed by a heuristic that would combine pre-conditions such that they could cover move branches. This heuristic has been implemented as an optional strategy in the Abductor tool and was subsequently made the default in the Infer tool.

[5] We chose user input as an easy to understand example of non-determinable input. Other cases of such input includes IO operations such as incoming network traffic or reading from a file.

[6] We use ":" to separate the formulas' pure and spatial parts (if any). In contrast to the program variables *hd*, etc., the ℓ_i variables are purely logical and implicitly universally quantified. We write program variables in formulas in *italic* and otherwise in `typewriter` font.

However, this heuristic does not guarantee that the found pre-conditions are sound, and it can easily miss safe pre-conditions even for simple loop-free code. Indeed, for the example above, the heuristic finds the combined pre-condition $curr \neq hd : hd \mapsto \ell_1 * lst \mapsto \ell_2 * curr \mapsto \ell_3 * out \mapsto \ell_4$, but it does not produce a sound pre-condition for the case `curr = hd`. We also remark that the heuristic is quite fragile as renaming the variable `hd` to `first` enables the heuristic to find a safe pre-condition for the case `curr = hd` in Infer while leading to a crash for Abductor.[7]

We finally note that recent work [33], developed concurrently with our approach, also addresses the problem of unsound pre-conditions for branching programs. They introduce a specialized operator called *tri-abduction*, which generalizes the setting of *bi-abduction* to simultaneously compute a combined pre-condition for two branches. While this operator offers more precision than the (classical) biabduction operator we rely on in this paper, it is unclear how to build a realistic symbolic execution based on the tri-abduction operator. To this date, such an analysis has only been sketched but not implemented. We comment more on the relationship to our approach in Section 7.

Shared Abduction. The fundamental problem discussed above is that the different program paths cannot be analyzed in isolation; instead, we must combine their preconditions. That is, biabduction-based analyzers need to track which program configurations can be reached from the same initial configuration and synchronize the abduced requirements. Moreover, the analysis needs to be precise in tracking which configurations are allowed to exchange such information – otherwise, inconsistencies can be introduced by exchanging information among independent program points. Our solution is, therefore, to track exactly which program configurations can be reached from a common pre-condition and explicitly share newly found requirements with all of these configurations (and only such configurations). Section 4 introduces how this technique, which we call *shared abduction*, allows sound handling of all kinds of branching.

Our technique has the advantage of being lightweight and easily implementable on top of an existing biabductive analysis. In the example above, our analysis first abduces the precondition $hd \mapsto \ell_1 * out \mapsto \ell_4$ for the case `in_mode = 0`. The analysis then proceeds with the branch for `in_mode = 1`, making a case distinction on $curr = hd$. Shared abduction retains the required allocation $hd \mapsto \ell_1 * out \mapsto \ell_4$ for both cases as this requirement is already part of the shared precondition. Then, by analyzing the nested branches, the requirements $curr \neq hd : hd \mapsto \ell_1 * curr \mapsto \ell_3 * out \mapsto \ell_4$ and $curr = hd : hd \mapsto \ell_1 * lst \mapsto \ell_2 * out \mapsto \ell_4$ are computed. We note that this guarantees the soundness of the found pre-condition and its completeness regarding the branching, thus outperforming the previous heuristic.

2.2 Shape Extrapolation for Biabductive Acceleration

Our second contribution aims at the analysis of loops, which generally requires accelerating the symbolic execution to allow the analysis to reach a fixed point.

[7] We observed this behavior with the commit f93cb281edb33510d0a300f1e4c334c6f14d6d26 found at https://github.com/facebook/infer and the publicly available Abductor release at http://www0.cs.ucl.ac.uk/staff/p.ohearn/abductor.html.

Problems. The prior technique for loop acceleration, proposed by [5,6] and adopted in [13], separately abstracts the pre- and post-condition with no other information than the formulas themselves taken into account. Intuitively, after analyzing some loop iterations and applying the abstraction operator, the obtained formulas will stabilize, and a fixed point is reached. Thereby, the abstraction follows the intuitive principle of collecting linked memory blocks with a similar layout into a single abstract shape predicate. In the context of simple singly-linked lists, this means that the abstraction procedure scans the formula for points-to predicates $x \mapsto \ell$ and $\ell \mapsto z$, linked by a location ℓ (i.e., the target of the first predicate contains the address of the second), or a linked list segment $ls(x, \ell)$ and a points-to $\ell \mapsto z$, respectively. Abstraction then replaces these predicates with the single predicate $ls(x, z)$. However, abstraction cannot be applied when there is a program variable y that references ℓ, e.g., as $y = \ell$. This is not supported since it would lose the information that variable y is allocated (note that ℓ does not occur in $ls(x, z)$ anymore). More generally, abstraction cannot be applied if there is a program variable y whose value depends on ℓ, such as $y = v \wedge \ell \mapsto v$. While the abstraction principle is intuitive, there are also multiple drawbacks, which we discuss next.

```
void weighted_sum(o_node *o,
                  long *sum){
  while (o != NULL) {
    i_node *i = o->inner;
    while (i != NULL) {
      *sum = (*sum) +
        (o->wgt * i->elem);
      i = i->next;
    }
    o = o->next_o;
} }
```

Listing 2: Nested list traversal

(1) For the example in Listing 2, Abductor, Infer, and Broom do not reach a fixed point for the inner loop and thus cannot synthesize any contract. This is because the value pointed to by sum after n loop iterations is $\ell_{sum} + \ell_1 \cdot \ell_w + \cdots + \ell_n \cdot \ell_w$, where ℓ_w is the value pointed to by o.`wgt` and the ℓ_i are the `elem` values of the list nodes traversed so far. Thus, the dependence on the values ℓ_i blocks abstraction (as described above). We note that the design of a stronger abstraction operator is not straight-forward because we also need to track values in memory precisely as they could be relevant for memory accesses based on pointer arithmetic in other parts of the program.

```
void traverse_skip_two(node *list) {
  node *tmp = list->next->next;
  while (tmp != NULL) {
    tmp = tmp->next;
} }
```

Listing 3: Offset list traversal

(2) A formula-based abstraction operator can easily lose too much information. For example, Abductor, Infer, and Broom fail for the simple example in Listing 3. The reason is as follows: The abstraction operation (as described above) contracts pointer chains of length at least two into a list segment, resulting in the formula $list \neq NULL : ls(list, NULL)$. This predicate describes a non-empty list segment with at least one node. However, this formula does not suffice as a pre-condition that guarantees memory safety because `traverse_skip_two` requires a list of length ≥ 2 as input.

(3) Furthermore, acceleration based on abstraction (as described above and implemented in Abductor, Infer, and Broom) can be highly inefficient. In general, every loop will require at least two (often three) analysis iterations, as abstraction can often only be applied after the second loop iteration and a fixed point can only be checked for after another iteration. In the presence of inner loops, such as for the example in Listing 2, this quickly multiplies, e.g., amounting to nine analysis iterations for the inner loop in Listing 2 just for the first analysis phase.

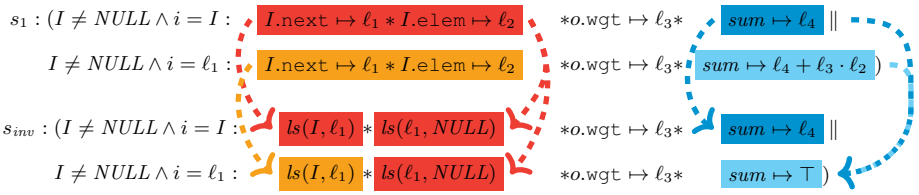

Fig. 2: State s_1 after the first loop iteration analysis and the constructed candidate invariant s_{inv}, with color-coded arrows showing the information flow between different subformulas

The problems described above are mostly related to the direct application of abstraction for acceleration and its missing ability to take into account more information about the loop, e.g. the observation that inductive data structures are often traversed one step at a time. For example, in Listing 2, the nested linked list is traversed in such a way to compute the weighted sum of the elements in the inner lists. It is apparent that each of the two loops operates on a local, shifting view of the respective traversed list plus some context. For the inner loop, this means that the loop only operates on a unique `i_node` at a time while also accessing the same variables `sum` and `o->wgt` in each iteration. Similarly, the outer loop only operates on one `o_node` at a time. These shifting views on the traversed shapes are akin to "local actions" (see [7]).

Biabductive Loop Acceleration. This observation allows us to extrapolate what the analysis abduces from a single iteration to arbitrarily many iterations and directly compute a candidate loop invariant if applicable. We call this heuristic *shape extrapolation*. It is part of our *biabductive loop acceleration*, which consists of three main steps: First, we use the analysis result of a single iteration to obtain locality information about the shape and the context; second, we use this information to extrapolate the shape to an abstract one; third, we check that the heuristically constructed state is a sound invariant.

In the case of `weighted_sum`, after the first iteration of the inner loop, the analysis finds the state s_1 depicted in Figure 2, consisting of a pre- and post-condition separated by $\parallel$. Our analysis then partitions the pre- as well as the post-condition into a shape and a context part, where the shape part is $I.\texttt{next} \mapsto \ell_1 * I.\texttt{elem} \mapsto \ell_2$, and the context is $o.\texttt{wgt} \mapsto \ell_3 * sum \mapsto \ell_4$ for the pre-condition and $o.\texttt{wgt} \mapsto \ell_3 * sum \mapsto \ell_4 + \ell_3 \cdot \ell_2$ for the post-condition. The following heuristic obtains this partitioning: We consider the changed variables (here `i`,`sum`) whose value moved to some pointer value (here `i` whose value moved to `i->next`). The predicates associated with these variables are then put into the shape parts and the others into the context.

Based on this partitioning, our procedure directly constructs a (candidate) loop invariant s_{inv} (see Figure 2). Thereby, our procedure accelerates the shape part of the pre- as well as the post-condition (here we obtain in both cases the list segment $ls(I, \ell_1)$, with internal next pointer `next` and data field `elem`). Intuitively, these predicates correspond to the loop iterations up to the current point. In addition, we add predicates $ls(\ell_1, NULL)$ to the shape part of s_{inv}, for both the pre-and post-condition, which are taken as the accelerated predicate $ls(I, \ell_1)$ of the pre-condition of s_1, where I has been replaced with l_1, which is the current value of i, and l_1 has been replaced with $NULL$, which has been obtained from the loop condition. Intuitively, these predicates corre-

spond to the future loop iterations up to the loop's termination. We refer to the red and orange colors in Fig. 2 to illustrate the information flow. The context part of s_{inv} keeps the context of s_1, except that our procedure abstracts the value of `sum` in the post-condition (with the unknown value $\top$) because it cannot be tracked precisely. Next, our procedure checks that s_{inv} is indeed a loop invariant, which requires one symbolic execution of the loop body and an entailment check.

Finally, based on the loop invariant our analysis constructs a contract that abstracts the inner loop and that can be used for the analysis of the outer loop:

$$(i = I : ls(I, NULL) * o.\texttt{wgt} \mapsto \ell_3 * sum \mapsto \ell_4 \parallel$$
$$i = NULL : ls(I, NULL) * o.\texttt{wgt} \mapsto \ell_3 * sum \mapsto \top).$$

Based on this contract, our procedure then also accelerates the outer loop in a similar fashion. Lastly, we note that our procedure requires exactly two iterations per loop (in sum four): one to analyze the effects of the loop and a second to check whether the constructed state is a loop invariant (as opposed to the nine iterations in sum mentioned earlier for the traditional acceleration). Similarly, our approach fails fast if the constructed candidate invariant is unsound instead of requiring a second analysis phase with many more analysis steps.

3 Preliminaries

We present our new techniques for a simple but standard setting that is described in the following since it does not require any specific logic fragment or biabduction method.

3.1 Programming Language and Memory Model

Let *Var* be a countably infinite set of variables and *Val* be a countably infinite set of values containing the value $\top$. Furthermore, let *Fld* be a finite set of field names and $Loc \subseteq Val$ be the set of memory locations such that $NULL \in Loc$.

Definition 1 (Programming Language). *We base our description on a C-like programming language. The language comprises standard expressions, statements for reading ($x_1 = *x_2.f$) and writing ($*x_1.f = x_2$) through pointers (with the C-like syntactic sugar of $x{->}f$ for $*x.f$), an additional non-determinism operator ?, as well as implicit control flow statements* ASSUME *and* ASSERT. *Moreover, we include the C-like commands for memory (de-)allocation* ALLOC *and* FREE.

Functions have a function name f, a list of argument variables $a_1, \ldots, a_n$, $n \geq 0$, and a function body $body_f$ that consists of a *control flow graph*. We represent *control flow graphs* (CFGs) as a tuple $(V, E, entry_f, exit_f)$ such that V is a set of program locations with $entry_f, exit_f \in V$, and $E \subseteq V \times stmt \times V$ is a set of edges between program locations labeled with program statements. A *trace* t of a CFG is an alternating sequence $[v_0, st_1, v_1, \ldots, st_n, v_n]$, $n \geq 0$, of vertices $v_i \in V$ and statements $st_{i+1} \in stmt$ following edges $(v_i, st_{i+1}, v_{i+1}) \in E$.

Note that CFGs as defined here can be used to model arbitrary branching and looping constructs (such as `if-then-else` and `while`). We will further assume:

1. Each function f is either loop-free or consists of a single loop such that the loop header is $entry_f$; i.e., we require that the CFG of f is either acyclic or all back-edges of f (the edges returning to a loop header) return to $entry_f$. This assumption is w.l.o.g, as loops that are embedded in a bigger context can be represented by calls to a function whose body is precisely the loop.[8]
2. The programs do not contain (mutually) recursive functions.
3. Each vertex $v \in V$ has at most two outgoing edges in E.

Definition 2 (Program Configuration). *A program configuration* $cnf \in Config$ *is either a pair* (S, H) *consisting of a stack* S *and a heap* H *or the dedicated err configuration. The stack* $S : Var \rightharpoonup Val$ *is a partial map from variables to values. The heap* $H : (Loc \times Fld) \rightharpoonup_{fin} Val$ *partially maps finitely many pairs of memory locations and field names into values.*

The semantics of the programming language is standard (see [29, Appendix A] for more details). We use the notation $(cnf_1, st) \rightsquigarrow cnf_2$ to denote that a program reaches a configuration cnf_2 from a configuration cnf_1 by executing a statement *st*. The semantics of traces is defined as the transitive closure $\rightsquigarrow^*$ with regard to the statements in the trace ($cnf_i \in Config$):

$$\begin{aligned} (cnf_1, [v_0]) &\rightsquigarrow^* cnf_1, \\ (cnf_1, [t, v_1, st_2, v_2]) &\rightsquigarrow^* cnf_3 \text{ if } (v_1, st_2, v_2) \in E \wedge (cnf_1, [t, v_1]) \rightsquigarrow^* cnf_2 \\ &\quad \wedge (cnf_2, st_2) \rightsquigarrow cnf_3. \end{aligned}$$

3.2 Separation Logic

Next, we introduce a simple separation logic (*SL*) fragment that is suited for biabduction-based shape analysis. Even though most shape analyses in recent literature are based on more sophisticated fragments, this simple fragment suffices to define our central contributions, which can be easily lifted to more powerful separation logic fragments as well (indeed, we use a more expressive fragment in our later presented experiments). The formulas of *SL* are based on symbolic heaps [1].

We differentiate program variables in *PVar* that occur in the program (and are uniquely identifiable for each function) and logical variables in *LVar* that only occur in *SL* formulas. We enforce a *normal form* [19,13] where program variables only occur once per formula (as part of Π_P in Figure 3), such that they are uniquely identified by a single expression at any point. Further, the dedicated logical variable $return_f$ denotes the return value of a function f, if any.

Definition 3 (Separation Logic). *The* separation logic fragment SL *contains the standard connectives of separation logic and an inductive predicate ls denoting a singly-linked list segment. Figure 3 shows the full syntax of* SL. *Symbolic heaps* φ *distinguish between spatial* (Σ) *and pure parts* (Π), *as well as program variable equalities* Π_P, *and are combined disjunctively* (Δ).

[8] More complex cyclic control flow, e.g. describing the common `break`/`continue`/`goto` statements can also be emulated by introducing auxiliary out parameters which are then used to guide the control flow.

Based on this, we define *contracts* for functions in the programming language to be pairs of formulas from *SL* where we call the parts of the pair a *pre-condition* and a *post-condition*, respectively, and denote them as $c.pre$ and $c.post$ for a contract c. Basic contracts for all statements in the language defined in *SL* can be found in [29, Appendix B]. Formulas from *SL* are evaluated against program configurations with the judgement $\vDash$. We note that we chose the standard semantics for $*$ and $\mapsto$. Furthermore, we also use *SL*'s *entailment* judgement, written $P \vdash Q$.

$$
\begin{aligned}
\varepsilon &:= NULL \mid k \in \mathbb{N} \mid \ell \in LVar \mid \mathsf{unop}\ \varepsilon \\
&\quad \mid \varepsilon_1\ \mathsf{binop}\ \varepsilon_2 \mid \top \\
\oplus &:= =\mid\neq\mid\leq\mid\geq\mid<\mid> \\
\Sigma &:= x.f \mapsto \varepsilon \mid \Sigma_1 * \Sigma_2 \mid ls(x,\varepsilon) \mid \mathsf{emp} \\
\Pi &:= \Pi_1 \wedge \Pi_2 \mid \mathsf{true} \mid \varepsilon_1 \oplus \varepsilon_2 \\
\Pi_P &:= \bigwedge_{x \in PVar} x = \varepsilon \quad \varphi := \Pi_P \wedge \Pi : \Sigma \\
\Delta &:= \varphi \vee \Delta \mid \varphi
\end{aligned}
$$

Fig. 3: The syntax of *SL*

We denote with $P[x/y]$ the formula P with variable $y \in LVar$ substituted with x or, if $y \in PVar$, the formula updated with $y = x$ as part of Π_P, respectively. We often denote a formula $\Pi_P \wedge \Pi : \Sigma$ by only $\Pi_P \wedge \Pi$ or $\Pi_P : \Sigma$ if $\Sigma = \mathsf{emp}$ or $\Pi = \mathsf{true}$, respectively. Also, we denote the composition of formulas $\varphi_1 = \Pi_P \wedge \Pi_1 : \Sigma_1$ and $\varphi_2 = \Pi_P \wedge \Pi_2 : \Sigma_2$ as $\varphi_1 * \varphi_2 := \Pi_P \wedge \Pi_1 \wedge \Pi_2 : \Sigma_1 * \Sigma_2$.

Definition 4 (Abstraction). *An abstraction function $\alpha : SL \to SL$ takes a formula in SL and returns a potentially different formula such that it abstracts a given formula P such that $P \vdash \alpha(P)$.*

Example 1. An abstraction procedure α as described by [8] abstracts consecutive pointer chains into list segments, e.g., for $PVar = \{x\}$:

$$\alpha(x = a \wedge a.\mathtt{next} \mapsto b * b.\mathtt{next} \mapsto c) \equiv x = a \wedge ls(a, c).$$[9]

3.3 Biabduction-Based Shape Analysis

Definition 5 (Biabduction). *Biabduction is the process of solving a query $P * \boxed{M} \vdash Q * \boxed{F}$ for given SL formulas P and Q by computing an antiframe (or missing part) M and a frame F such that the entailment is valid.*

We are only interested in solutions for M that do not contradict P, as otherwise, the entailment would be trivially valid. A *biabduction procedure* is then an algorithm that, given two formulas, either computes a fitting frame and anti-frame or fails. The steps to compute a frame and anti-frame are called *frame inference* and *abduction*, respectively.

For the sake of saving space, we do not develop a full biabduction procedure here but refer the reader to [6,5,13] for detailed descriptions.

Definition 6 (Analysis States). *An analysis state s is an intermediate contract $(P \parallel Q)$ where $P \in \varphi$ and $Q \in \Delta$. To distinguish these from finished contracts, we call P the candidate pre-condition ($s.pre$) and Q the current post-condition ($s.curr$). In analysis*

[9] As this step loses information about b, it is only applied in contexts in which b is not relevant otherwise. See, e.g., [16]. This is the case here since $b \notin PVar$ and $\nexists y \in PVar.\ \Pi_P \vdash y = b$.

states, each function argument $a_i \in PVar$ is associated with an anchor variable $A_i \in AnchVar \subseteq LVar$ (in upper case) denoting its value at $entry_f$. We omit equalities of the form $x = X$ from Π_P if they are not relevant. These anchor equalities also do not appear in finished contracts.

Biabductive Symbolic Execution Step. Let there be an analysis state $(P \parallel Q)$ at a program location l for a statement st with contract $(L \parallel R)$ and a location l' such that $(l, st, l') \in E$. Then st can be symbolically executed by solving the biabduction query $Q * \boxed{M} \vdash L * \boxed{F}$ resulting in the new analysis state $(P * M \parallel R * F)$. As in [6,5,13], we require that (1) $var(M) \subseteq LVar$ and that (2) $P * M$ is satisfiable. If such an M does not exist, we say that the biabduction *fails*.

Definition 7 (Biabduction-based Shape Analysis). *A basic* biabduction-based shape analysis $A_{B,\alpha}$ *uses a biabduction procedure B and an abstraction procedure α to analyze programs in our programming language. Thereby, it analyzes the functions* bottom-up *along the call tree, starting from its leaves. In each step, the analysis takes an analysis state and symbolically executes the next statement from it by updating the state accordingly. In the case of multiple contracts, the analysis has to determine the applicable ones and continue from each of these.*

Furthermore, the analysis runs for a function f until it reaches a fixed point, i.e., until no new analysis states are computed. A common way to check for this condition is to check whether new analysis states entail already computed ones. To enforce termination, $A_{B,\alpha}$ also applies α to abstract the analysis states at loop heads. Finally, the pairs of candidate pre-conditions and current post-conditions forming the analysis states that reached $exit_f$ become its contracts.

We now fix an arbitrary, but correct biabduction-based shape analysis $A_{B,\alpha}$, which we extend in the following sections.

Definition 8 (Soundness of Analysis States). *An analysis state $s = (P \parallel Q)$ is called sound for a trace t, written as the Hoare triple $\{P\}\ t\ \{Q\}$, iff*

$$\forall cnf, cnf' \in Config.\ cnf \vDash P \wedge (cnf, t) \rightsquigarrow^* cnf' \implies cnf' \neq err \wedge cnf' \vDash Q.$$

Similarly, a function contract $c = (P, Q)$ is sound for $body_f$, written $\{P\}\ body_f\ \{Q\}$, iff $\{P\}\ t\ \{Q\}$ holds for all traces $t = [entry_f, \ldots, exit_f]$ through $body_f$.

The *initial analysis state* s_0 for function f has $s_0.pre = \texttt{true}$, $s_0.curr = \bigwedge\{x = X \mid x \in PVar \wedge X \in AnchVar\}$, which denotes that each program variable has a fixed but initially unrestricted value (anchor) at the start of f.

Handling of ASSUME. Following the seminal work [6,5] and the more recent [13], we define biabductive shape analysis to split its states at branching points according to the branching condition. As the literature contains sufficient explanations of this mechanism (called assume-as-assume and assume-as-assert), we only give a brief intuition here. If the branching condition can be expressed in terms of the function arguments, i.e., if the branch taken can be statically determined purely from the function arguments, the analysis includes the two cases into the pre-conditions of the resulting states. This

treatment is equivalent to handling the branching condition's ASSUME statements as if they were ASSERT statements instead. Otherwise, the analysis states for the branches have the same pre-condition, and the branching condition cases are only added to the corresponding post-conditions.

4 Sound Branching Analysis with Shared Abduction

```
1 int nested(node *x, node *y,
       node *z){
2 if (?) {
3   if (y != NULL) {
4     return y->data;
5   } else {
6     return z->data;
7   }
8 } else {
9   return x->data;
10 } }
```

Listing 4: Nested branching

As the example in Listing 1 is rather convoluted, we introduce the simpler Listing 4 to show how exactly the technique works. There, the function `nested` loads from one of the three pointer arguments, depending on a non-deterministic condition `?` on Line 2 and a deterministic one on Line 3. Regardless of the values of the function arguments, an execution can either take the `then` or the `else` branch of the outer `if-then-else`.

Therefore, the original analysis simply splits the analysis state without abducing any pre-condition. In contrast, the branches of the inner `if-then-else` can be distinguished by whether the argument `y` is initially a null pointer, leading the analysis to abduce different pre-conditions for each branch. Altogether, the classical biabduction-based shape analysis will find three unsound contracts for the function, one for each possible code path, similar to the following:

$$(x.\texttt{data} \mapsto \ell_1 \parallel \mathit{return}_{\mathit{nested}} = \ell_1 : x.\texttt{data} \mapsto \ell_1)$$
$$(y \neq \mathit{NULL} : y.\texttt{data} \mapsto \ell_2 \parallel y \neq \mathit{NULL} \wedge \mathit{return}_{\mathit{nested}} = \ell_2 : y.\texttt{data} \mapsto \ell_2)$$
$$(y = \mathit{NULL} : z.\texttt{data} \mapsto \ell_3 \parallel y = \mathit{NULL} \wedge \mathit{return}_{\mathit{nested}} = \ell_3 : z.\texttt{data} \mapsto \ell_3)$$

As introduced in Section 2.1, our new technique overcomes this unsoundness issue and shares requirements abduced with related analysis states. To guarantee that the requirements are only shared with actually related analysis states, we introduce so-called *extended analysis states* or *worlds* for short.

Definition 9 (Worlds). Worlds *comprise a shared pre-condition* P *and multiple current post-conditions* $Q_i^{l_i}$ *at possibly different program locations* l_i*:* $(P \parallel Q_0^{l_0} \vee \cdots \vee Q_n^{l_n})$

We stress the seemingly small but crucial difference between the current post-conditions used in our notion of worlds and the previously defined abstract states: the latter are, in general, also allowed to use disjunctions but are missing the labeling by program locations (allowing the disjuncts to be associated with different program paths). Moreover, worlds do not require the logic itself to contain disjunctions but merely simulates them with its structure.

Definition 10 (Soundness of Worlds). *A world* $w = (P \parallel Q_0^{l_0} \vee \cdots \vee Q_n^{l_n})$ *is sound for a trace* $t = [v_0, \ldots, v_n]$*, written* $\{P\}\ t\ \{Q_0^{l_0} \vee \cdots \vee Q_n^{l_n}\}$*, iff* $\forall \mathit{conf}, \mathit{conf}' \in \mathit{Config}.\ \mathit{conf} \vDash P \wedge (\mathit{conf}, t) \rightsquigarrow^* \mathit{conf}' \implies \mathit{conf}' \neq \mathit{err} \wedge \exists i.\ l_i = v_n \wedge \mathit{conf}' \vDash Q_i.$

Definition 11 (Shared Abduction). *If the analysis finds a non-empty anti-frame for any of the world's current post-conditions Q_i, it is added to the shared pre-condition P and to all other current post-conditions. We call this shared abduction. This step is motivated by the frame rule of separation logic and works as follows: If M and F are the solution to the biabduction query $Q_i * \boxed{M} \vdash L * \boxed{F}$ where $(L \parallel R)$ is the contract of the statement st that is the label of the edge $(l_i, st, l_{i'})$, then the world $(P \parallel Q_0^{l_0} \vee \cdots \vee Q_i^{l_i} \vee \cdots \vee Q_n^{l_n})$ gets updated to:*

$$\left(P * M \parallel (Q_0 * M)^{l_0} \vee \cdots \vee (Q_i * M)^{l_i} \vee \cdots \vee (Q_n * M)^{l_n} \vee (F * R)^{l_{i'}}\right).$$

Analysis with Worlds. Whereas analysis states can be split at branching statements by simply duplicating them and adding the respective assumptions, world splits need to be treated differently. The two branches must share their abduced pre-conditions if the branch taken cannot be determined from the initial program state. Therefore, such a branching point with condition c at a location l_i with two successor locations l_j and l_k for a current post-condition Q_i leads to transforming the world from $(P \parallel Q_0^{l_0} \vee \cdots \vee Q_i^{l_i} \vee \cdots \vee Q_n^{l_n})$ to $(P \parallel Q_0^{l_0} \vee \cdots \vee Q_i^{l_i} \vee \cdots \vee Q_n^{l_n} \vee (Q_i \wedge c)^{l_j} \vee (Q_i \wedge \neg c)^{l_k})$, where two new post-conditions are added to the world.

In contrast, if the branch can be determined from the initial program state, the whole world must be split into two to ensure shared abduction works correctly. This means that the world at the branching point is exchanged with two new worlds:

$$(P \wedge c \parallel (Q_0 \wedge c)^{l_0} \vee \cdots \vee (Q_i \wedge c)^{l_i} \vee \cdots \vee (Q_n \wedge c)^{l_n} \vee (Q_i \wedge c)^{l_j}),$$
$$(P \wedge \neg c \parallel (Q_0 \wedge \neg c)^{l_0} \vee \cdots \vee (Q_i \wedge \neg c)^{l_i} \vee \cdots \vee (Q_n \wedge \neg c)^{l_n} \vee (Q_i \wedge \neg c)^{l_k}).$$

Theorem 1 (Loop-free Soundness with Worlds). *Let $A_{B,\alpha}$ return only sound contracts for functions without branching. Further, let $A'_{B,\alpha}$ be the biabduction-based shape analysis obtained by extending $A_{B,\alpha}$ to use worlds as its analysis states and to apply shared abduction. Then, the contracts computed by $A'_{B,\alpha}$ for loop-free functions are sound. **Proof**: See [29, Appendix D.1].*

Example 2. With these ideas, the function in Listing 4 can be analyzed as follows. At the start of the function, the world is equivalent to an initial analysis state:

$$(\mathsf{true} \parallel (x = X \wedge y = Y \wedge z = Z)_0^1)$$

We denote program locations with their respective lines in the listing and only show the current post-conditions with the highest line number for each branch. Furthermore, we add subscripts to identify the different current post-conditions and worlds uniquely. At the outer `if-then-else`, the current post-condition is split into two as the branching condition cannot be related to the function arguments due to non-determinism. We further ignore the condition in the formula as it has no further relevance either way.

$$(\mathsf{true} \parallel (x = X \wedge y = Y \wedge z = Z)_0^3 \vee (x = X \wedge y = Y \wedge z = Z)_1^9)$$

If the analysis chooses w.l.o.g. to first proceed with post-condition 1, it will abduce that $X.\mathtt{data}$ needs to be allocated and share this information with the rest of the world:

$$(X.\mathtt{data} \mapsto \ell_1 \parallel (x = X \wedge y = Y \wedge z = Z : X.\mathtt{data} \mapsto \ell_1)_0^3$$
$$\vee (x = X \wedge y = Y \wedge z = Z \wedge \mathit{return}_{nested} = \ell_1 : X.\mathtt{data} \mapsto \ell_1)_1^{11})$$

Thus, the current post-condition in the `then` branch now also requires as a pre-condition that $X.\texttt{data}$ is allocated and will not be unsound due to missing this information. The analysis can then choose to proceed with the current post-condition 0 and find that it can relate the branching condition with the function arguments. Therefore, the world needs to be split, as the two cases of condition are expressed as part of the world's pre-condition. To be more precise, the world is split based on whether Y is $NULL$ (with omitted anchor equalities):

$$\begin{aligned}(Y \neq NULL : &X.\texttt{data} \mapsto \ell_1 \parallel (\cdots \wedge Y \neq NULL : X.\texttt{data} \mapsto \ell_1)_0^4 \\ &\vee (\cdots \wedge return_{nested} = \ell_1 \wedge Y \neq NULL : X.\texttt{data} \mapsto \ell_1)_1^{11})_0, \\ (Y = NULL : &X.\texttt{data} \mapsto \ell_1 \parallel (\cdots \wedge Y = NULL : X.\texttt{data} \mapsto \ell_1)_0^6 \\ &\vee (\cdots \wedge return_{nested} = \ell_1 \wedge Y = NULL : X.\texttt{data} \mapsto \ell_1)_1^{11})_1.\end{aligned}$$

The two worlds will then abduce different required pre-conditions in further steps and finally result in the following (simplified) contracts for the function `nested`:

$$\begin{aligned}(y \neq NULL : &x.\texttt{data} \mapsto \ell_1 * y.\texttt{data} \mapsto \ell_2 \parallel \\ &(y \neq NULL \wedge return_{nested} = \ell_2 : x.\texttt{data} \mapsto \ell_1 * y.\texttt{data} \mapsto \ell_2) \\ &\vee (y \neq NULL \wedge return_{nested} = \ell_1 : x.\texttt{data} \mapsto \ell_1 * y.\texttt{data} \mapsto \ell_2)), \\ (y = NULL : &x.\texttt{data} \mapsto \ell_1 * z.\texttt{data} \mapsto \ell_3 \parallel \\ &(y = NULL \wedge return_{nested} = \ell_3 : x.\texttt{data} \mapsto \ell_1 * z.\texttt{data} \mapsto \ell_3) \\ &\vee (y = NULL \wedge return_{nested} = \ell_1 : x.\texttt{data} \mapsto \ell_1 * z.\texttt{data} \mapsto \ell_3)).\end{aligned}$$

4.1 Comparison with Disjunctive Domains

It may be tempting to consider shared abduction with worlds to be just a disjunctive closure of conjunctive formulas used commonly in various abstract interpretation approaches. However, when using a disjunctive closure, the symbolic execution is typically performed independently for each disjunct, perhaps followed by attempts to join some of the disjuncts or to prune them away using entailment checks—as done in [10,6,13]. In contrast, our analysis with worlds differs in that (1) the worlds are, in fact, not purely disjunctive due to a single precondition shared by all current post-conditions in a world and due to working with sets of worlds, (2) state splits either result in two new post-conditions or two new worlds, and (3) the symbolic execution from a single disjunct can influence all other disjuncts in the same world via shared abduction.

5 Biabductive Loop Acceleration via Shape Extrapolation

We first introduce the central steps of our technique for a simplified setting. In this setting, loops only have loop conditions of the form $x \neq NULL$ where x is a function parameter. Furthermore, we assume that loops do not contain branching. We will show how to lift these restrictions in Section 5.2.

```
void free_list(node *x) {
  while (x != NULL) {
    node *aux = x;
    x = x->next;
    free(aux);
} }
```

Listing 5: Deallocating a list

We will explain the steps of our biabductive loop acceleration with the help of the example in Listing 5, which falls into the fragment of programs allowed in the simplified setting. The example shows a simple loop that frees a given list node by node. As such, the expected contract would be $(ls(x, NULL) \parallel \mathsf{emp})$.

5.1 Basic Biabductive Loop Acceleration

Whereas Procedure 1 describes biabductive loop acceleration on a high level, the following paragraphs describe the main steps of the procedure in more detail.

Procedure 1 Biabductive loop acceleration

Input: A function f consisting of a loop l with body $body_l$ and exit condition e_l
Output: A sound contract c for f or FAILURE

$s_0.pre \leftarrow \mathsf{true}, \quad s_0.curr \leftarrow \bigwedge\{x = X \mid x \in PVar\}$

Compute $s_1 \leftarrow A_{B,\alpha}(body_l, s_0)$
$(\tau_{pre} * \rho_{pre} \parallel \tau_{curr} * \rho_{curr}) \leftarrow$ PARTITION(s_1)

$\mathcal{P}, \mathcal{Q} \leftarrow$ SHAPEEXTRAPOLATION$(\tau_{pre}, \tau_{curr})$
Construct s_{inv} from $\mathcal{P}$, $\mathcal{Q}$, ρ_{pre}, and ρ_{curr}
$s_2 \leftarrow A_{B,\alpha}(body_l, s_{inv})$
Check that $s_2.curr \vdash s_{inv}.curr$

Construct s_{final} from $\mathcal{P}$, $\mathcal{Q}$, ρ_{pre}, and ρ_{curr}
return $c \leftarrow s_{final}$

Our algorithm first analyzes a single loop iteration starting from the initial analysis state s_0. If this analysis run ends in a state s_1, the algorithm then continues by determining which parts of s_1 describe the shape of the traversed data structure, i.e., the traversed singly-linked lists in our simplified setting. To this end, the algorithm partitions the candidate precondition as well as the current post-condition of the state s_1 into subformulas $\tau_{pre/curr}$ and $\rho_{pre/curr}$ such that the τ formulas contain the *transformed*[10] parts of the state that should be related to the shape, whereas the *remaining* parts of the state are collected in the subformulas ρ, which comprise both completely unchanged predicates as well as changed memory locations that are not part of the shape. This separation is done for both the pre- and current post-condition of the state s_1 to capture changes to the shape of the data structure. Some more technical details of the partitioning, which are not needed now, will be presented in Section 5.3.

Example 3. For Listing 5, the analysis finds the state $s_1 := (X.\texttt{next} \mapsto \ell_1 \parallel x = \ell_1)$ after one loop iteration. There, the partition of s_1 is trivially $\tau_{pre} := s_1.pre$ and $\tau_{curr} := s_1.curr$ as this simple loop does not affect anything except the traversed list. On the other hand, the inner loop of Listing 2 does not change the shape of the traversed list but accesses and changes further parts of the program state. As a result, the partitions are $\tau_{pre} = \tau_{curr} := I.\texttt{next} \mapsto \ell_1 * I.\texttt{elem} \mapsto \ell_2$ and $i = I$ or $i = \ell_1$, respectively, for the predicates that relate to the list and $\rho_{pre} = o.\texttt{wgt} \mapsto \ell_3 * sum \mapsto \ell_4$ and $\rho_{curr} = o.\texttt{wgt} \mapsto \ell_3 * sum \mapsto (\ell_4 + \ell_3 \cdot \ell_2)$ for the ones relating to the context.

The main step of our procedure is the construction of the candidate loop invariant s_{inv}. For that, we first need to find an abstract description of the shape of the involved data structures. The abstraction must satisfy specific properties described below that are needed to ensure the soundness of the approach. We call this step *shape extrapolation* and provide a minimum viable heuristic implementing it in Section 5.3. However, we stress that this algorithm can be changed as long as the properties in Figure 4 hold.

[10] Here "transformed" means the changed value of the loop variable x and the shape that is described in between the old and the new value of x.

In general, we need shape extrapolation to abstract the two subformulas τ_{pre} and τ_{curr} to list-segment shapes $\mathcal{P}(X, x)$ and $\mathcal{Q}(X, x)$ where the parameter X denotes the first node of the list segment and x denotes the current position in the segment; we omit the parameters X and x and simply write $\mathcal{P}$ and $\mathcal{Q}$ when there is no danger of confusion. We require $\mathcal{P}$ and $\mathcal{Q}$ to satisfy the three properties given in Figure 4. These conditions are generalized in Figure 5, and the proof of soundness (see [29, Appendix D.2]) shows that they allow to establish a loop invariant.

1. $\tau_{pre} \vdash \mathcal{P}$ and $\tau_{curr} \vdash \mathcal{Q}$,
2. $\mathcal{Q} \wedge X = x \vdash \mathsf{emp}$,
3. $\mathcal{P}[a/X, b/x] * \mathcal{P}[b/X, c/x] \vdash \mathcal{P}[a/X, c/x]$

Fig. 4: Extrapolation properties

Intuitively, Point (1) of Figure 4 simply ensures that $\mathcal{P}$ and $\mathcal{Q}$ are actual abstractions of τ_{pre} and τ_{curr}, respectively. In addition, Property (2) ensures that $\mathcal{Q}$ only describes the so-far traversed and transformed part of the list. Thereby, if $X = x$, i.e., at the start of the loop, the so-far traversed and transformed part of the list must be empty. Finally, (3) states that consecutive list segments may always be composed into one list segment.

Example 4. In the example in Listing 5, the shape τ_{pre} obtained after one loop iteration is extrapolated (see Procedure 2) to the formula $ls(X, \ell_1)$. Since $x = \ell_1$, the shape $\mathcal{P}$ becomes $ls(X, x)$. On the other hand, $\tau_{curr} = \mathsf{emp}$ does not contain any spatial predicates, and so the extrapolation produces emp as $\mathcal{Q}$, since the transformation of the list consists in deleting it – if the list was just traversed, we would obtain $ls(X, x)$. It is easy to verify that all properties of Figure 4 are satisfied.

In contrast to previous analyses, we explicitly construct a candidate loop invariant from the abstract shapes $\mathcal{P}$ and $\mathcal{Q}$ and subsequently check whether it is sound. The candidate loop invariant s_{inv} is meant to describe an intermediate state of the loop:

$$s_{inv} := (\rho_{pre} * \mathcal{P} * \mathcal{P}[x/X, NULL/x] \parallel \rho_{curr} * \mathcal{Q} * \mathcal{P}[x/X, NULL/x]) .$$

The pre-condition of this state contains two (sub-)shapes $\mathcal{P}$ and $\mathcal{P}[x/X, NULL/x]$ where the first describes the already traversed list segment starting in X and ending in the current value of x, whereas the latter denotes the not yet traversed part of the list starting at x and ending in $NULL$. Due to the extrapolation Property (3), the two subshapes together form the full extrapolated shape $ls(X, NULL)$. In contrast, the post-condition also takes into account the effects of the loop on the already traversed list segment and, thus, contains $\mathcal{Q}$ instead of $\mathcal{P}$.

To prove that s_{inv} is a loop invariant, the analysis also needs to check whether the post-condition's memory footprint is sufficient for another loop iteration and whether it also holds after this iteration. This is proven by analyzing another loop iteration starting from s_{inv} in which the abduction of new pre-condition predicates is disallowed, thus forcing the analysis to fail if the shapes describe an insufficient memory footprint.

Suppose the invariant checking step successfully finishes the symbolic execution of the loop body in some state s_2. In that case, this implies that the loop body can be safely executed from the state s_{inv}. Next, we check whether $s_2.curr \vdash s_{inv}.curr$, i.e. whether s_{inv} is actually a loop invariant. If the check succeeds, the shapes are sound for all loop iterations, and the loop acceleration procedure can continue with the final step.

Example 5. It trivially holds that the following state is invariant for the loop in Listing 5, i.e., it is sound before and after each loop iteration.

$$s_{inv} := (ls(X, \ell_1) * ls(\ell_1, NULL) \parallel x = \ell_1 : ls(\ell_1, NULL))$$

As the state depicts the program at an arbitrary point of the iteration, it contains both the already traversed shape $ls(X, \ell_1)$ in the pre-condition (which has been freed in the post-condition) and the unchanged, still-to-traverse shape $ls(\ell_1, NULL)$.

Finalizing. Lastly, the loop analysis is finalized by constructing the final state reached after finishing the loop from the shapes $\mathcal{P}$ and $\mathcal{Q}$ as

$$s_{final} := (\rho_{pre} * \mathcal{P}[NULL/x] \parallel x = NULL * \rho_{curr} * \mathcal{Q} * \mathcal{P}[x/X, NULL/x])$$

This state is simply obtained from s_{inv} by adding the negated loop condition $x = NULL$ and using extrapolation Property (3) to simplify the pre-condition. If the extrapolated shape additionally satisfies the property $\mathcal{P}(x/X, NULL/x) \wedge x = NULL \vdash \mathsf{emp}$ (we call this Property (3.5)), which is natural for traversing linked lists until the end, the final state can be simplified even further.

Example 6. For the loop in Listing 5, the freed list is represented by the shape $\mathcal{Q}$ being empty, making the final state $s_{final} := (ls(X, NULL) \parallel x = NULL * ls(x, NULL))$. Since the list segment to $NULL$ satisfies Property (3.5), we obtain the expected final state. In addition, this state is also the contract of the function `free_list`, and thus the analysis reaches its end for this function.

5.2 Lifting Restrictions on Biabductive Loop Acceleration

We now explore how the restrictions introduced above can be lifted to make biabductive loop acceleration more applicable in practice. We write $\overline{x}$ for an ordered list of elements x_i with $0 \leq i \leq n$ for some n. We denote by $f(\overline{x})$ the ordered list $\overline{y}$ where $y_i = f(x_i)$ for $0 \leq i \leq n$.

```
void either_list(node *x) {
  node *head = x;
  while (x != NULL
    && x->next != head)
  {...} }
```

Listing 6: Cyclic/to-null lists

Extension: General Loop Conditions. The first restriction that we lift concerns the loop condition. We assume that the loop condition e_l is of the form $e_l = \bigwedge_i e_i$ with single atomic conjuncts e_i of arbitrary form. An example of such a loop can be seen in Listing 6, which handles both cyclic and null-terminated lists equally. Handling such a more general loop condition requires further adjustments to the loop acceleration procedure. To be able to express multiple exit conditions that relate to multiple different variables, the algorithm needs to be able to describe the traversed shape relative to these variables. Hence, the shapes $\mathcal{P}$ and $\mathcal{Q}$ are now parameterized over all program variables changed throughout the loop—namely, all variables x for which $s_1.curr.\Pi \not\vdash x = X$. We call the set of these variables χ and re-define the $\mathcal{P}$ and $\mathcal{Q}$ shapes as $\mathcal{P}(\overline{X}, \overline{x})$ and $\mathcal{Q}(\overline{X}, \overline{x})$, respectively, where $\overline{x}$ is the ordered list of the variables from χ that occur in $\mathcal{P}$ and $\mathcal{Q}$, and $\overline{X}$ is the ordered list of the corresponding anchor variables. Below, we will use $\mathcal{P}(\overline{a}, \overline{b})$ to

denote the predicate $\mathcal{P}(\overline{X}, \overline{x})[\overline{a}/\overline{X}, \overline{b}/\overline{x}]$, i.e., the predicate obtained from $\mathcal{P}(\overline{X}, \overline{x})$ by simultaneously substituting the variables $\overline{X}$ with $\overline{a}$, and $\overline{x}$ with $\overline{b}$ (we will use the same notation for $\mathcal{Q}$). Note that for lists, this is equal to setting the two parameters of the list segment predicate to a and b, respectively.

With this notation, we re-define the properties of extrapolation to consider the new parameters in Figure 5. We further define a mapping $\text{EXIT}(x)$ of variables $x \in \chi$ to the values they can have at a loop exit. As these values can be challenging to determine from the loop condition alone, we restrict the map to hold only logically constant values, i.e., $NULL$ or other program variables outside of χ (as their values stay constant throughout the loop), and define the other entries to map to fresh logical variables instead.

1. $\tau_{pre} \vdash \mathcal{P}(\overline{X}, \overline{x}) \land \tau_{curr} \vdash \mathcal{Q}(\overline{X}, \overline{x})$,
2. $\mathcal{Q}(\overline{X}, \overline{x}) \land \bigwedge_{x \in \chi} X = x \vdash \mathsf{emp}$,
3. $\mathcal{P}(\overline{a}, \overline{b}) * \mathcal{P}(\overline{b}, \overline{c}) \vdash \mathcal{P}(\overline{a}, \overline{c})$.

Fig. 5: General extrapolation properties

Furthermore, the post-condition of the final state has to encode that any of the loop conditions can be unsatisfied for the program to leave the loop. This is done by taking the disjunction of the previous final state post-condition combined with one dissatisfied loop condition (note that the disjunction represents the world's current post-conditions):

$$s_{final}.curr := \bigvee_i \left(\neg e_i * \rho_{curr} * \mathcal{Q}(\overline{X}, \overline{x}) * \mathcal{P}(\overline{x}, \text{EXIT}(\overline{x}))\right).$$

Extension: Branching Loop Body. Branching in loop bodies can be handled by collecting all states s_1 after the first loop iteration analysis, extrapolating their shapes, and combining them if possible into one compound shape via a join operation akin to the ones described in [10] or [26]. Because such an operation is mostly orthogonal to the central ideas of shape extrapolation, we refer to the literature for more details.

Extension: Overlapping Shape Changes. The extension to allow for more general loop conditions can lead to problems with shape extrapolation if the involved shapes overlap, i.e., if the new and old memory locations to which program variables point to are the same. This can, e.g., happen if a list is reversed (see [29, Appendix F.2]). Such cases can be detected if the new value of a program variable in χ is the anchor of another variable. In the example of list reversal, the program variable tracking the reversed list will be set to the initial value of the original list as that list's first node becomes the last node in the reversed one. Such an overlap would cause problems in the implementation of SHAPEEXTRAPOLATION presented as Procedure 2. To circumvent this problem, the analysis symbolically executes further additional loop iterations to find a program state in which there is no overlap anymore, and only then performs the extrapolation.

Extension: Further Loop Effects. As depicted in Listing 2, loops can not only traverse data structures but also change the program state in arbitrary ways. In such cases, the candidate invariant s_{inv} might not be an actual invariant, i.e. $s_2.curr \not\vdash s_{inv}.curr$. To handle such cases, we apply a join in the corresponding pure value domain of the analysis. In the simplest case, this step exchanges the values of variables and memory locations that are the cause of $s_2 \not\vdash s_{inv}$ with the value $\top$. In the example Listing 2, the value stored at `sum` after the first iteration is $\ell_{sum} + \ell_1 \cdot \ell_w$, resulting in the points-to predicate $sum \mapsto \ell_{sum} + \ell_1 \cdot \ell_w$ being a part of s_{inv}. After the second iteration, the

predicate changes to $sum \mapsto \ell_{sum} + \ell_1 \cdot \ell_w + \ell_2 \cdot \ell_w$, which does not entail its counterpart in s_{inv}. However, by joining the two values of the memory location to $sum \mapsto \top$, the entailment is ensured. The same problem actually occurs if the initial value of a variable does not entail its representation in the invariant, e.g., because it is set to a constant in the loop (see [29, Appendix F.4]). In this case, we also need to abstract the variable's values in $s_{inv}.curr$ to $\top$, thus guaranteeing that s_{inv} also holds before the first iteration.

Theorem 2 (Soundness of Shape Extrapolation). *Let $A_{B,\alpha}$ compute only sound contracts for loop-free functions. If Procedure 1 uses $A_{B,\alpha}$, then Procedure 1 with all extensions described in this section applied to a loop l either fails or returns a contract (P, Q) such that $\{P\}\ l\ \{Q\}$.* ***Proof****: See [29, Appendix D.2].*

5.3 Shape Extrapolation

We now propose a concrete shape extrapolation procedure based on the principles presented above. This procedure is supposed to be the easiest possible heuristic that suffices to find reasonable loop invaraiants. To this end, it follows the original idea of obtaining inductive shapes through abstraction, but in a "smarter" way.

The initial partitioning is one of the most crucial steps for our shape extrapolation procedure. The τ formulas are built by collecting all predicates that describe the shape traversed, i.e., the shape between the anchors and the new values of the variables in χ. These shapes contain all transitively reachable predicates, where reachability is defined spatially. Thereby, a predicate is reachable if there exists a sequence of points-to and list predicates that pairwise overlap in their source/drain variables, modulo variable equalities. For example, if $x \in \chi$, then $\ell_1 = \ell_4 : X \mapsto \ell_1 * ls(\ell_1, \ell_2) * \ell_4.data \mapsto \ell_3$ contains only predicates reachable from the anchor X.

In addition to the partitioning of variables, our concrete shape extrapolation algorithm also needs to know the new value of the variables in χ. We encapsulate this information in the *transformation map* μ which maps $x \in \chi$ to $\ell_x \in LVar$ such that $s_1.curr.\Pi_P \vdash x = \ell_x$. Recall that, due to the normal form of *SL*, every program variable only occurs in a single equality such as $x = \ell_1$, and so μ can be computed by simply comparing their values before and after the loop.

Procedure 2 gives a detailed description of our concrete shape extrapolation procedure. It computes $\mathcal{P}$ (and $\mathcal{Q}$) by first extrapolating the corresponding τ_i into two copies τ_i^1 and τ_i^2. These two copies are supposed to represent the shape accessed by two consecutive loop iterations via an intermediate, fresh auxiliary location. The resulting formulas are then combined via separating conjunctions and abstracted by the abstraction function α to form the abstract shapes θ, which in turn get parameterized by renaming schemas to the final $\mathcal{P}$ and $\mathcal{Q}$. The use of two copies is a heuristic that has proven to be reliable in making the abstraction find better abstract shapes. Note that, in Procedure 2, we omit additional renamings of logical

Procedure 2 ShapeExtrapolation

Input: τ_{pre}, τ_{curr}
Output: Parametric extrapolated shapes $\mathcal{P}$ and $\mathcal{Q}$
Compute the maps χ and μ as described in the text
// $i \in \{pre, curr\}$, x' fresh
$\tau_i^1 \leftarrow \tau_i[x'/\mu(x) \mid x \in \chi]$
$\tau_i^2 \leftarrow \tau_i[x'/X \mid x \in \chi]$
$\theta_i \leftarrow \alpha(\tau_i^1 * \tau_i^2)$

$\mathcal{P}(\overline{x}_1, \overline{x}_2) \leftarrow \theta_{pre}\left[\overline{x}_1/\overline{X}, \overline{x}_2/\mu(\overline{x})\right]$
$\mathcal{Q}(\overline{x}_1, \overline{x}_2) \leftarrow \theta_{curr}\left[\overline{x}_1/\overline{X}, \overline{x}_2/\mu(\overline{x})\right]$
return $\mathcal{P}, \mathcal{Q}$

variables for clarity, as these only help to guide the abstraction but do not affect the resulting shapes any further.

Example 7. In the example from Listing 2, the inner loop can be extrapolated as follows: The procedure takes the effect $\tau_{pre} := I.\texttt{elem} \mapsto \ell_1 * I.\texttt{next} \mapsto \ell_2$ from the transformation and introduces the two auxiliary formulas $\tau^1_{pre} := I.\texttt{elem} \mapsto \ell_1 * I.\texttt{next} \mapsto i'$ and $\tau^2_{pre} := i'.\texttt{elem} \mapsto \ell_1 * i'.\texttt{next} \mapsto \ell_2$ where i' is the auxiliary location representing the intermediate value of i. From these formulas, the abstraction then finds the abstract shape $\theta_{pre} := ls(I, i)$. This abstracted shape is then the basis for the extrapolated shape $\mathcal{P}$. Similarly, $\mathcal{Q}$ is computed to be $ls(I, i)$, too.

5.4 Limitations

We note that shape extrapolation is a heuristic, which is sound (see Theorem 2) but inherently incomplete. The extrapolation step can fail if the partitioned information does not suffice to find a reasonable shape, e.g., if the abstraction function cannot find a fitting inductive shape predicate. However, since our shape extrapolation procedure imitates the loop acceleration procedure of the original analysis, it is guaranteed to be applicable for at least the same programs but in a fundamentally sound (and oftentimes faster) way. Furthermore, we note that shape extrapolation is currently limited to list-like data structures that are traversed linearly. List manipulation is, however, by far the most frequent data structure pattern in low-level code, and so we have focused our efforts on this kind of data structures, in accordance with prior work [6,13]. Nonetheless, we believe that shape extrapolation, which is based on the intuition of locality, can be extended toward tree-like data structures in future work.

6 Implementation and Experimental Evaluation

6.1 Prototype Implementation

We have implemented our techniques as a proof-of-concept in the prototype analyzer Broom [13] written in OCaml and call the resulting tool Brush. It is available as an artifact on Zenodo [28]. The original Broom implements a biabduction-based shape analysis with a focus on low-level primitives and byte-precise memory management and is sound for functions without branching [13, see Theorem 3]. However, since Broom is also still a prototype that focuses more on exact handling of complex memory manipulation than on scalability, neither Broom nor Brush are able to handle large-scale code bases yet. Our new techniques, especially shape extrapolation, improve scalability, but Brush still shares most of its code with Broom and is thus not as mature as industrial-strength tools such as Infer.

6.2 Implementation Limitations

As Brush is largely based on the source code of Broom and does not differ much from it apart from our new techniques, they share mostly the same limitations. On the one hand,

neither tool supports recursive functions. Similarly, they can handle neither stack allocations nor switch-case statements. On the other hand, the logic both tools are based on contains only inductive predicates for linked lists with parameters describing the shape of single nodes. Therefore, the tools can, in general, not analyze programs containing other inductive data structures.

Furthermore, we remark that the running times of Broom and Brush are much higher than for comparable tools, which in part is due to the need for precise pointer arithmetic. This precision is achieved, among other things, by calling an SMT solver, which is more costly for simpler cases than using native solvers, such as in Infer.

6.3 Case Study

We have conducted experiments with two research questions in mind: (1) whether Brush can handle new use cases that existing tools cannot handle; (2) whether Brush is also at least as efficient as Broom or would even improve scalability.

Qualitative Experiments. To answer research question (1), we ran all four analyzers on selected examples that are either presented in [13, Table 1], are a part of the test suite for Broom, or are hand-crafted test cases for shared abduction and shape extrapolation. The results of our case study are depicted in Table 1. All test files are included in the accompanying material. We primarily investigated whether the analyzers found the expected bugs and sound contracts or whether they report other spurious errors.

We note that the biabduction-based shape analysis that Infer was based on is deprecated nowadays, and Infer's focus has shifted from over-approximation to under-approx-imation (see [23,18]). Due to this, we not only compare Brush with the release v1.1.0[11] which was also used for comparison in [13], but also with its predecessor tool Abductor.[12] We excluded the second-order biabduction tool S2 from our experiments since it is quite limited and cannot be applied to most of our benchmark programs.[13]

The ten tests from [13, Table 1] are program fragments of 30–200 LOC. Each test consists of a set of library functions, including creation of a linked-list, insertion and deletion of an element from the list. Moreover, 8 of the tests also contain a top-level test harness performing a concrete manipulation of the particular list. There are three types of lists: (i) circular doubly-linked lists, (ii) linux-lists taken from the Linux kernel, and (iii) intrusive lists.[14] The 43 tests from `broom/tests` are regression tests of Broom. Each one is usually up to 10 LOC and tests the analysis of a particular kind of statement.

[11] Available at https://github.com/facebook/infer/releases/tag/v1.1.0.

[12] Available at http://www0.cs.ucl.ac.uk/staff/p.ohearn/abductor.html.

[13] Of the 73 programs in Table 1, the tool reported internal errors for 56 cases while causing segmentation faults for nine further cases. If we only compare the 52 programs without loops, it fails for 43 instances and causes segmentation faults in 4 further cases. The internal errors range from unsupported language features such as pointer arithmetic (3 programs) to linker errors with unknown symbols (13 cases/6 cases without loops) and unsupported type casts (35/30 cases). All of these cases are correctly handled and accepted by standard C compilers as utilized as frontends by Broom and Brush.

[14] Described by [31] and implemented in https://github.com/robbiev/coh-linkedlist.

Table 1: Examples handled correctly (✓) and incorrectly (×) by the analyzers

Class of inputs	# of test cases	Broom	Infer	Abductor	Brush
[13, table 1]	10	10✓/0×	0✓/10×	0✓/10×	10✓/0×
tests from `broom/tests`	47	47✓/0×	34✓/14×	12✓/35×	47✓/0×
`*_branches.c`	2	0✓/2×	0✓/2×	0✓/2×	2✓/0×
`nested_*.c`	3	1✓/2×	3✓/0×	3✓/0×	3✓/0×
`motivation*.c`	3	0✓/3×	0✓/3×	0✓/3×	3✓/0×
`sll*.c`	3	3✓/0×	0✓/3×	0✓/3×	3✓/0×
other	5	3✓/2×	0✓/5×	3✓/2×	5✓/0×
overall	73	64✓/9×	47✓/26×	18✓/55×	73✓/0×

The newly added hand-crafted use cases are small-scale programs (10–70 LOC), which are, however, rather challenging for the existing analyzers. In particular, the `*_branches` test cases contain multiple cases of non-determinable branching, which can lead to the unsoundness described in Section 2.1. The `nested_*` programs contain multiple examples of nested loops and nested lists. The `sll_*` test cases contain whole programs that create, iterate and destroy singly-linked lists. The `motivation` programs are as described in Section 2. Lastly, the other test cases contain programs with more complex list allocation, deallocation, and transformation.

We specifically note that we have not used common benchmark sets such as the SV-COMP memory-safety benchmark, as these consist primarily of closed programs and focus on data structures that cannot be described by the logic of Broom and Brush. Thus, these benchmark sets lie outside the scope of this work, and we have instead used test cases that allow us to evaluate our research questions explicitly.

In the table, we use ✓ to denote that at least one (sound) contract was computed for each function within the particular example and that the expected errors were reported without false positives. On the other hand, we use × to denote that either no contract could be computed (for a function that would have a sound contract) or the respective tool reported a false positive.

We conclude that shared abduction and shape extrapolation enable Brush to work for strictly more programs than Broom or any of the other tools. This is especially important, as these small but challenging test cases are mostly based on realistic iteration patterns that can be found in code bases such as the Linux kernel.

Quantitative Experiments. In another series of experiments, we also evaluated the runtime of Brush versus Broom[15] on test cases that both tools can handle. Some test cases were split to survey the runtime of single, interesting functions without their calling context. All experiments were run ten times on an Intel Core i7-1260P CPU with 32GiB RAM, and we took the mean over all runs. A plot of the results can be found in Figure 6, the raw data is displayed in [29, Appendix E]. The overall means for programs with loops are 1.85s for Brush to 38.2s for Broom, whereas the means for loop-free branching programs are 22.3s for Brush and 24.8s for Broom. The overall means are 7.24s for Brush to 14.0s for Broom. These numbers show that Brush provides a signif-

[15] We used the commit a361d01badf45c420b57158f2e6d738cb45d1dd9 found at https://pajda.fit.vutbr.cz/rogalew/broom with small additional bug fixes.

Table 2: Loop iterations analyzed by the analyzers (Brush/Broom) until a fixed point was reached, as well as how many loops were present in the programs (in brackets)

copy_alloc	dll-as-sll-traverse	reversal	nested_lists2
2/6 (1)	2/6 (1)	3/8 (1)	6/14 (3)
sll-fst-shared	**sll-fst-shared-alloc**	**sll-fst-shared-iter**	**sll**
4/12 (2)	2/8 (1)	2/6 (1)	6/20 (3)
sll-alloc	**sll-shared-sll-after**	**sll-shared-sll-after-alloc**	**overall**
2/6 (1)	6/21 (3)	4/14 (2)	39/121 (19)

icant speedup over Broom. For the examples with branching but without loops, we at least find a tendency towards faster runtime for Brush.

We directly relate the time improvements of Brush with the number loop iterations analyzed. As seen in Table 2, Brush only requires a fraction of loop iterations due to shape extrapolation for all examples from Table 1, that both Broom and Brush can handle and which contain loops. The mean over all examples is 3.55 iterations for Brush and 11 for Broom. We note that in most cases, Brush requires exactly two iterations per loop, which corresponds to the initial analysis and the invariant checking iteration as described in Section 5. Only the case `reversal` (see [29, Appendix F.2]) requires 3 iterations, as it reverses a list such that the new and old shapes partially overlap. As described in the paragraph about *Extension: Overlapping Shape Changes*, the analysis requires an additional iteration to reach a state in which this overlap has been removed.

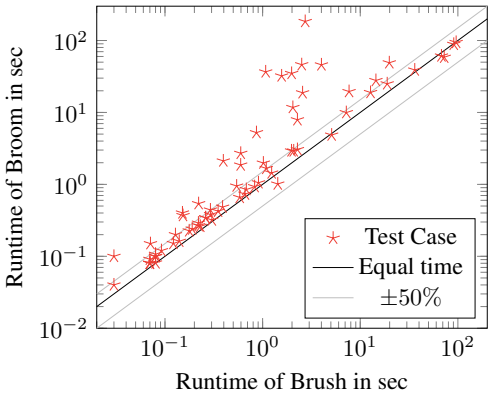

Fig. 6: Runtime of Broom and Brush in sec. for tests from Table 1 that both tools handled

We conclude that shared abduction improves both the precision and the performance of the analysis compared to Broom, while biabductive loop acceleration considerably improves the performance of the analysis for the benchmarks.

7 Related Work

Biabduction-based Shape Analysis. Our work builds on biabduction-based shape analysis introduced by Calcagno et al. [6,5] and later implemented in Infer [4]. Our new techniques avoid the unsoundness issues of the first phase of the analysis and constitute significant theoretical and practical advancements as demonstrated in Sections 4 to 6.

The approach of [6,5] was extended in the Broom analyzer [13] by ways of handling low-level primitives and byte-precise memory handling. Since these extensions are orthogonal to the problems of unsoundness, our techniques are equally effective for improving Broom as depicted in Section 6.

Another related work is the *second-order biabduction* by Le et al. [17]. Their approach does not consider a fixed class of inductive predicates, but discovers them as part of the analysis by instantiating second-order variables with a technique called *shape inference*. The analysis first collects the unknown predicates with corresponding relational assumptions and synthesizes fitting shapes in a second step. Their computation method ensures that these shapes make the resulting contracts sound. Albeit this makes their technique similar to shape extrapolation, the approach of [17] can handle more complex shapes of dynamic data structures. On the other hand, it requires solving the complex predicate inference problem for which their tool uses a relatively simple algorithm. In our experience, this algorithm can easily fail even for programs with simple data structures if they are not compatible with the shape inference procedure. Furthermore, their implementation is rather limited, as described in Section 6.

Very recently, Spies et al. [30] combined biabductive reasoning with auto-active, foundational program verification. Their tool Quiver takes C programs and specification sketches as annotations as input, translates them into a representation in the Caesium C semantics [27], and finally infers and proves full function specifications in the proof assistant Coq. The central reasoning mechanism of Quiver is called *abductive deductive verification*, which is closely related to biabduction. As Quiver does not only work with predicates for memory safety but also with a refinement type system for C, it has a broader focus than our work. However, Quiver requires the user to provide specification sketches, refinement types, and loop invariants, while our work focuses on fully automated, biabductive shape analysis.

Other Shape Analyses. There are many different shape analysis methods not based on biabduction in literature. Of these, the Predator analyzer [10] based on *symbolic memory graphs* is quite successful with regard to the Competition on Software Verification (SV-COMP), see [2,3]. Their approach handles abstraction, entailment, and state pruning as special cases of a general graph joining procedure. They focus on closed programs, and their approach uses function summaries that are computed in a top-down fashion, following the call tree (whereas biabduction-based shape analysis works from the bottom up). This top-down fashion requires a re-analysis of functions for different contexts but can ignore irrelevant code paths. As Predator implements a classic forward analysis that does not compute contracts with pre-conditions, Predator circumvents the problem of unsoundness. On the other hand, Predator only works on closed programs and can thus not be used for modular and incremental analysis.

Another approach to shape analysis has been recently introduced by Illous et al. [15,14]. They utilize *transformers* to describe the effects of functions and compute these with regard to the calling context in a top-down fashion, as in Predator. The transformer-based analysis is built around a transformer abstract domain and is based on abstract interpretation. We note that the use of transformers has partially motivated the inner workings of our shape analysis procedure. The authors noted that biabduction might be applicable for transformers as well, and we strengthen this point by showing how our shape extrapolation procedure combines both ideas to some degree.

We also like to mention the work [9], where the authors consider overlaid data structures. Their technique is based on a fragment of separation logic that differentiates per

object and per field separation. Note that the separation logic fragment we base our work on uses per field separation, while per-object separation is only enforced implicitly.

Other Related Analyses. Recent work about *Incorrectness (Separation) Logic* (ISL) [23,24,18] has introduced a different approach to program analysis. This line of work does not focus on verifying the absence of memory bugs in an over-approximating way but instead tries to find bugs in an under-approximative way. Although the bug-finding ability of incorrectness logics makes them very useful in practice, over-approximating analyses are still relevant for certification and low-level systems software.

Lastly, the emergence of incorrectness logic has also motivated the development of combined logic systems that inherit the benefits of both over- and under-approximating logics. Recent work in this direction includes *Exact Separation Logic* [20] as well as *Outcome (Separation) Logic* [32,33]. In particular, *tri-abduction*, introduced in [33], offers an alternative solution to the branching problem we address with shared abduction. By solving the abduction problem for the pre-conditions required by both branches, the tri-abduction operation can potentially compute better contracts than the (greedy) techniques proposed in this paper, which will first solve the abduction problem for one branch and then for the other. This increased precision, however, comes with the burden of implementing a new operator, whereas we can simply lift existing implementations of biabduction operators to shared abduction. We believe that the increased precision of the triabduction operation is rarely needed, and thus the more lightweight solution of shared abduction is preferable in practice. For our experiments, the precision of shared abduction was sufficient. At the same time, the approach from [33] has not yet been implemented into a tool that could be used for experimental comparison.

8 Conclusion and Future Work

This work introduces the two novel techniques of shared abduction and biabductive loop acceleration with shape extrapolation. We provide soundness proofs for both techniques and implement them in our prototype analyzer Brush, which is based on the state-of-the-art analyzer Broom. We experimentally demonstrate that these techniques enable our biabduction-based shape analysis to find sound contracts in a single analysis phase. In particular, we show that shared abduction and shape extrapolation enable Brush to analyze strictly more programs than Broom or any of the Infer versions, and to considerably improve the performance compared to Broom.

While our work is limited to non-recursive programs, we believe that shape extrapolation can easily be extended to recursive programs (e.g., tree traversals) and that this direction constitutes an exciting avenue for future work. We also hope to incorporate techniques that track the content of data structures, i.e., the data values stored in the data structure. Specifically, we would like to enrich the logic and biabduction procedure to track data values, e.g., for tracking the value of sum in the example of Listing 2. For this, we plan to take inspiration from prior work that combines shape domains and data domains, such as the product domain studied in [11]. Another interesting direction for future research is whether the idea underlying shape extrapolation has application in the synthesis of heap-manipulating programs, e.g., as studied in [21].

Acknowledgments. This work was supported by the Czech Science Foundation project 23-06506S and the FIT BUT internal project FIT-S-23-8151. The work of the Austrian team leading to this result has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101034440. The collaboration of the teams was also partially supported under the project VASSAL: "Verification and Analysis for Safety and Security of Applications in Life" funded by the European Union under Horizon Europe WIDERA Coordination and Support Action/Grant Agreement No. 101160022.

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

Code Availability The proof-of-concept implementation of our techniques, called Brush in the text, is available as an artifact [28].

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS. pp. 97–109. LNCS, Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30538-5_9
2. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Biere, A., Parker, D. (eds.) TACAS. pp. 347–367. LNCS, Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_21
3. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Finkbeiner, B., Kovács, L. (eds.) TACAS. pp. 299–329. LNCS, Springer, Cham (2024). https://doi.org/10.1007/978-3-031-57256-2_15
4. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 459–465. LNCS, Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_33
5. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. SIGPLAN Not. **44**(1), 289–300 (2009). https://doi.org/10.1145/1594834.1480917
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6) (2011). https://doi.org/10.1145/2049697.2049700
7. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS. pp. 366–378 (2007). https://doi.org/10.1109/LICS.2007.30
8. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS. pp. 287–302. LNCS, Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11691372_19
9. Drăgoi, C., Enea, C., Sighireanu, M.: Local shape analysis for overlaid data structures. In: SAS. pp. 150–171. LNCS, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_10
10. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Logozzo, F., Fähndrich, M. (eds.) SAS. pp. 215–237. LNCS, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_13
11. Giet, J., Ridoux, F., Rival, X.: A product of shape and sequence abstractions. In: Hermenegildo, M.V., Morales, J.F. (eds.) SAS. pp. 310–342. LNCS, Springer, Cham (2023). https://doi.org/10.1007/978-3-031-44245-2_15

12. Gulavani, B.S., Chakraborty, S., Ramalingam, G., Nori, A.V.: Bottom-up shape analysis. In: Palsberg, J., Su, Z. (eds.) SAS. pp. 188–204. LNCS, Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03237-0_14
13. Holík, L., Peringer, P., Rogalewicz, A., Šoková, V., Vojnar, T., Zuleger, F.: Low-Level Bi-Abduction. In: Ali, K., Vitek, J. (eds.) ECOOP. LIPIcs, vol. 222, pp. 19:1–19:30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.19
14. Illous, H., Lemerre, M., Rival, X.: A relational shape abstract domain. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NASA Formal Methods. pp. 212–229. LNCS, Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_15
15. Illous, H., Lemerre, M., Rival, X.: Interprocedural shape analysis using separation logic-based transformer summaries. In: Pichardie, D., Sighireanu, M. (eds.) SAS. pp. 248–273. LNCS, Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65474-0_12
16. Kaindlstorfer, D.: Enhancing Abstraction and Symbolic Execution for Shape Analysis of C-Programs operating on Linked Lists. Diploma thesis, TU Wien (2023). https://doi.org/10.34726/hss.2023.109623
17. Le, Q.L., Gherghina, C., Qin, S., Chin, W.N.: Shape analysis via second-order bi-abduction. In: Biere, A., Bloem, R. (eds.) CAV. pp. 52–68. LNCS, Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_4
18. Le, Q.L., Raad, A., Villard, J., Berdine, J., Dreyer, D., O'Hearn, P.W.: Finding real bugs in big programs with incorrectness logic. Proc. ACM Program. Lang. **6**(OOPSLA1) (2022). https://doi.org/10.1145/3527325
19. Magill, S., Nanevski, A., Clarke, E.M., Lee, P.: Inferring invariants in separation logic for imperative list-processing programs (2015), draft
20. Maksimović, P., Cronjäger, C., Lööw, A., Sutherland, J., Gardner, P.: Exact separation logic: Towards bridging the gap between verification and bug-finding. In: Ali, K., Salvaneschi, G. (eds.) ECOOP. LIPIcs, vol. 263, pp. 19:1–19:27. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl (2023). https://doi.org/10.4230/LIPIcs.ECOOP.2023.19
21. Polikarpova, N., Sergey, I.: Structuring the synthesis of heap-manipulating programs. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290385
22. Qin, S., He, G., Chin, W.N., Craciun, F., He, M., Ming, Z.: Automated specification inference in a combined domain via user-defined predicates. Sci. Comput. Program. **148**(C), 189–212 (2017). https://doi.org/10.1016/j.scico.2017.05.007
23. Raad, A., Berdine, J., Dang, H.H., Dreyer, D., O'Hearn, P., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: CAV. pp. 225–252. LNCS, Springer, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-53291-8_14
24. Raad, A., Berdine, J., Dreyer, D., O'Hearn, P.W.: Concurrent incorrectness separation logic. Proc. ACM Program. Lang. **6**(POPL) (2022). https://doi.org/10.1145/3498695
25. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society, USA (2002). https://doi.org/10.1109/LICS.2002.1029817
26. Rysavy, L.: Join operators for bi-abductive analysis of low-level code. Diploma thesis, TU Wien (2024). https://doi.org/10.34726/hss.2024.119373
27. Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., Garg, D.: RefinedC: automating the foundational verification of c code with refined ownership types. In: PLDI. pp. 158–174. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3453483.3454036

28. Sextl, F., Rogalewicz, A., Vojnar, T., Zuleger, F.: Artifact for "compositional shape analysis with shared abduction and biabductive loop acceleration" (2025). https://doi.org/10.5281/zenodo.14623977
29. Sextl, F., Rogalewicz, A., Vojnar, T., Zuleger, F.: Compositional shape analysis with shared abduction and biabductive loop acceleration (Extended Version) (2024), https://arxiv.org/abs/2307.06346
30. Spies, S., Gäher, L., Sammler, M., Dreyer, D.: Quiver: Guided abductive inference of separation logic specifications in coq. Proc. ACM Program. Lang. **8**(PLDI) (2024). https://doi.org/10.1145/3656413
31. Wyatt, P.: Avoiding game crashes related to linked lists (2012), http://www.codeofhonor.com/blog/avoiding-game-crashes-related-to-linked-lists
32. Zilberstein, N., Dreyer, D., Silva, A.: Outcome logic: A unifying foundation for correctness and incorrectness reasoning. Proc. ACM Program. Lang. **7**(OOPSLA1) (2023). https://doi.org/10.1145/3586045
33. Zilberstein, N., Saliling, A., Silva, A.: Outcome separation logic: Local reasoning for correctness and incorrectness with computational effects. Proc. ACM Program. Lang. **8**(OOPSLA1) (2024). https://doi.org/10.1145/3649821