Low-Level Bi-Abduction

Lukáš Holík
FIT, Brno University of Technology, Czech Republic

Petr Peringer
FIT, Brno University of Technology, Czech Republic

Adam Rogalewicz
FIT, Brno University of Technology, Czech Republic

Veronika Šoková
FIT, Brno University of Technology, Czech Republic

Tomáš Vojnar
FIT, Brno University of Technology, Czech Republic

Florian Zuleger
Faculty of Informatics, TU Wien, Austria

Abstract

The paper proposes a new static analysis designed to handle open programs, i.e., fragments of programs, with dynamic pointer-linked data structures – in particular, various kinds of lists – that employ advanced low-level pointer operations. The goal is to allow such programs be analysed without a need of writing analysis harnesses that would first initialise the structures being handled. The approach builds on a special flavour of separation logic and the approach of bi-abduction. The code of interest is analyzed along the call tree, starting from its leaves, with each function analysed just once without any call context, leading to a set of contracts summarizing the behaviour of the analysed functions. In order to handle the considered programs, methods of abduction existing in the literature are significantly modified and extended in the paper. The proposed approach has been implemented in a tool prototype and successfully evaluated on not large but complex programs.

2012 ACM Subject Classification Theory of computation → Separation logic; Theory of computation → Logic and verification; Software and its engineering → Formal software verification

Keywords and phrases programs with dynamic linked data structures, programs with pointers, low-level pointer operations, static analysis, shape analysis, separation logic, bi-abduction

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.19

Related Version *Extended Version*: `arXiv:2205.02590` [22]

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.11`

Funding The Czech authors were supported by the project 20-07487S of the Czech Science Foundation, the FIT BUT internal project FIT-S-20-6427, and L. Holík by the ERC.CZ project LL1908.

1 Introduction

Programs with complex dynamic data structures and pointer operations are notoriously difficult to write and understand. This holds twice when a need to achieve the best possible performance drives programmers, especially those working in the C language on which we concentrate, to start using advanced low-level pointer operations such as pointer arithmetic, bit-masking information on pointers, address alignment, block operations with blocks that are split to differently sized fields (of size not known in advance), which can then be merged again, and reinterpreted differently, and so on. It may then easily happen that the resulting programs contain nasty errors, such as null-pointer dereferences, out-of-bound references, double free operations, or memory leaks, which can manifest only under some rare circumstances, may escape traditional testing, and be difficult to discover once the program is in production.

To help discover such problems (or show their absence), suitable static analyses with formal roots may help. However, the problem of analysing programs with dynamic pointer-linked data structures, sometimes referred to as *shape analysis*, belongs among the most difficult analysis problems, which is related to a need of efficiently encoding and handling potentially infinite sets of graph structures of in-advance unknown shape and unbounded size, corresponding to the possible memory configurations.

Moreover, the problem becomes even harder when one needs to analyse not entire programs, equipped with some analysis harness generating instances of the data structures to be handled, but just *fragments of code*, which simply start handling some dynamic data structures through pointers without the structures being initialised first. At the same time, in practice, the possibility of analysing code fragments is highly preferred since programmers do not like writing specialised analysis harnesses for initialising data structures of the code to be analysed (not speaking about that writing such harnesses is error-prone too). Moreover, the possibility of analysing code fragments can also help scalability of the analysis since it can then be performed in a modular way.

In this paper, we propose a new analysis designed to analyse programs and even *fragments* of programs with *dynamic pointer-linked data structures* that can use advanced *low-level pointer-manipulating operations* of the form mentioned above. In particular, we concentrate on sequential C programs without recursion and without function pointers manipulating various forms of *lists* – singly-linked, doubly-linked, circular, nested, and/or intrusive, which are perhaps the most common kind of dynamic linked data structures in practice.

Our approach uses a special flavor of *separation logic* (SL) [33, 24] with *inductive list predicates* [2] to characterize sets of program configurations. To be able to handle code fragments, we adopt the principle of *bi-abductive analysis* proposed over SL for analysing programs without low-level pointer operations in [6, 7]. Our work can thus be viewed as an extension of the approach of [6, 7] to programs with truly low-level operations (i.e., pointer arithmetic, bit-masking on pointers, block operations with blocks of variable size, their splitting to fields of in-advance-not-fixed size, merging such fields back, and reinterpreting them differently, etc.). As will become clear, handling such programs requires rather non-trivial changes to the abduction procedure used in [6, 7] – intuitively, one needs new analysis rules for block splitting and merging, new support for operations such as pointer plus, pointer minus, or block operations (like `memcpy`), and also modified support for operations like memory allocation or deallocation (to avoid deallocation of parts of blocks). Moreover, to support splitting of memory blocks to parts, gradually learning their bounds and fields, and to allow for embedding data structures into other data structures not known in advance (as commonly done, e.g., in the so-called intrusive lists), we even switch from using the traditional *per-object separating conjunction* in our SL to a *per-field separating conjunction* (as used, e.g., in [14] in the context of analysing so-called overlaid data structures), requiring separation not on the level of allocated memory blocks but their fields. As an additional benefit, our usage of per-field separating conjunction then allows us to represent more compactly even some operations on traditional data structures (without low-level pointer manipulation).

As common in bi-abductive analyses, we analyse programs, or their fragments, along their *call tree*, starting from the *leaves* of the call tree (for the time being, we assume working with non-recursive programs only). Each function is analysed just once, without any knowledge about its possible call contexts. For each function, the analysis derives a set of so-called *contracts*, which can then be used when this function is called from some other function higher up in the call hierarchy. A contract for a function f is a pair (P, Q) where P is a precondition under which f can be safely executed (without a risk of running into some memory error such as a null-dereference), and Q is a postcondition that is guaranteed to be satisfied upon exit from f provided it was called under the given precondition. Both P and Q are described using our flavor of SL. In fact, as also done in [6, 7], our analysis runs in two phases: the first phase derives the preconditions, while the second phase computes the postconditions. Like in [6, 7], the computed set of contracts may *under-approximate* the set of all possible safe preconditions of f (e.g., some extreme but still safe preconditions need not be discovered). However, for each computed contract (P, Q), the post-condition Q is guaranteed to *over-approximate* all configurations that result from calling the function under the pre-condition P.

We have implemented our approach in a prototype tool called *Broom*. We have applied the tool to a selection of code fragments dealing with various kinds of lists, including very advanced implementations taken from the Linux kernel as well as the intrusive list library (for a reference, see our experimental section). Although the code is not large in the number of lines of code, it contains very advanced pointer operations, and, to the best of our knowledge, Broom is currently the only analyser that is capable of analysing many of the involved functions.

Related work

In the past (at least) 25 years there have appeared numerous approaches to automated *shape analysis* or, more generally, analysis of programs with unbounded dynamically-linked data structures. These approaches differ in the formalisms used for encoding sets of configurations of programs with such data structures, in their level of automation, classes of supported data structures, and/or properties of programs that are targeted by the analysis: see, e.g., [25, 34, 2, 37, 9, 39, 38, 20, 10, 3, 16, 21, 31].

Not many of the existing approaches offer a reasonably general support of *low-level pointer operations* (such as pointer arithmetic, address alignment, masking information on pointers, block operations, etc.). Some support of low-level pointer operations appears in multiple of these approaches, but it is often not much documented. In fact, such a support often appears in some *ad hoc* extension of the tool implementing the given approach only, without any description whatsoever. According to the best of our knowledge, the approach of [16], based on so-called *symbolic memory graphs (SMGs)*, currently provides probably the most systematic and generic solution for the case of programs with low-level pointer operations and various kinds of linked lists (including advanced list implementations such as those used in the Linux kernel). Specialised approaches to certain classes of low-level programs, namely, *memory allocators*, then appear, e.g., in [5, 19].

In this work, we get inspired by some of the analysis capabilities of [16], but we aim at removing one of its main limitations – namely, the fact that it cannot be applied to a *fragment of code*. Indeed, [16] expects the analysed program to be *closed*, i.e., the analysed functions must be complemented by a *harness* that initializes all the involved data structures, which severely limits applicability of the approach in practice (since programmers are often reluctant to write specialised analysis harnesses).

Approaches allowing one to analyse *open code*, i.e., *code fragments*, with dynamic linked data structures are not frequent in the literature. Perhaps the best known of these works is the approach of *bi-abduction* based on *separation logic* with (possibly nested) list predicates proposed in [6, 7] and currently available in the Infer analyser [4].[1] This approach is another of the approaches that inspired our work, and we will be referring to various technical details of that paper later on. However, despite Infer contains some support of pointer arithmetic, it is not very complete (as our experiments will show), and the approach presented in [6, 7] does not at all study low-level pointer operations of the form that we aim at in this paper. Moreover, it turns out that adding a support of such operations (e.g., dealing with blocks of memory of possibly variable size, splitting them to fields of variable size, merging such fields back and reinterpreting their contents differently, having pointers with variable offsets, supporting rich pointer arithmetic, etc.) requires rather non-trivial changes and extensions to the bi-abduction mechanisms used in [6, 7].

An approach of *second-order bi-abduction* based also on separation logic was proposed in [28] and several follow-up papers such as [11]. The authors consider recursive programs with pointers and propose a calculus for automatic derivation of sets of equations describing the behaviour of particular functions. A solution of such a set of equations leads to a set of contracts for the considered functions. The technique is in some sense quite general – unlike [6, 7] and unlike our approach, it can even automatically learn *recursive predicates* describing the involved data structures, including trees, skip lists, etc. Moreover, the derivation of the equations is a cheap procedure, and no widening is needed, again unlike in [6, 7] and unlike in our approach. On the other hand, finding a solution of the generated equations is a hard problem, and the authors provide a simple heuristic designed for a specific shape of the equations only, which fails in various other cases.

Finally, we mention the Gillian project, a *language-independent framework* based on separation logic for the development of *compositional symbolic analysis* tools, including tools for whole-program symbolic execution, verification of annotated code, as well as bi-abduction [36, 35, 30, 29]. The works on Gillian concentrate on the generic framework it develops, and the published description of the supported bi-abductive analysis, perhaps most discussed in [35], is unfortunately not very detailed. In particular, it is not clear whether and how much the approach supports the low-level features of pointer manipulation that we are aiming at here (e.g., pointer arithmetic, bit-masking on addresses, etc.). According to the source code that we were able to find in the Gillian repository, the examples mentioned in the part of [35] devoted to bi-abduction do not use low-level pointer manipulation features such as pointer arithmetic. It is also mentioned in [35] that Gillian supports bi-abduction up to a predefined bound only, whereas we do not require such a bound. Further, in contrast to the present work, [35] assumes that the size of memory chunks being dynamically allocated is known, and the complex reasoning needed to resolve this issue is left for the future.

We also note that there is a vast body of work on *automated decision procedures* for various fragments of separation logic and problems such as satisfiability and entailment – see, e.g., [18, 23, 26, 27, 17]. However, it is not immediate how to apply these logics inside a program analysis tool. This is because the best (i.e., logically weakest) solution to the abduction problem $\varphi * [?] \models \psi$, which is a central problem for compositional program analyses, with $*$ being the separating conjunction, is given by the formula $\varphi \mathrel{-\!\!*} \psi$, which makes use of the magic wand operator $\mathrel{-\!\!*}$, and the cited logics do not provide support for the magic wand. This is for principle reasons: it has been observed in the literature that magic wand operators are "difficult to eliminate" [1]; further, it has been shown that adding only the

[1] The approach [6, 7] mentions a generalisation to other classes of data structures, but – to the best of our knowledge – this extension has not been implemented and evaluated, and so it is not clear how well it would work in practice.

singly-linked list-segment predicate to a propositional separation logic that includes the magic wand already leads to undecidability of the satisfiability problem [13]. A notable exception is the recent work [32] on a new semantics for separation logic, which enables decidability of a propositional separation logic that includes the magic wand and the singly-linked list-segment predicate (and also discusses applications to the abduction problem); however, the fragment considered in [32] is not expressive enough to cover the low-level features considered in this work such as, pointer arithmetic, memory blocks, etc., and, at present, it is unclear whether the decidability result can be extended to a richer logic. For the above reasons, we will in this paper not target a complete procedure for the (bi-)abduction problem, but rather, following [6, 7], develop approximate procedures and evaluate their usefulness in our case studies.

Main contributions of the paper

The paper proposes a new approach for automated bi-abductive analysis of programs and fragments of programs with pointers, different kinds of linked lists, and low-level memory operations. The approach is formalised, implemented in a prototype tool, and experimentally evaluated. In summary, we make the following contributions:

- A specialised dialect of separation logic suitable for automated abductive analysis of programs with lists and low-level memory operations (we use a separating conjunction between single fields and not whole memory blocks as in related approaches, and support fields of unknown and even variable size as well as unknown block boundaries).
- Contracts for basic programming statements that reflect our low-level memory model (see, e.g., the contracts of the `malloc` and `free` statements), and support for specific statements that permit low-level pointer manipulation (e.g., pointer addition).
- A set of rules for automated abductive analysis, which not only includes variants of rules from related approaches, but also new kinds of rules required for handling low-level memory operations (e.g., block splitting).
- A prototype implementation that supports bit-precise reasoning based on a reduction of (un-)satisfiability of separation logic to (un-)satisfiability of SMT over the bit-vectors.
- An experimental evaluation of the approach on a number of challenging programs.

2 An Illustration of the Approach on an Example

Before we start with a systematic description of our approach, we present its core ideas on an example. We attempt to informally explain the involved notions, yet, due to the complexity of the issues, some prior knowledge of separation logic with *inductive list predicates*, e.g., [2], and ideally also bi-abduction analysis [6, 7] is helpful.

As our illustrative example, we consider the code manipulating cyclic doubly-linked lists shown in Fig. 1.[2] The example is inspired by the principle of *intrusive lists* (as used, e.g., in Linux kernel lists) where all list operations are defined on some simple list-linking structure that is then nested into user-defined structures. It is these user-defined structures that carry the data actually stored in the lists. The list manipulating functions, however, know nothing about these larger structures. However, the fact that contracts (summaries) derived for

[2] The code is written in C. Our later presented low-level programming language for which we will formalise our approach is not C but rather close to some of the intermediate languages used when compiling C. We, however, feel that describing the example in such a language would not be very understandable. Moreover, all constructions used in our example can be translated to the later considered language.

functions dealing with the small linking structures are later to be applied on the larger, user-defined structures is already problematic for some existing analyses.

In the code of our illustrative example, the function `init_dll` creates an initial cyclic doubly-linked list consisting of a single node. The function `insert_after` can then insert a new element into the list after its item pointed by l.

Let us note that while the code of the example in Fig. 1 may seem to not use pointer arithmetic, the code in fact uses pointer arithmetic on the level of the intermediate code we analyse. Indeed, each expression `x->field` is translated to `*(x+offsetof(field))`. It is of course true that once all the types and fields are known and fixed, one can avoid dealing with pointer arithmetic in this case. On the other hand, the fact that we systematically handle it through pointer arithmetic allows us to smoothly handle even the cases when the types and offsets stop being known and/or constant (upon which approaches based on dealing with field names fail).

As indicated already in the introduction, we analyse the given code fragment according to its *call tree*, starting from the leaves (assuming there is no recursion). Each function is analysed just once, without any call context. If successful, the analysis derives a set of contracts for the given function where each contract is a pair (P, Q) consisting of a (conjunctive) pre-condition and (a possibly disjunctive) post-condition. In our introductory example, we will restrict ourselves to the simplest case, namely, having a single, purely conjunctive contract. In the contracts, both the pre- and post-condition are expressed as SL formulae. The analysis is *compositional* in that contracts derived for some functions are then used when analysing functions higher up in the call hierarchy (moreover, we will view even particular pointer manipulating statements as special atomic functions and describe them by pre-defined contracts).

We begin the illustration of our analysis by analysing the `init_dll` function. We start the analysis by annotating the first line by the pair $(x = X, x = X)$. In this pair, the first component is the so-far derived pre-condition of the function, and the second component is the current symbolic state of the function under analysis. Here, the variable X records the value of the program variable x at the beginning of the function. While x will be changing in the function, X will never change, and we will be able to gradually generate constraints on its value to express what must hold for x at the entry of the function.

After symbolically executing the statement `x->next = x`, we derive that the address X must correspond to some allocated memory, containing some unknown value L_1. This gives us the pre-condition $X \mapsto L_1$ that is an SL formula stating exactly the fact that X is allocated and stores the value L_1. The symbolic state is then advanced to say that X is allocated and stores the value X, i.e., it points to itself, which is encoded as $X \mapsto X$ in SL.

After the subsequent statement `x->prev = x`, assuming that we work with 64 bit (i.e., 8 bytes) wide addresses, we add to the precondition the fact that the memory address $X + 8$ is allocated as well. Moreover, the formula $\mathfrak{b}(X) = \mathfrak{b}(X + 8)$ says that X and $X + 8$ belong to the same memory block, i.e., they were, e.g., allocated using one `malloc` statement (in fact, we use $\mathfrak{b}(X)$ to denote the – so-far unknown – base address of the block). The symbolic state is updated by the fact that the value at the address $X + 8$ is also equal to X, i.e., $X + 8 \mapsto X$.

Since there are no further statements in the function, there is no branching, no loops, and all the statements are deterministic, the final *contract* for the function is unique and consists of the final pre-condition $P \equiv X \mapsto L_1 * X + 8 \mapsto L_2 * \mathfrak{b}(X) = \mathfrak{b}(X + 8) * x = X$ and the post-condition $Q \equiv X \mapsto X * X + 8 \mapsto X * \mathfrak{b}(X) = \mathfrak{b}(X + 8) * x = X$ obtained from the final symbolic state. Here, we use "$*$" to denote a *per-field separating conjunction*, which,

```
struct dll { struct dll *next, *prev; };
struct emb_dll {int value; struct dll link; };

void init_dll(struct dll *x) {
            $P \equiv x = X, \quad Q \equiv x = X$
      x–¿next = x;
            $P \equiv X \mapsto L_1 * x = X, \quad Q \equiv X \mapsto X * x = X$
      x–¿prev = x;
            $P \equiv X \mapsto L_1 * X + 8 \mapsto L_2 * \mathfrak{b}(X) = \mathfrak{b}(X+8) * x = X,$
            $Q \equiv X \mapsto X * X + 8 \mapsto X * \mathfrak{b}(X) = \mathfrak{b}(X+8) * x = X$
} summary:
      $P \equiv X \mapsto L_1 * X + 8 \mapsto L_2 * \mathfrak{b}(X) = \mathfrak{b}(X+8) * x = X,$
      $Q \equiv X \mapsto X * X + 8 \mapsto X * \mathfrak{b}(X) = \mathfrak{b}(X+8) * x = X$

void insert_after(struct dll *l, *j) {
            $P \equiv l = L * j = J, \quad Q \equiv l = L * j = J$
      struct dll *n = l–¿next;
            $P \equiv L \mapsto N * l = L * j = J, \quad Q \equiv L \mapsto N * l = L * j = J * n = N$
      j–¿next = n;
            $P \equiv L \mapsto N * J \mapsto B_1 * l = L * j = J, \quad Q \equiv L \mapsto N * J \mapsto N * l = L * j = J * n = N$
      j–¿prev = l;
            $P \equiv L \mapsto N * J \mapsto B_1 * J + 8 \mapsto B_2 * \mathfrak{b}(J) = \mathfrak{b}(J+8) * l = L * j = J,$
            $Q \equiv L \mapsto N * J \mapsto N * J + 8 \mapsto L * \mathfrak{b}(J) = \mathfrak{b}(J+8) * l = L * j = J * n = N$
      l–¿next = j;
            $P \equiv L \mapsto N * J \mapsto B_1 * J + 8 \mapsto B_2 * \mathfrak{b}(J) = \mathfrak{b}(J+8) * l = L * j = J,$
            $Q \equiv L \mapsto J * J \mapsto N * J + 8 \mapsto L * \mathfrak{b}(J) = \mathfrak{b}(J+8) * l = L * j = J * n = N$
      n–¿prev = j;
            $P \equiv L \mapsto N*J \mapsto B_1*J+8 \mapsto B_2*N+8 \mapsto B_3*\mathfrak{b}(J) = \mathfrak{b}(J+8)*\mathfrak{b}(N) = \mathfrak{b}(N+8)*l = L*j = J,$
            $Q \equiv L \mapsto J * J \mapsto N * J + 8 \mapsto L * N + 8 \mapsto J * \mathfrak{b}(J) = \mathfrak{b}(J+8) * \mathfrak{b}(N) = \mathfrak{b}(N+8) * l = L *$
            $j = J * n = N$
} summary:
      $P \equiv L \mapsto N * J \mapsto B_1 * J + 8 \mapsto B_2 * N + 8 \mapsto B_3 * \mathfrak{b}(J) = \mathfrak{b}(J+8) * \mathfrak{b}(N) = \mathfrak{b}(N+8) * l = L * j = J,$
      $Q \equiv L \mapsto J * J \mapsto N * J + 8 \mapsto L * N + 8 \mapsto J * \mathfrak{b}(J) = \mathfrak{b}(J+8) * \mathfrak{b}(N) = \mathfrak{b}(N+8) * l = L * j = J$

int main() {
            $P \equiv \mathsf{emp}, \quad Q \equiv \mathsf{emp}$
      struct emb_dll *x = malloc(sizeof(struct emb_dll));
            $P \equiv \mathsf{emp}, \quad Q \equiv \exists X.\ X \mapsto \top[24] * X = \mathfrak{b}(X) * x = X$
      init_dll(&(x–¿link));
            $P \equiv \mathsf{emp}, \quad Q \equiv \exists X, L_1.\ X \mapsto \top[8] * L_1 \mapsto L_1 * L_1 + 8 \mapsto L_1 * X = \mathfrak{b}(X) = \mathfrak{b}(L_1) = \mathfrak{b}(L_1+8) *$
            $L_1 = X + 8 * x = X$
      struct emb_dll *i = malloc(sizeof(struct emb_dll));
            $P \equiv \mathsf{emp}, \quad Q \equiv \exists I, X, L_1.\ I \mapsto \top[24] * X \mapsto \top[8] * L_1 \mapsto L_1 * L_1 + 8 \mapsto L_1 * L_1 = X + 8 * X =$
            $\mathfrak{b}(X) = \mathfrak{b}(L_1) = \mathfrak{b}(L_1+8) * I = \mathfrak{b}(I) * x = X * i = I$
      init_dll(&(i–¿link));
            $P \equiv \mathsf{emp}, \quad Q \equiv \exists I, X, L_1, L_2.\ i \mapsto \top[8] * L_2 \mapsto L_2 * L_2 + 8 \mapsto L_2 * X \rightarrow \top[8] * L_1 \mapsto$
            $L_1 * L_1 + 8 \mapsto L_1 * L_2 = I + 8 * L_1 = X + 8 * X = \mathfrak{b}(X) = \mathfrak{b}(L_1) = \mathfrak{b}(L_1+8) * I = b(I) =$
            $b(L_2) = b(L_2+8) * x = X * i = I$
      insert_after(&(x–¿link), &(i–¿link));
            $P \equiv \mathsf{emp}, \quad Q \equiv \exists I, X, L_1, L_2.\ I \mapsto \top[8] * L_2 \mapsto L_1 * L_2 + 8 \mapsto L_1 * X \mapsto \top[8] * L_1 \mapsto L_2 * L_1 +$
            $8 \mapsto L_2 * L_2 = I+8*L_1 = X+8*X = \mathfrak{b}(X) = b(L_1) = \mathfrak{b}(L_1+8)*I = \mathfrak{b}(I) = \mathfrak{b}(L_2) = \mathfrak{b}(L_2+8)*$
            $x = X * i = I$

      . . .
}
```

Figure 1 An illustrative example of a code working with cyclic doubly-linked lists and its analysis. The C expressions like `ptr->field` can be seen as syntactic sugar for expressions using pointer arithmetic of the form `*(ptr + offsetof(field))`. The $\mathfrak{e}(X)$ predicates representing *the end of the block pointed by* X are dropped from the (P, Q) pairs for simplicity.

intuitively, means that while the addresses X and $X + 8$, which are allocated by the formulae $X \mapsto L_1$ and $X + 8 \mapsto L_2$, may – though need not – belong to a single memory block, the values stored at these addresses within the block do not overlap.[3]

The same principles are then used for the computation of the contracts for the `insert_after` and `main` functions. Here, let us just highlight a situation that happens, e.g., upon the `j->next = n` statement of `insert_after`. Notice that, in its case, the so-far computed precondition P must be extended by the new requirement $J \mapsto B_1$, stating that J must be allocated, and Q is then extended by the fact $J \mapsto N$, which is the effect of executing the given statement. At the same time, however, the rest of the previously computed symbolic state of the program Q stays untouched (in general, only some part may be preserved). Given the current symbolic state Q and a statement, the problem of deriving which precondition is missing and which part of the state will remain untouched is denoted as the *bi-abduction problem*, and a procedure looking for its solution is a *bi-abduction procedure*. The computed missing part of the pre-condition is called the *anti-frame*, and the computed part of the current symbolic state not modified by the statement being executed is called the *frame*.

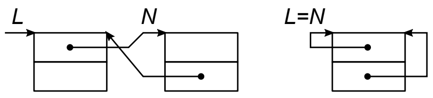
When analysing the `main` function, one does already need not re-analyse the `init_dll` and `insert_after` functions – instead, one simply uses their contracts. For simplicity, we assume here that `malloc` always succeeds, and hence even `main` is deterministic. After the execution of `malloc`, we use the special predicate $x \mapsto \top[24]$ to express that a sequence of 24 bytes of undefined contents was allocated. We allow such blocks (as well as all other kinds of blocks that arise during the analysis) be *split* to smaller parts whenever this is needed for applying a contract of some function (or statement). That happens, e.g., on lines b and d of the `main` function where the block $X \mapsto \top[24]$ created by `malloc` is split to 3 fields as described by $X \mapsto \top[8] * X + 8 \mapsto \top[8] * X + 16 \mapsto \top[8]$. The last two of the fields then match the precondition of `init_dll`, and the first one becomes a frame (untouched by the function).

Without now going into further details, we note that analysing more complex functions requires one to solve multiple more problems. For example, if there appears some non-determinism, one needs to start working with contracts with disjunctive post-conditions and even with sets of such contracts. If the code contains loops, one needs to prevent the analysis from diverging while generating more and more points-to predicates. For that, one can use widening in the form of a list abstraction. The resulting over-approximation may then, however, render some generated pre-/post-condition pairs unsound, leading to a need to run another phase of the analysis that will start from the computed pre-conditions and check, without using abduction any more, what post-condition the code can really guarantee. We discuss all these issues in the extended version of this paper [22].

However, before proceeding, let us stress how significantly the above-mentioned use of the per-field separation distinguishes our approach from its predecessor bi-abduction analysis [6, 7]. That analysis would use *whole-block* predicates of the form $X \mapsto dll(next : A, prev : B)$ to describe instances of `struct dll`, while we use the formula $X \mapsto A * X + 8 \mapsto B * \mathfrak{b}(X) = \mathfrak{b}(X + 8)$. The per-field separating conjunction allows us to (1) express partial information about a block and (2) infer a precondition where two (or more) fields can be in the same block as well as in different blocks. Point 1 helps us to generate contracts of functions where we do not know the exact sizes of the allocated block – e.g., `init_dll` does not require the

[3] In a formula $a \mapsto b * c \mapsto d$ with a per-object separating conjunction, a and c are two distinct objects allocated in memory (while b and d need not be allocated and may coincide). With a per-field separating conjunction, a and c are allowed to be non-overlapping *fields* of the same allocated object.

pointer x to point to an instance of `struct dll`, it can be, e.g., used on larger structures, such as, e.g., `struct emb_dll`, that *embed* the original structure. Point 2 is used in the contract of `insert_after` where the formula $L \mapsto N * N + 8 \mapsto B_3$ describes a memory where it may be that $L = N$ as well as $L \neq N$. The contract for `insert_after` can then be applied on a circular doubly-linked list consisting of a single item ($L = N$) as well on lists consisting of more items ($L \neq N$) – see the figure below for an illustration.



Note that when one uses the whole-block predicate, the precondition of `insert_after` in the form $L \mapsto \mathit{dll}(\mathit{next} : N, \mathit{prev} : \_) * N \mapsto \mathit{dll}(\mathit{next} : \_, \mathit{prev} : B_3) * J \mapsto \mathit{dll}(\mathit{next} : B_1, \mathit{prev} : B_2)$ requires $L \neq N$, and hence it is not covering the two above mentioned cases. One can of course sacrifice performance of the analysis and generate multiple contracts by modifying the abduction rules – e.g., one can non-deterministically introduce an alias $L = N$ before inferring the anti-frame on line v of `main` to get the pre-condition $L \mapsto \mathit{dll}(\mathit{next} : L, \mathit{prev} : \_) * L = N$. Introducing such non-determinism is, however, costly. That is why, as we will see in our experiments, it is not done in tools such as Infer, which can then cause that such tools will miss some function contracts (or generate incomplete contracts that will not be applicable in some common cases: such as insertion into a list of length 1).

An additional example is provided in the technical report [22], where pointer arithmetic and bit-masking are directly visible in the C-code.

3 Memory Model

In the following, we introduce the memory model that we use in this paper. *Values* are sequences of bytes, i.e., $\mathit{Val} = \mathit{Byte}^+$, where bytes are 8-bit words. Sequences of bytes can be interpreted as numbers – either signed or unsigned, which we leave as a part of the operations to be applied on the sequences (including conversion operations). We designate a subset of the values $\mathit{Loc} = \mathit{Byte}^N \subseteq \mathit{Val}$ as *locations* where $N \geqslant 1$ is the byte-width of words of a given architecture and where byte sequences to be interpreted as locations are always understood as unsigned. The null pointer is represented by $0 \in \mathit{Loc}$ in our memory model.

We will use so-called *stack-block-memory triplets* (SBM triplets for short) as *configurations* of our memory model in order to define the operational semantics of programs (and also to define the semantics of our separation logic later on):

Stack. We assume some set of *variables* Var where each variable $x \in \mathit{Var}$ has some fixed positive size, denoted as $\mathsf{size}(x)$. Then, *Stack* is the set of total functions $\mathit{Var} \to \mathit{Val}$ such that each variable is mapped to a byte sequence whose length is according to the size of the variable, i.e., for each stack $S \in \mathit{Stack}$ and variable $x \in \mathit{Var}$, we have $S(x) \in \mathit{Byte}^{\mathsf{size}(x)}$.

Memory. *Mem* is the set of partial functions $\mathit{Loc} \rightharpoonup \mathit{Byte}$ that define the contents of allocated memory locations.

Blocks. We use $\mathit{Interval} = \{ [l, u) \mid l < u \text{ where } l, u \in \mathit{Loc}\}$ to denote intervals of subsequent memory locations where we include the lower bound and exclude the upper bound. Intuitively, an interval $[l, u) \in \mathit{Interval}$ will denote which locations were allocated at the same time (and must thus also be deallocated together, can be subtracted using pointer subtraction, etc.). $\mathit{Block} = \{\, [l,u) \in \mathit{Interval} \mid l \neq 0\}$ are intervals whose lower bound is not 0 (recall that null is represented by $0 \in \mathit{Loc}$ in our memory model). $\mathit{Blocks} \subseteq (\mathbf{2}_{\mathit{fin}})^{\mathit{Block}}$ is the set of all finite sets of non-overlapping blocks, i.e., for all $B \in \mathit{Blocks}$ and for all $[l_1,u_1),[l_2,u_2) \in B$ such that either $l_1 \neq l_2$ or $u_1 \neq u_2$, we have that either $u_1 \leqslant l_2$ or $u_2 \leqslant l_1$.

Configurations. *Config* consists of all triplets $(S,B,M) \in \mathit{Stack} \times \mathit{Blocks} \times \mathit{Mem}$ such that the set of allocated blocks and the locations whose contents is defined are linked as follows:

- For every $\ell \in \mathit{Loc}$ s.t. $M(\ell)$ is defined, there is a block $[l,u) \in B$ s.t. $\ell \in [l,u)$.[4]

We introduce functions $\mathfrak{b}_B, \mathfrak{e}_B : \mathit{Loc} \to \mathit{Loc}$, parameterized by some set of blocks $B \in \mathit{Blocks}$, which return the base or end address, respectively, of the block to which a given location belongs, i.e., given some $\ell \in \mathit{Loc}$, we set $\mathfrak{b}_B(\ell) = l$ in case there is some $[l,u) \in B$ with $\ell \in [l,u)$, and $\mathfrak{b}_B(\ell) = 0$, otherwise. Likewise for $\mathfrak{e}_B(\ell)$.

Axioms. For later use, we note that, building on the above notation, we can express the requirements for locations to be within their associated block and for blocks to be non-overlapping in the form of the following two axioms:

$$\forall \ell.\ \mathfrak{b}_B(\ell) = 0 \vee \mathfrak{b}_B(\ell) \leqslant \ell < \mathfrak{e}_B(\ell)$$
$$\forall \ell,\ell'.\ (0 < \mathfrak{b}_B(\ell) < \mathfrak{e}_B(\ell') \leqslant \mathfrak{e}_B(\ell) \vee 0 < \mathfrak{b}_B(\ell') < \mathfrak{e}_B(\ell) \leqslant \mathfrak{e}_B(\ell')) \rightarrow \\ \mathfrak{b}_B(\ell) = \mathfrak{b}_B(\ell') \wedge \mathfrak{e}_B(\ell) = \mathfrak{e}_B(\ell')$$

Notation. Given a (partial) function f, $f[a \hookrightarrow b]$ denotes the (partial) function identical to f up to $f[a \hookrightarrow b](a) = b$. Moreover, $f[a \hookrightarrow \bot]$ denotes the (partial) function identical to f up to being undefined for a.

4 A Low-level Language and Its Operational Semantics

We now state a simple low-level language together with its operational semantics. The language is close to common intermediate languages into which programs in C are compiled by compilers such as `gcc` or `clang`. We assume that a type checker ensures that variables of the right sizes are used, guaranteeing, in particular, that the left-hand side (LHS) and right-hand side (RHS) of an assignment are of the same size or that the dereference operator is only applied to locations. We do not include the operators of *item access* (`.` and `->`) nor *indexing* (`[]`) into our language as their usage can be compiled to using *pointers*, *pointer arithmetic*, and the *dereference operator* (`*`) as indeed commonly done by compilers. Likewise, we do not include the *address-of operator* (`&`) whose usage can be replaced by storing all objects whose address should be derived via `&` into dynamically allocated memory, followed by using pointers to such memory, as also done automatically by some compilers. Further, we assume the `sizeof` and `offsetof` operators be resolved and transformed to constants.

We now present the statements of our low-level language together with their operational semantics. The semantics is defined over configurations, which we introduced in the previous section. The semantics maintains the following invariant:

[4] Note that we do not require the reverse, i.e., that all locations of a block are allocated. This is because our separation logic is set up to work with partially allocated blocks. In particular, the separating conjunction needs to break up blocks into partial blocks. We note, however, that the semantics of our programming language maintains the invariant that each block is always fully allocated.

- For every $[l, u) \in B$ and every $\ell \in [l, u)$, $M(\ell)$ is defined.

We start with rules describing various *assignment statements* possibly combined with pointer dereferences either on the LHS or RHS. In the rules (and further on), we use $M[\ell, \ell')$ to denote the byte sequence $M(\ell)M(\ell+1)\cdots M(\ell'-1)$:

$$(S, B, M) \xrightarrow{x:=k} (S[x \hookrightarrow k], B, M) \text{ for some value } k \in \mathit{Val}$$

$$(S, B, M) \xrightarrow{x:=y} (S[x \hookrightarrow S(y)], B, M)$$

$$(S, B, M) \xrightarrow{x:=*y} \text{ if } \mathfrak{b}_B(S(y)) = 0 \text{ or } S(y) + \mathsf{size}(x) > \mathfrak{e}_B(S(y)), \\ \text{then } \mathit{error} \text{ else } (S[x \hookrightarrow M[S(y), S(y) + \mathsf{size}(x))], B, M)$$

Note that, in the case of $x := *y$, one needs to read $\mathsf{size}(x)$ bytes from the adress $S(y)$. This is impossible if the condition $S(y) + \mathsf{size}(x) > \mathfrak{e}_B(S(y))$ holds.

$$(S, B, M) \xrightarrow{*x:=y} \text{ if } \mathfrak{b}_B(S(x)) = 0 \text{ or } S(x) + \mathsf{size}(y) > \mathfrak{e}_B(S(x)), \\ \text{then } \mathit{error} \text{ else } (S, B, M[[S(x), S(x) + \mathsf{size}(y)) \hookrightarrow S(y)])$$

We continue by *memory allocation*. We treat 0-sized allocations as an error.[5] For non-zero-sized allocations, the allocation can always fail and return null, otherwise the successfully allocated memory block is initialized with some arbitrary value[6]:

$$(S, B, M) \xrightarrow{x=malloc(z)} \text{ if } S(z) = 0 \text{ then } \mathit{error} \text{ else either } (S[x \hookrightarrow \mathsf{null}], B, M) \text{ or} \\ (S[x \hookrightarrow \ell], B \cup \{[\ell, \ell + S(z))\}, M[[\ell, \ell + S(z)) \hookrightarrow k]) \text{ for some } k \in \mathit{Byte}^{S(z)} \text{ and } \ell > 0 \\ \text{such that } \ell + S(z) \leqslant 2^{8N} \text{ and } [\ell, \ell + S(z)) \text{ does not overlap with any } [l, u) \in B$$

The calloc function, which nullifies the allocated block, can be defined analogically to `malloc`, by just changing $M[[\ell, \ell + S(z)) \hookrightarrow k]$ to $M[[\ell, \ell + S(z)) \hookrightarrow 0^{S(z)}]$. The `realloc` function, which shrinks or enlarges a block, possibly moving it to a different memory location, can be reduced to a sequence of other statements, and so we do not introduce it explicitly for brevity.

The *deallocation* of memory is modelled by the following rule:[7]

$$(S, B, M) \xrightarrow{free(x)} \text{ if } S(x) \neq \mathfrak{b}_B(S(x)) \text{ then } \mathit{error} \\ \text{else } (S, B \backslash \{[S(x), \mathfrak{e}_B(S(x)))\}, M[[S(x), \mathfrak{e}_B(S(x))) \hookrightarrow \bot])$$

The low-level language further contains a collection of binary and unary operations denoted as bop and uop, respectively. The operations of adding an offset to a pointer (ptrplus) and pointer subtraction (ptrsub) are special and handled separately. The operation ptrplus for *adding a (possibly negative) offset to a pointer* requires its pointer argument to be defined,

[5] The C standard says that the behaviour in this case is user-defined, the allocation can return null or a non-null value, which, however, cannot be dereferenced. However, since such an allocation is usually suspicious, many analysers flag it as an error/warning. We adopt the same approach, but if need be, the rules could be changed to handle such allocations according to the standard.

[6] Notice that 2^{8N} gives the largest address that can be expressed using words with the byte-width N.

[7] Notice that we do not need a rule for deallocating zero-sized blocks since we do not allow such blocks to be created.

and, in accordance with the C standard, the result must be within the appropriate memory block plus one byte (i.e., it may point just behind the end of the block).[8] The operation ptrsub for *pointer subtraction* is special in that it requires its pointer operands to be defined, to have the same base, and to point inside an allocated block or just behind its end. We also support the `memcpy` statement (and can simulate the `memmove` statement). To encode *conditional branching* arising from conditional statements or loops, we introduce the `assume` statement that models conditions $x \bowtie y$ for $\bowtie \in \{=, \neq, \leqslant, <, \geqslant, >\}$. We allow *functions* without a return value, not referring to global variables, having parameters passed by reference only, with the names of the parameters unique to each function, and not having local variables. We also introduce the `assert` statement that is similar to the `assume` statement, but it checks at runtime whether the specified condition holds, and it fails if this is not the case. The operational semantics of all these statements can be found in [22].

5 Separation Logic

We now introduce a separation logic that supports reasoning about low-level memory models as introduced earlier. Our separation logic (SL) has the following syntax:

$$\begin{aligned}\varphi \ ::= \ & \varepsilon_1 \mapsto \varepsilon_2 \mid \varepsilon_1 \mapsto k[\varepsilon_2] \mid \varepsilon_1 \mapsto \top[\varepsilon_2] \mid \varphi_1 * \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathsf{ls}_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2) \mid \\ & \mathsf{dls}_{\Lambda(x,y,z)}(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2') \mid \mathsf{emp} \mid \mathit{true} \mid \varepsilon_1 \bowtie \varepsilon_2 \mid \exists x.\varphi \\ \bowtie \ ::= \ & = \mid \neq \mid \leqslant \mid < \mid \geqslant \mid > \qquad \varepsilon \ ::= \ k \mid x \mid \mathfrak{b}(\varepsilon) \mid \mathfrak{e}(\varepsilon) \mid \mathsf{uop}\,\varepsilon \mid \varepsilon_1 \,\mathsf{bop}\, \varepsilon_2\end{aligned}$$

Variables and Values. Our SL formulae are stated over the same set of variables *Var* and values *Val* that we introduced in the definition of our memory model. In particular, the variables x, y, z and the values k of our SL formulae are drawn from *Var* and *Val*, respectively.

Size. Variables, values, operators, and expressions in our logic are typed by their *size*. We will only work with formulae where the variables and values respect the sizes expected by the involved operations and predicates. For every expression ε, we denote by $\mathsf{size}(\varepsilon)$ the size of the value to which this expression may evaluate. We remark on the choice of working with fixed sizes: We intentionally do not permit variables of variable size because (1) such variables are typically not supported by low-level languages and (2) variables of variable size allow one to model strings, which would make our language vastly more powerful (allowing one to model all kinds of string operations)[9].

Points-To Predicates. The points-to predicate $\varepsilon_1 \mapsto \varepsilon_2$ denotes that the byte sequence ε_2 is stored at the memory location ε_1. Due to we are working with expressions of fixed size, every model of $\varepsilon_1 \mapsto \varepsilon_2$ must allocate exactly $\mathsf{size}(\varepsilon_2)$ bytes. In addition, we introduce two restricted cases of points-to predicates where the RHS is of parametric size: namely, $\varepsilon_1 \mapsto k[\varepsilon_2]$ and $\varepsilon_1 \mapsto \top[\varepsilon_2]$ that allow us to say that ε_1 points to an array of ε_2 bytes that either all have the same constant value k or have any value, respectively. These predicates allow us to, e.g., express that some block of memory is nullified, which is often crucial to

[8] We are aware that this requirement is not respected in some real-life pograms, such as, e.g., the implementation of lists in Linux. We will later mention that our approach can be relaxed to handle such cases too.

[9] We believe that extending our later presented analysis to such variables is possible (by recording the length of the target object as another parameter of the points-to predicate), but we leave it for future work in order not to complicate the basic approach we propose.

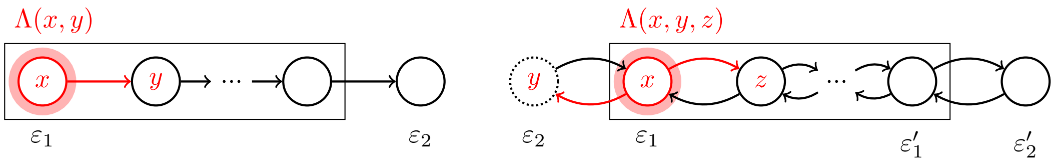


Figure 2 An illustration of the meaning of the $\mathsf{ls}_{\Lambda(x,y)}(\varepsilon_1,\varepsilon_2)$ and $\mathsf{dls}_{\Lambda(x,y,z)}(\varepsilon_1,\varepsilon_2,\varepsilon'_1,\varepsilon'_2)$ formulae.

know when analysing advanced implementations of dynamic data structures [16]. We lift the notion of size to the RHS of these points-to predicates as follows: $\mathsf{size}(k[y]) = \mathsf{size}(\top[y]) = y$. In $\varepsilon_1 \mapsto k[\varepsilon_2]$, we require k to be a single byte, i.e., $\mathsf{size}(k) = 1$.

Notation. Given a formula φ, we write $var(\varphi)$ to denote the *free* variables of φ (as usual a variable is *free* if it does appear within an existential quantification). Further, given an expression ε, we write $var(\varepsilon)$ for all variables appearing in ε.

Terminology. We call formulae that do not contain the disjunction operator ($\vee$) *symbolic heaps*. We will mostly work with symbolic heaps in this paper. Disjunctions of symbolic heaps will be only used on the RHS of (some) contracts. We call formulae that do not contain existential quantification ($\exists$) *quantifier-free*. Our SL contains the relational predicates $\varepsilon_1 \bowtie \varepsilon_2$, which include equality and disequality; these predicates are traditionally called *pure* in the separation logic literature. We follow this terminology and call any separating conjunction of such predicates a *pure formula*.

List-Segment Predicates. List segments in our SL are parameterized by a *segment* predicate $\Lambda(x,y)$ or $\Lambda(x,y,z)$ for singly-linked or doubly-linked lists, respectively; see Fig. 2 for an illustration of the semantics of $\mathsf{ls}_{\Lambda(x,y)}(\varepsilon_1,\varepsilon_2)$ and $\mathsf{dls}_{\Lambda(x,y,z)}(\varepsilon_1,\varepsilon_2,\varepsilon'_1,\varepsilon'_2)$ for $\Lambda(x,y) \equiv x \mapsto y$ and $\mathsf{dls}_{\Lambda(x,y,z)} \equiv x \mapsto z * x + 8 \mapsto y$. We note that our list-segment predicates only have two or three free variables, respectively, which prevents the logic from, e.g., describing non-global heap objects shared by list elements. However, more parameters could be introduced in a similar fashion to other works [2]. We have not done so here since it would complicate the notation, and we take this issue as orthogonal to the techniques we propose.

Binary and Unary Operators. uop and bop denote some arbitrary set of binary and unary operators, respectively. We assume this set to include at least the usual operators ($+$, $-$, $*$, &, |, ...) available in low-level languages as well as a special substring operator $\cdot[\cdot,\cdot)$ on byte sequences where $k[i,j)$ for some $k = b_0 \cdots b_{l-1} \in Byte^l$ and $0 \leqslant i \leqslant j \leqslant l$ denotes the byte sequence $b_i \cdots b_{j-1}$. Since we work with variables of fixed size, we basically assume a version of each uop and bop for every possible operand size. We further remark that unary operators uop can be used for modelling the casting to different sizes.

Semantics. We now define the semantics of our SL over SBM triplets $(S,B,M) \in Config$:

$(S,B,M) \models \varepsilon_1 \mapsto \varepsilon_2$ iff

$$\mathsf{dom}(M) = [\llbracket\varepsilon_1\rrbracket_{S,B}, \llbracket\varepsilon_1\rrbracket_{S,B} + \mathsf{size}(\varepsilon_2)) \text{ and } M[\llbracket\varepsilon_1\rrbracket_{S,B}, \llbracket\varepsilon_1\rrbracket_{S,B} + \mathsf{size}(\varepsilon_2)) = \llbracket\varepsilon_2\rrbracket_{S,B}$$

where

$$\llbracket k\rrbracket_{S,B} = k, \llbracket x\rrbracket_{S,B} = S(x), \llbracket\mathfrak{b}(\varepsilon)\rrbracket_{S,B} = \mathfrak{b}_B(\llbracket\varepsilon\rrbracket_{S,B}), \llbracket\mathfrak{e}(\varepsilon)\rrbracket_{S,B} = \mathfrak{e}_B(\llbracket\varepsilon\rrbracket_{S,B}),$$
$$\llbracket\mathsf{uop}\,\varepsilon\rrbracket_{S,B} = \mathsf{uop}(\llbracket\varepsilon\rrbracket_{S,B}), \text{ and } \llbracket\varepsilon_1\,\mathsf{bop}\,\varepsilon_2\rrbracket_{S,B} = \llbracket\varepsilon_1\rrbracket_{S,B}\ \mathsf{bop}\ \llbracket\varepsilon_2\rrbracket_{S,B}$$

$(S, B, M) \models \varepsilon_1 \mapsto k[\varepsilon_2]$ iff
$\mathsf{dom}(M) = [\llbracket\varepsilon_1\rrbracket_{S,B}, \llbracket\varepsilon_1\rrbracket_{S,B} + \llbracket\varepsilon_2\rrbracket_{S,B})$ and $M[\llbracket\varepsilon_1\rrbracket_{S,B} + i] = k$ for all $0 \leqslant i < \llbracket\varepsilon_2\rrbracket_{S,B}$

$(S, B, M) \models \varepsilon_1 \mapsto \top[\varepsilon_2]$ iff $\mathsf{dom}(M) = [\llbracket\varepsilon_1\rrbracket_{S,B}, \llbracket\varepsilon_1\rrbracket_{S,B} + \llbracket\varepsilon_2\rrbracket_{S,B})$

We remark on the difference between the three points-to predicates: the predicate $\varepsilon_1 \mapsto \varepsilon_2$ fixes the exact sequence of bytes ε_2 that is stored from location ε_1 onwards, and the number of bytes is known (the size of ε_2); the predicate $\varepsilon_1 \mapsto k[\varepsilon_2]$ states that there are ε_2 number of bytes stored from location ε_1 onwards (note that the number of bytes ε_2 is symbolic), and each of these bytes equals k; and the predicate $\varepsilon_1 \mapsto \top[\varepsilon_2]$ works in the same way except that the bytes stored are not fixed.

$(S, B, M) \models \varphi_1 * \varphi_2$ iff there are some M_1, M_2 with $M = M_1 \uplus M_2, (S, B, M_i) \models \varphi_i$

$(S, B, M) \models \varphi_1 \vee \varphi_2$ iff $(S, B, M) \models \varphi_1$ or $(S, B, M) \models \varphi_2$

$(S, B, M) \models \mathsf{emp}$ iff $\mathsf{dom}(M) = \varnothing$ $\qquad$ $(S, B, M) \models \mathit{true}$ always holds

$(S, B, M) \models \varepsilon_1 \bowtie \varepsilon_2$ iff $\mathsf{dom}(M) = \varnothing$ and $\llbracket\varepsilon_1\rrbracket_{S,B} \bowtie \llbracket\varepsilon_2\rrbracket_{S,B}$

We point out that pure formulae constrain the heap to be empty. This is typically not required by separation logics that support classical (non-separating) conjunction at least on pure sub-formulae. However, we exclude the classical conjunction in order to simplify the presentation and hence need to constrain the heap of pure formulae to be empty.

$(S, B, M) \models \exists x.\varphi(x)$ iff there is some $v \in \mathit{Val}$
and a fresh variable $u \in \mathit{Var}$ s.t. $(S[u \hookrightarrow v], B, M) \models \varphi(u)$

$(S, B, M) \models \mathsf{ls}_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2)$ iff $(S, B, M) \models \varepsilon_1 = \varepsilon_2$ or
$(S, B, M) \models \varepsilon_1 \neq \varepsilon_2 * \mathsf{true}$ and there is some $\ell \in \mathit{Loc}$
and a fresh variable $u \in \mathit{Var}$ s.t. $(S[u \hookrightarrow \ell], B, M) \models \Lambda(\varepsilon_1, u) * \mathsf{ls}_{\Lambda(x,y)}(u, \varepsilon_2)$

$(S, B, M) \models \mathsf{dls}_{\Lambda(x,y,z)}(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2')$ iff $(S, B, M) \models \varepsilon_1 = \varepsilon_2' * \varepsilon_2 = \varepsilon_1'$ or
$(S, B, M) \models \varepsilon_1 \neq \varepsilon_2' * \varepsilon_2 \neq \varepsilon_1' * \mathsf{true}$ and there is some $\ell \in \mathit{Loc}$ and a fresh variable
$u \in \mathit{Var}$ such that $(S[u \hookrightarrow \ell], B, M) \models \Lambda(\varepsilon_1, u, \varepsilon_2) * \mathsf{dls}_{\Lambda(x,y,z)}(u, \varepsilon_1, \varepsilon_1', \varepsilon_2')$

Satisfiability and Entailment. We say that an SL formula φ is *satisfiable* iff there is a model (S, B, M) such that $(S, B, M) \models \varphi$. We say that an SL formula φ_1 *entails* an SL formula φ_2, denoted $\varphi_1 \models \varphi_2$, iff we have that $(S, B, M) \models \varphi_2$ for every model (S, B, M) such that $(S, B, M) \models \varphi_1$.

Restrictions on the Segment Predicates. From now on, we put further restrictions on the segment predicates $\Lambda(x, y)$ and $\Lambda(x, y, z)$: (1) Λ needs to be of the shape $\exists x_1, \ldots, x_k.\varphi$ for some quantifier-free symbolic heap φ. Intuitively, this condition is required since quantifier-free symbolic heaps are the formulae on which the symbolic execution described in Section 6 is based on and the existential quantification allows to hide some nested data. (2) Λ needs to be *block-closed* in the sense defined below.

Block-closedness. A formula φ is *block-closed* iff, for all $(S, B, M) \models \varphi$ and $\ell \in \mathsf{dom}(M)$, we have that $[\mathfrak{b}(\ell), \mathfrak{e}(\ell)) \subseteq \mathsf{dom}(M)$. Intuitively, block-closedness ensures that all points-to assertions in a formula add up to whole blocks. We require block-closedness in order to ensure that list-segments correspond to our intuition and connect different memory blocks (i.e., we exclude models where multiple or all nodes of list-segments belong to the same block). Technically, the requirement of block-closedness makes it easier to formulate rules for materialisation of list-segment nodes in the abduction procedure and for entailment checking. We leave lifting the restriction of block-closedness for future work. A sufficient condition for block-closedness, which is easy to check, is that all points-to assertions in φ can be organized in groups $\varepsilon_i \mapsto \Upsilon_i$, for $1 \leqslant i \leqslant n$, where Υ represents either y, $k[y]$, or $\top[y]$, such that $\varepsilon_i = \varepsilon_{i-1} + \mathsf{size}(\Upsilon_i)$ for all $1 < i \leqslant n$, and φ implies that $\mathfrak{e}(\varepsilon_1) - \mathfrak{b}(\varepsilon_1) = \sum_{i=1..n} \mathsf{size}(\Upsilon_i)$.

6 Contracts of Functions and Their Generation

Our analysis is based on generating *contracts of functions* along the call tree, starting from its leaves. The contracts summarize the semantics of the functions under analysis. We may also compute multiple contracts for the same function where each contract provides a valid summary of the function; the contracts might, however, differ in the preconditions under which they apply.

6.1 Contracts of Functions

We assume a set of *variables* $Var = PVar \uplus LVar$ that is partitioned into two disjoint infinite set of *program variables* $PVar$ and *logical variables* $LVar$ (also called *ghost* variables). For functions $f(x_1, \ldots, x_n)$ with parameters x_i, we always require $x_1, \ldots, x_n \in PVar$ (we assume that $x_1, \ldots, x_n$ are the only variables occurring in the body of f). To summarize the semantics of a function $f(x_1, \ldots, x_n)$, we use (sets of) *contracts* of the form $\{P\} f(x_1, \ldots, x_n) \{Q\}$ where

- the *pre-condition* P is a quantifier-free symbolic heap, and
- the *post-condition* Q is a disjunction of formulas of the form $\exists U_Q.(Q_{free} * Q_{eq})$ such that Q_{free} is a quantifier-free symbolic heap with $var(Q_{free}) \subseteq LVar$, Q_{eq} is the formula $x_1 = \varepsilon_1 * \cdots * x_n = \varepsilon_n$ for some expressions ε_i with $var(\varepsilon_i) \subseteq LVar$, and $U_Q = (var(Q_{free} * Q_{eq}) \cap LVar) \backslash var(P)$. Note that every disjunct of the post-condition Q describes the heap by a formula over the logical variables (the formula Q_{free}) and fixes the values of the program variables in terms of expressions over the logical variables (the formula Q_{eq}) where all logical variables that do not appear in the pre-condition P are existentially quantified (on the other hand, those logical variables that appear in the pre-condition may be implicitly considered as universally quantified).
- We call a contract *conjunctive* if the post-condition $Q \equiv Q_1 \vee \cdots \vee Q_l$ consists of a single disjunct (i.e., $l = 1$), and *disjunctive* otherwise.

Soundness of contracts. We will now state what it means for a contract to be sound. As usual we stipulate that configurations satisfying the pre-condition lead to configurations satisfying the post-condition. In addition, we also require that we can always add a *frame* to the pre-/post-condition, i.e., a formula describing a part of the heap untouched by the function[10]. Here, a frame F is any symbolic heap with $\mathit{var}(F) \subseteq \mathit{LVar}$. A contract $\{P\}f(x_1,\ldots,x_n)\{Q\}$ is called *sound* iff, for all frames F, all triples (S,B,M) such that $(S,B,M) \models F * P$, and all executions of $f(x_1,\ldots,x_n)$ that start from (S,B,M) and end in some configuration (S',B',M')[11], it holds that $(S',B',M') \models F * Q$.

6.2 Contracts for Basic Statements

We give below contracts for the basic statements of our programming language stated as functions (basic statements may be viewed as special built-in functions). For simplicity (and w.l.o.g.), we assume that it never happens that the same variable appears both at the LHS and RHS of an assignment[12]. Recall that emp is implicit in all otherwise pure constraints (and so we do not need to repeat it):

- Function $\mathtt{assign}(x,y)$ with the body $x := y$:

  $$\{y = Y\}\ \mathtt{assign}(x,y)\ \{x = Y\ *\ y = Y\}.$$

- Function $\mathtt{const}_k(x)$ with the body $x := k$:

  $$\{\mathsf{emp}\}\ \mathtt{const}_k(x)\ \{x = k\}.$$

- Function $\mathtt{load}(x,y)$ with the body $x := *y$:

  $$\{y = Y\ *\ Y \mapsto z\}\ \mathtt{load}(x,y)\ \{x = z\ *\ y = Y\ *\ Y \mapsto z\}$$

  with $Q_{\mathit{free}} \equiv Y \mapsto z$ and $Q_{\mathit{eq}} \equiv x = z\ *\ y = Y$.
- Function $\mathtt{store}(x,y)$ with the body $*x := y$:

  $$\{x = X\ *\ y = Y\ *\ X \mapsto z\}\ \mathtt{store}(x,y)\ \{x = X\ *\ y = Y\ *\ X \mapsto Y\}$$

  with $Q_{\mathit{free}} \equiv X \mapsto Y$ and $Q_{\mathit{eq}} \equiv x = X\ *\ y = Y$.
- Function $\mathtt{malloc}(x,y)$ that either succeeds or fails to allocate memory through $x := \mathtt{malloc}(y)$:

  $$\{y = Y\}\ \mathtt{malloc}(x,y)\ \{x = \mathsf{null}\ *\ y = Y \vee \exists u.\ x = u\ *\ \nu(u,Y)\ *\ y = Y\}$$

  where $\nu(u,Y) = u \mapsto \top[Y] * \mathfrak{b}(u) = u\ *\ \mathfrak{e}(u) = u + Y$. Note that either $Q_{\mathit{free}} \equiv \nu(u,Y)$ and $Q_{\mathit{eq}} \equiv x = u\ *\ y = Y$ or $Q_{\mathit{free}} \equiv \mathsf{emp}$ and $Q_{\mathit{eq}} \equiv x = \mathsf{null}\ *\ y = Y$. A very similar contract can be used for calloc, just with $u \mapsto \top[Y]$ changed to $u \mapsto 0[Y]$. We remark that the contracts for $\mathtt{malloc}$ and calloc are the only disjunctive contracts among the contracts for the basic statements of our programming language.
- Function $\mathtt{free}(x)$ called with the null argument:

  $$\{x = X\ *\ X = \mathsf{null}\}\ \mathtt{free}(x)\ \{x = X\ *\ X = \mathsf{null}\}$$

- Function $\mathtt{free}(x)$ called over a non-null argument:

  $$\{x = X\ *\ X \mapsto \top[y] * \mathfrak{b}(X) = X\ *\ \mathfrak{e}(X) = X + y\}\ \mathtt{free}(x)\ \{x = X\}$$

[10] That is, we directly incorporate the well-known *frame rule* from the separation-logic literature into our notion of soundness. We choose to do so for economy of exposition and for making the paper self-contained. As an alternative one could derive the validity of the frame rule from the fact that all contracts of the basic statements, as stated in Section 6.2, are *local actions* in the sense of [8] (which is equivalent to Lemma 1 stated in this paper).

[11] Note that $\mathsf{dom}(S') = \mathsf{dom}(S)$ and that we have $S'(x) = S(x)$ for all $x \in \mathit{LVar}$ because logical variables do not occur in the program and hence are never updated.

[12] We may assume this because assignments such as $x := *x$ can always be rewritten to the sequence $y := *x; x := y$ (at the cost of introducing a fresh variable y).

Note that a block to be freed may be split into multiple fields at the time of freeing. We, however, do not need to deal with this issue here since the later presented bi-abduction rules will split the LHS of the contract of `free` such that it can match the fragmented block.

- Functions $\mathtt{assign}_{\mathsf{bop}}(x, y, z)$ with the body $x := y \,\mathsf{bop}\, z$ for binary operators bop (and likewise for unary operators uop):

$$\{y = Y \ * \ z = Z\}\ x := y \,\mathsf{bop}\, z\ \{x = Y \,\mathsf{bop}\, Z \ * \ y = Y \ * \ z = Z\}$$

- Function $\mathsf{ptrplus}(x, y, z)$ with the body $x := y \,\mathsf{ptrplus}\, z$ for the case when the result is within the block of the pointer to which an offset is added:

$$\{y = Y \ * \ z = Z \ * \ \varphi_{Y,Z}\}\ x := y \,\mathsf{ptrplus}\, z\ \{x = Y + Z \ * \ y = Y \ * \ z = Z \ * \ \varphi_{Y,Z}\}$$

for $\varphi_{Y,Z} \equiv \mathfrak{b}(Y) \neq 0 \ * \ \mathfrak{b}(Y) = \mathfrak{b}(Y + Z) * \mathfrak{e}(Y) = \mathfrak{e}(Y + Z)$.

- Function $\mathsf{ptrplus}(x, y, z)$ with the body $x := y \,\mathsf{ptrplus}\, z$ for the case when the result points one byte past the block of the pointer to which an offset is added:

$$\{y = Y \ * \ z = Z \ * \ \varphi_{Y,Z}\}\ x := y \,\mathsf{ptrplus}\, z\ \{x = Y + Z \ * \ y = Y \ * \ z = Z \ * \ \varphi_{Y,Z}\}$$

for $\varphi_{Y,Z} \equiv \mathfrak{b}(Y) \neq 0 \ * \ Y + Z = \mathfrak{e}(Y)$.

- Function $\mathsf{ptrsub}(x, y, z)$ with the body $x := y \,\mathsf{ptrsub}\, z$:

$$\{y = Y \ * \ z = Z \ * \ \varphi_{Y,Z}\}\ x := y \,\mathsf{ptrsub}\, z\ \{x = Y - Z \ * \ y = Y \ * \ z = Z\}$$

for $\varphi_{Y,Z} \equiv \mathfrak{b}(Y) \neq 0 \ * \ \mathfrak{b}(Y) \leqslant Z \leqslant \mathfrak{e}(Y)$.

- Function $\mathtt{assume}_{\bowtie}(y, z)$ with the body $\mathtt{assume}(y \bowtie z)$:

$$\{y = Y \ * \ z = Z\}\ \mathtt{assume}(y \bowtie z)\ \{y = Y \ * \ z = Z \ * \ y \bowtie z\}$$

- Function $\mathtt{assert}_{\bowtie}(y, z)$ with the body $\mathtt{assert}(y \bowtie z)$:

$$\{y = Y \ * \ z = Z \ * \ y \bowtie z\}\ \mathtt{assert}(y \bowtie z)\ \{y = Y \ * \ z = Z \ * \ y \bowtie z\}$$

Finally, the contract for `memcpy` is more complex, and we defer it to [22] for space reasons. We now state the soundness of the contracts for the basic statements of our programming language:

▶ **Lemma 1.** *Let stmt be a basic statement and let $\{P\}\ f(x_1, \ldots, x_n)\ \{Q\}$ be a contract for stmt as stated above. Then, the contract is sound, i.e., for all frames F, all configurations (S, B, M) such that $(S, B, M) \models F * P$, and all executions of $f(x_1, \ldots, x_n)$ that start from (S, B, M) and end in some configuration (S', B', M'), it holds that $(S', B', M') \models F * Q$.*

Proof. Direct from the semantics of our programming language as stated in Section 4. ◀

6.3 Contract Generation

We now sketch the generation of contracts for an arbitrary user-defined function $f(x_1, \ldots, x_n)$. Our analysis proceeds along the call tree, starting from its leaves. Hence, we can assume to already have computed contracts for nested function calls. (Recall that, in this paper, we limit ourselves to non-recursive functions.) We derive contracts by (forward) symbolic execution. The symbolic execution starts at the beginning of f and maintains a pair of formulae P and Q, representing the so-far computed part of the *pre-condition* of the function f and the *current symbolic state*. The symbolic execution will guarantee that configurations that satisfy P lead to configurations satisfying Q after executing the so-far analysed statements. P and Q will change throughout the symbolic execution because we keep restricting the precondition P and advancing the symbolic state Q. The symbolic execution is set up such that the program variables $x_1, \ldots, x_n$ may be updated, while all other variables will never be modified (but, of course, fresh variables may be introduced and assigned at any time). The symbolic execution is initialised by introducing fresh logical variables $X_1, \ldots, X_n$ and setting $P \equiv Q \equiv x_1 = X_1 * \cdots * x_n = X_n$. In each step, the symbolic execution needs to solve a bi-abduction problem in order to advance the symbolic state Q with regard to the contract of a function call or a basic statement. The bi-abduction procedure might discover that the current symbolic state Q does not suffice to safely call the function, in which case either a strengthening of the precondition P is returned or the procedure fails. We describe our procedure for solving the abduction problem (the procedure for discovering missing pre-conditions) in the next section, and refer the reader to our technical report [22] for the full bi-abduction procedure. In order to derive sound contracts, we follow the two-round analysis approach of [7]: The first round (called `PreGen` in [7]) infers a set of pre-/post-condition pairs (P, Q), but there is no guarantee about the soundness of the inferred (P, Q). For each pre-/post-condition pair (P, Q) computed in the first round, the second round (called `PostGen` in [7]) discards the post-condition Q and re-starts the symbolic execution from the pre-condition P *not allowing the strengthening of the pre-condition throughout the symbolic execution*, which either fails or results in a set of pre-/post-condition pairs $(P, Q_1), \ldots, (P, Q_l)$. In the latter case, we return $(P, Q_1 \vee \cdots \vee Q_l)$, which is guaranteed to be a sound contract.

We refer an interested reader to our technical report [22] for the details on how we implement the two-round analysis of [7] and for accompanying examples.

7 Bi-Abduction Procedure

We now state our rules for computing a solution to the abduction problem. In the below rules, we will use the notation $\varphi * [M] \rhd \psi$ to denote that we are deriving the solution M to the abduction problem $\varphi * [?] \models \psi$. The rules are to be applied in the stated order.[13]

We start with a rule allowing us to learn *missing pure constraints.*

$$\texttt{learn-pure}\ \frac{\varphi * \pi * [M] \ \rhd\ \psi}{\varphi * [\pi * M] \ \rhd\ \psi * \pi}\ \pi \text{ pure}$$

The `match` rule presented below allows one to *match points-to predicates* from the LHS and RHS that have the same source location ($\varepsilon = \varepsilon'$) and points-to fields ζ, ζ' of the same size. Then we learn that the target fields are the same too. We note here that this rule is as a special case of the `split-pt-pt-right` rule presented further on, but we show it here as an easy case to start from. We discharge entailment checks of the form $\varphi_1 \models \varphi_2 * true$ where φ_2 is a pure formula (e.g. $\varepsilon = \varepsilon'$) by checking unsatisfiability of the formula $\psi_1 \wedge \neg\psi_2$ where ψ_i is a translation of the SL formula φ_i to bitvector logic. The translation procedure is sketched in our technical report [22].

$$\texttt{match}\ \frac{\varphi * \zeta = \zeta' * [M] \ \rhd\ \psi}{\varphi * \varepsilon \mapsto \zeta * [\zeta = \zeta' * M] \ \rhd\ \psi * \varepsilon' \mapsto \zeta'}\ \mathsf{size}(\zeta) = \mathsf{size}(\zeta') \text{ and } \varphi \models \varepsilon = \varepsilon' * true$$

[13] As for non-determinism within single rules, which can sometimes be applied in multiple ways, our implementation currently uses the first applicable option (with backtracking to the other options only in case that the first option turns out to result in an unsatisfiable abduction strategy). A better strategy is an open question for future research.

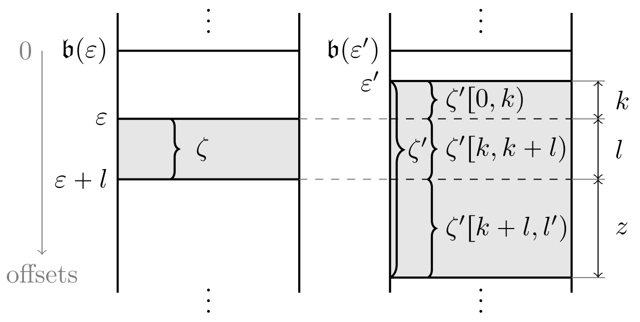
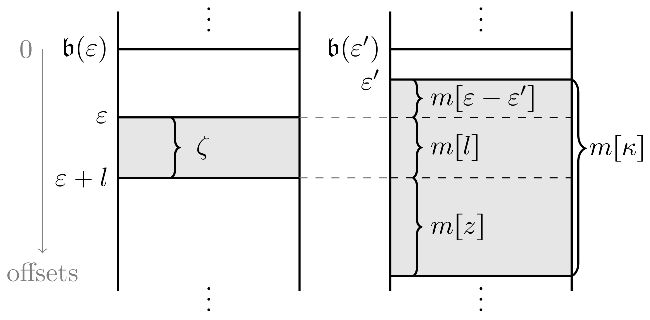


Figure 3 An illustration of the `split-pt-pt-right` rule where $z = l' - k - l$.



0
$\mathfrak{b}(\varepsilon)$
ε
$\varepsilon + l$
offsets
ζ
$\mathfrak{b}(\varepsilon')$
ε'
$m[\varepsilon - \varepsilon']$
$m[l]$
$m[z]$
$m[\kappa]$

Figure 4 An illustration of the `split-pt-bl-right` rule where $z = \kappa - (\varepsilon - \varepsilon') - l$.

As illustrated in Fig. 3, the next presented `split-pt-pt-right` rule allows one to deal with pointers ε, ε' to fields ζ, ζ' that lie at possibly different addresses but within blocks of the same base address. Moreover, the RHS target field ζ' can be larger. In this case, the field ζ' is *split* to three *byte sequences* $\zeta'[0,k)$, $\zeta'[k,k+l)$, and $\zeta'[k+l,l')$, some of which can be empty, and the middle byte sequence is matched with the LHS target field ζ. (We recall that $k[i,j)$ denotes the substring of k that starts at index i and ends at index j.)

$$\texttt{split-pt-pt-right}\ \frac{\varphi * \zeta = \zeta'[k,k+l) * [M] \ \rhd\ \psi * \varepsilon' \mapsto \zeta'[0,k) * (\varepsilon + l) \mapsto \zeta'[k+l,l')}{\varphi * \varepsilon \mapsto \zeta * [\zeta = \zeta'[k,k+l) * M] \ \rhd\ \psi * \varepsilon' \mapsto \zeta'}\ C$$

In the above rule, the condition C requires that there are some $k, l, l' \in \mathbb{N}$ with $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \varepsilon = \varepsilon' + k \ * \ true$, $\mathsf{size}(\zeta) = l$, $\mathsf{size}(\zeta') = l'$, and $l + k \leqslant l'$. We note that, in the above formulation of the rule `split-pt-pt-left`, we assume $0 < k$ and $k + l < l'$ in order to avoid cluttering the rule by additional case distinctions; in the case of $0 = k$ or $k + l = l'$, however, we need to remove $\varepsilon' \mapsto \zeta'[0,k)$ or $(\varepsilon + l) \mapsto \zeta'[k+l,l')$, respectively, from the RHS of the premise of the rule. There is a symmetric rule `split-pt-pt-left` for the LHS.

The `split-pt-bl-right` rule presented below and illustrated in Fig. 4 is an analogy of the rule `split-pt-pt-right` presented above, but, this time, with the RHS field, which is being split, of non-constant size. The rule covers both types of such fields that we allow: sequences of bytes of undefined values (then $m = \top$ in the rule) or sequences of the same byte (then $m \in \mathit{Byte}$).

$$\texttt{split-pt-bl-right}\ \frac{\varphi * \chi * [M] \ \rhd\ \psi * \varepsilon' \mapsto m[\varepsilon - \varepsilon'] * (\varepsilon + l) \mapsto m[z] * K}{\varphi * \varepsilon \mapsto \zeta * [\chi * M] \ \rhd\ \psi * \varepsilon' \mapsto m[\kappa]}\ C$$

Above, we require that $m = \top$ and $\chi \equiv \mathsf{emp}$, or $m \in \mathit{Byte}$ and $\chi \equiv \zeta = m^l$. Further, $\mathsf{size}(\zeta) = l$, C requires that $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \varepsilon' \leqslant \varepsilon \ * \ \varepsilon + l \leqslant \varepsilon' + \kappa \ * \ \mathit{true}$, z is some fresh variable with $\mathsf{size}(z) = N$, and $K \equiv z = \kappa - (\varepsilon - \varepsilon') - l$. There is a symmetric rule `split-pt-bl-left` for the LHS.

We now present an analogy of the above rule for the case when we need to split a field of constant size that appears on the RHS. In order to be able to split the RHS field we will also require the LHS field to be of constant size.

$$\texttt{split-bl-pt-right}\ \frac{\varphi * \chi * [M] \ \rhd \ \psi * \varepsilon' \mapsto \zeta'[0,k) * (\varepsilon + l) \mapsto \zeta'[k+l, l')}{\varphi * \varepsilon \mapsto m[\kappa] * [\chi * M] \ \rhd \ \psi * \varepsilon' \mapsto \zeta'}\ C$$

In the above rule, the condition C requires that there are some $k, l, l' \in \mathbb{N}$ with $\varphi \models \kappa = l \ * \ \mathit{true}$, $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \varepsilon = \varepsilon' + k \ * \ \mathit{true}$, $\mathsf{size}(\zeta') = l'$, and $k + l \leqslant l'$. In the rule, either $m = \top$ and $\chi \equiv \mathsf{emp}$, or $m \in \mathit{Byte}$ and $\chi \equiv \zeta'[k, k+l) = m^l$. There is a symmetric rule `split-bl-pt-left` for the LHS.

We are finally getting to the `split-bl-bl-right` rule that matches two fields that are both of non-constant sizes while splitting the RHS field if need be.

$$\texttt{split-bl-bl-right}\ \frac{\varphi * [M] \ \rhd \ \psi * \varepsilon' \mapsto m'[\varepsilon - \varepsilon'] * \varepsilon + \kappa \mapsto m'[z] * K}{\varphi * \varepsilon \mapsto m[\kappa] * [M] \ \rhd \ \psi * \varepsilon' \mapsto m'[\kappa']}\ C$$

In the rule, either $m' = \top$ or $m = m'$. Further, C is the condition that requires $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \varepsilon' \leqslant \varepsilon \ * \ \varepsilon + \kappa \leqslant \varepsilon' + \kappa' \ * \ \mathit{true}$ and $K \equiv z = \kappa' - (\varepsilon - \varepsilon') - \kappa$. As before, there is also a symmetric rule `split-bl-bl-left` for splitting on the LHS.

Next, we present a rule that allows one to match a points-to predicate on the LHS against a singly-linked list segment on the RHS. In fact, the rule does not directly perform the matching, but it facilitates it by *materialising* the first cell out of the list segment. The matching itself (possibly combined with splitting) is then performed by the above rules. We expect that the cells of the list segment are described using a formula of the form $\Lambda(x,y) \equiv \exists u_1, \ldots, u_k.\lambda(x, y, u_1, \ldots, u_k)$.

$$\texttt{slseg-pt-ls-right}\ \frac{\varphi * \varepsilon \mapsto \zeta * [M] \ \rhd \ \psi * \lambda[\varepsilon'/x, z/y, z_1/u_1, \ldots, z_k/u_k] * \mathsf{ls}_{\Lambda(x,y)}(z, \zeta')}{\varphi * \varepsilon \mapsto \zeta * [M] \ \rhd \ \psi * \mathsf{ls}_{\Lambda(x,y)}(\varepsilon', \zeta')}\ C$$

Above, C is the condition that $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \mathit{true}$ and $z, u_1, \ldots, u_k$ are some fresh variables.

We next present a version of the above rule for the case of a list segment on the LHS. Note that, in this case, we must require the list segment be non-empty. In the rule, C is the condition that $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \ * \ \varepsilon \neq \zeta \ * \ \mathit{true}$ and $z, u_1, \ldots, u_k$ are some fresh variables.

$$\texttt{slseg-pt-ls-left}\ \frac{\varphi * \lambda[\varepsilon/x, z/y, z_1/u_1, \ldots, z_k/u_k] * \mathsf{ls}_{\Lambda(x,y)}(z, \zeta) * [M] \ \rhd \ \psi * \varepsilon' \mapsto \zeta'}{\varphi * \mathsf{ls}_{\Lambda(x,y)}(\varepsilon, \zeta) * [M] \ \rhd \ \psi * \varepsilon' \mapsto \zeta'}\ C$$

The following rule allows one to remove from the LHS a list segment that forms an initial part of a list segment that appears on the RHS. The condition C requires that $\varphi \models \varepsilon = \varepsilon' \ * \ \mathit{true}$ and that $\Lambda(x,y) \models \Lambda'(x,y)$[14].

[14] We note that this kind of entailment query cannot be discharged in the way we sketched above for the case when the RHS of the entailment is a pure formula (intuitively, one would need some negation over SL). However, such queries can be discharged by a slight modification of the bi-abduction procedure presented in this section – for details see our technical report [22].

$$\texttt{slseg-ls-ls}\frac{\varphi * [M] \;\rhd\; \psi * \mathsf{ls}_{\Lambda'(x,y)}(\zeta,\zeta')}{\varphi * \mathsf{ls}_{\Lambda(x,y)}(\varepsilon,\zeta) * [M] \;\rhd\; \psi * \mathsf{ls}_{\Lambda'(x,y)}(\varepsilon',\zeta')}\; C$$

The further rule allows one to remove a possibly empty list segment from the RHS. A corresponding rule for list segments of the LHS is only needed for entailment checking (cf. [22]).

$$\texttt{slseg-remove-right}\frac{\varphi * [M] \;\rhd\; \psi}{\varphi * [M] \;\rhd\; \psi * \mathsf{ls}_{\Lambda(x,y)}(\varepsilon,\zeta)}\; \varphi \models \varepsilon = \zeta \;*\; \mathit{true}$$

We have similar rules for *doubly-linked lists* as the ones stated above, which we omit here for ease of exposition (we point out that `dllseg-pt-ls-left` and `dllseg-pt-ls-right` come in two versions because a doubly-linked list can be unrolled from the left as well as from the right).

Next, we state a rule that allows one to *finish* the abduction process.

$$\texttt{learn-finish}\frac{}{\varphi * [\psi] \;\rhd\; \psi * \mathit{true}}\; \varphi * \psi \text{ is satisfiable}$$

The side condition "$\varphi * \psi$ is satisfiable" is intended to ensure that the abduction solution ψ does not lead to useless contracts: a contract $\{\varphi * \psi\}\, f(\cdots)\, \{\cdots\}$ with $\varphi * \psi$ unsatisfiable does not have a configuration that satisfies its pre-condition! Unfortunately, we only have an approximate procedure for checking the satisfiability of symbolic heaps (see our technical report [22]). However, contracts with an unsatisfiable pre-condition are still sound. Hence, we employ the best-effort strategy of using our approximate procedure to prevent as many useless abduction solutions as possible in order to minimize the number of inferred contracts.

Finally, we state two rules of "last resort" that involve quite some guessing and hence can mislead the abduction process and make it fail (or lead to its exponential explosion when all possible variants of applying the rules are attempted). Intuitively, they allow one to *claim equal* fields whose *equality* is not known, but whose *disequality* is not known either (moreover, in the weaker case, one also checks that it can be shown that the fields lie within the same memory block).

$$\texttt{alias-weak}\frac{\varphi * \chi(\varepsilon) * \varepsilon = \varepsilon' * [M] \;\rhd\; \psi * \chi'(\varepsilon')}{\varphi * \chi(\varepsilon) * [\varepsilon = \varepsilon' * M] \;\rhd\; \psi * \chi'(\varepsilon')}\; C_1$$

$$\texttt{alias-strong}\frac{\varphi * \chi(\varepsilon) * \varepsilon = \varepsilon' * [M] \;\rhd\; \psi * \chi'(\varepsilon')}{\varphi * \chi(\varepsilon) * [\varepsilon = \varepsilon' * M] \;\rhd\; \psi * \chi'(\varepsilon')}\; C_2$$

In the rules, $\chi(x)$ and $\chi'(x)$ are any predicates of the form $x \mapsto \_$, $\mathsf{ls}\_(x,\_)$, $\mathsf{dls}\_(x,\_,\_,\_)$, or $\mathsf{dls}\_(\_,\_,x,\_)$. Further, C_1 is the condition that $\varphi \models \mathfrak{b}(\varepsilon) = \mathfrak{b}(\varepsilon') \;*\; \mathit{true}$ and that *not* $\varphi \models \varepsilon \neq \varepsilon' \;*\; \mathit{true}$. On the other hand, C_2 requires that not $\varphi \models \varepsilon \neq \varepsilon' \;*\; \mathit{true}$ only.

The *alias-weak/strong* rules are used in the following situations:

- There is *no other applicable rule*. Instead of failing due to the impossibility of applying other rules, we try to introduce an alias (if possible, by the *alias-weak* rule) and continue with the abduction using the *match*, *split*, or *slseg/dllseg* rules.
- We wish to infer *multiple abduction solutions*. In such a case, whenever *learn-finish* is applicable, we use it to derive one abduction solution, record it, revert *learn-finish*, and then try to derive other solutions by applying an *alias* rule, followed by applying the other rules again.

We now state the correctness of the abduction procedure:

▶ **Theorem 2.** *Let M be any solution computed by the abduction rules, i.e., we have $\varphi * [M] \rhd \psi$. Then, $\varphi * M \models \psi$.*

Proof. We prove the property by induction on the number of rule applications. We observe that the claim holds for the axiom, i.e., the rule `learn-finish`). We further note that, for all non-axiomatic rules of the shape

$$\text{rule-name}\,\frac{\varphi' * [M'] \;\rhd\; \psi'}{\varphi * [M] \;\rhd\; \psi}\; C,$$

we have that $\varphi' * M' \models \psi'$ implies $\varphi * M \models \psi$ (under the condition C). Hence, the claim holds. ◀

Moreover, we observe that the antiframe M is guaranteed to be a quantifier-free symbolic heap in case the input φ to the abduction procedure is a quantifier-free symbolic heap (the abduction rules maintain this shape of φ).

▶ **Example 3.** We consider the abduction problem

$$X \mapsto a * X + 8 \mapsto z * [?] \models Y \mapsto u * Y + 8 \mapsto w * u \mapsto v * X = Y * \mathit{true}.$$

Its solution by our abduction rules looks as follows:

$$\texttt{learn-pure}\,\dfrac{\texttt{match}\,\dfrac{\texttt{match}\,\dfrac{\texttt{learn-finish}\,\dfrac{}{X = Y * a = u * z = w * [u \mapsto v] \;\rhd\; u \mapsto v * \mathit{true}}}{X + 8 \mapsto z * X = Y * a = u * [z = w * u \mapsto v] \;\rhd\; Y + 8 \mapsto w * u \mapsto v * \mathit{true}}}{X \mapsto a * X + 8 \mapsto z * X = Y * [a = u * z = w * u \mapsto v] \;\rhd\; Y \mapsto u * Y + 8 \mapsto w * u \mapsto v * \mathit{true}}}{X \mapsto a * X + 8 \mapsto z * [X = Y * a = u * z = w * u \mapsto v] \;\rhd\; Y \mapsto u * Y + 8 \mapsto w * u \mapsto v * X = Y * \mathit{true}}$$

8 Implementation and Experimental Evaluation

We have implemented the proposed techniques in a prototype tool called Broom. Its source code is publicly available[15] under GNU GPLv3. The tool itself is implemented in OCaml. The SMT queries produced by the tool are answered using the Z3 solver [12]. The front-end of Broom is based on Code Listener [15], a framework providing access to the intermediate code of a compiler (as, e.g., `gcc`).

Our approach requires one to answer entailment queries $\varphi_1 \models \varphi_2$ at several points. If φ_2 is pure, we translate $\varphi_1 \wedge \neg\varphi_2$ from SL into the bitvector theory and ask the underlying SMT solver. However, this cannot be easily done when φ_2 contains a spatial predicate (our fragment of SL is not closed under negation). While it might be possible to develop a general (sound and complete) entailment procedure, e.g., extending [26], we decided to use an approximation based on similar principles as our bi-abduction procedure. We give details of the translation of SL to the bitvector theory as well as of our more general approximated entailment procedure in our technical report [22].

We note that, in the implementation of Broom, we relaxed the requirement put on the *ptrplus* operation of our minilanguage (Sec. 4), which requires that the pointer resulting from the expression $y + z$ stays within the allocated block – i.e., $\mathfrak{b}(S_B(y)) \leqslant S_B(y) + S_B(z) \leqslant \mathfrak{e}(S_B(y))$. According to the C standard, the relaxation of this condition leads to

[15] `https://pajda.fit.vutbr.cz/rogalew/broom`

undefined behaviour, but it is often used in low-level system code as, e.g., in the Linux list implementation. In our implementation, we allow pointers to have values outside of the allocated blocks, but we explicitly track their *provenance* (i.e., the basis wrt which they are defined) using the $\mathfrak{b}$ predicate.

Broom comes with a number of parameters that can be set for the analysis, with the most important being the following ones:

- *Solver timeouts:* Timeouts of the underlying solver can be set separately for symbolic execution, widening, and formula simplification. Using a timeout, one can balance between speed and precision. With a lower timeout, the analysis is faster, but some functions need not be fully analysed due to an abduction or widening failure. The default timeouts used in our later presented experiments are 2000ms for the symbolic execution, 200ms for widening, and 100ms for formula simplification.
- *Number of loop unfoldings:* A limit on the number of loop unfoldings is used to stop the loop analysis when a fixpoint is not computed within a given number of loop iterations. Then, either no contract or partial contracts are returned. The default value used in our experiments is 5.
- *Abduction strategy:* The abduction strategy can be set as follows: In the standard configuration, it follows the order of rules presented in Sec. 7. The tool also supports an alternative strategy where the `alias-weak/strong` rules are used to derive multiple abduction solutions as discussed in Sect. 7. This may lead to an exponential blow-up in the number of contracts for particular functions (a lot of them useless) together with a blowup of the running time. On the other hand, this strategy allows us to fully verify some of our most complicated code fragments (namely, the intrusive lists discussed below). As a part of our future research, we would like to study some heuristically-driven application of this strategy that would not explore so many useless contracts.

Finally, we would like to stress that Broom is now in a stage of a very early prototype, intended mainly to illustrate the theoretical potential of our technique, with huge space for performance optimizations. As a primary source of possible optimisations, we see the way how Broom interacts with the SMT solver (the cost of SMT queries represents a very significant part of the cost of the entire analysis). One way that we see as highly promising for optimisations in this direction is to use static pre-evaluation of some SMT queries – if one can statically evaluate a query, an expensive solver call can be avoided. This can significantly limit the number of SMT queries and improve the running time. We have already partially implemented some static pre-evaluation for the $\varphi \models \varepsilon = \varepsilon' * true$ queries within `match`/`split` abduction rules, which alone reduced the running time by 25 % at some examples. Further optimization possibilities then lie, e.g., in incremental solving, caching solver results, and/or introducing heuristics to decrease the amount of nondeterminism in the abduction rules. As for the last mentioned possibility, especially in the case of the `match`/`split` rules there can be several candidate predicates $\varepsilon \mapsto \zeta$ on the LHS and several candidate predicates $\varepsilon' \mapsto \zeta'$ on the RHS, which one needs to consider, and it would be very helpful to have some guidance in this process.

8.1 Experiments

We evaluate our tool Broom on a set of experiments in which we analyse various fragments of list manipulating code. Since Broom is in a highly prototypical stage, we do not venture into analysing large code bases. Instead, we concentrate on shorter but complex code highlighting what our approach implemented in the tool can handle (and what other tools do typically not manage).

The considered code was pre-processed in the following ways: (1) All appearances of the so-far unsupported constructions `&var` and `var.next` were replaced by `p_var` and `p_var->next`, respectively, where `p_var = alloca(sizeof(*p_var))`. (2) We replaced `for` loops with integer bounds by non-deterministic `while` loops because our abstraction and entailment are currently very limited when working with integers. Both of the above is planned to be resolved within our future work. Further, we analysed all the code assuming that heap allocation always succeeds.

The experiments were run on a machine with an Intel i7-4770 processor with 32 GiB of memory. The current implementation of Broom uses a single core only. We compare our results with those of Infer v1.1.0[16] and Gililan (PLDI'20 version)[17], which are the only tools we are aware of that can analyse at least some of the code we are interested in. We note that Infer was running with debug information enabled (using the command `infer run --debug`) as we wanted to manually check the obtained contracts. The debug option may increase the running time of Infer, but, as one can see in Table 1, the running times are not an issue for Infer.

Table 1 presents a comparison of the results obtained using Broom, Infer, and Gillian on our collection of list-manipulating code fragments.[18] To get the results, Broom was used with its standard abduction strategy where the `alias-weak`/`alias-strong` rules are used only if no other rule is applicable. For each of the cases, the table gives first the total number of functions that the benchmark consists of. Next to it, separately for Broom, Infer, and Gillian, we give the time the tools took for the analysis. Further, we list the number of functions for which the respective tool produced a non-trivial contract. There are up to three numbers in the form *a*/*b*/*c* (*b* or *c* can be omitted), representing the number of functions for which the respective tool computes (a) complete contracts, (b) sound but only partial contracts, and (c) error contracts – i.e., preconditions under which a given function is bound to fail, which are provided by Gillian only. Finally, we also provide a remark whether the tool reported some error (or whether it itself hit some internal error). The expected and really obtained analysis results are encoded as follows (including internal errors of an analyser): OK= *no error found*[19], DF= *double free*, ML= *memory leak*, IE= *internal error*, PE= *internal parsing error*.

We now discuss the individual cases in more detail – when doing so, we concentrate on comparing the results of Broom with those of Infer that can get somewhat closer to the results of Broom:

- `circ-DLL`: This example deals with a simple implementation of *circular doubly-linked lists* (whose part is, in fact, used as the running example in Fig. 1). The code includes functions for inserting the first element, inserting another element after an existing one, and for removing elements. Apart from that there is a higher-level function that inserts the first element, the second element, and them removes one of them.[20] The code contains

[16] `https://github.com/facebook/infer/releases/tag/v1.1.0`

[17] `https://github.com/GillianPlatform/Gillian/releases/tag/PLDI20`

[18] All the code is available together with our tool.

[19] We note that, as far as our experience reaches, Gillian produces its error contracts whenever there is a risk of a null-pointer dereference. In many cases, e.g., in the Linux list library, the error summaries provide a correct result, which, however, does not take into account the fact that the library is designed such that the appropriate functions are never called with a null argument. At the same time, Gillian may miss real, higher-level errors present in the code, which were those we expected to be reported. In such cases, we say in the table in the column for obtained results that the (expected) error was not found.

[20] This function can be viewed as an analysis harness while we were stressing that our analysis does not

Table 1 Experiments with the standard abduction strategy of Broom and a comparison with results obtained from Infer and Gillian.

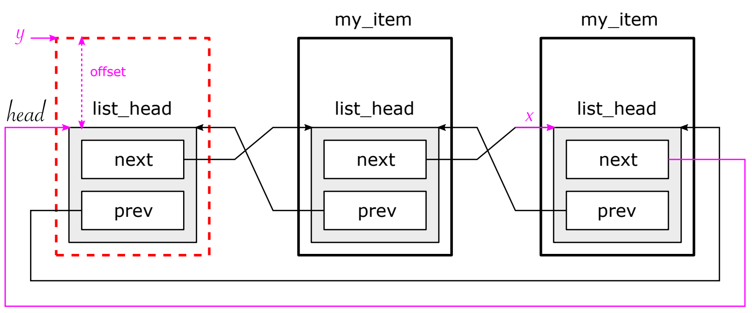
Name	Exp. result	Fncs total	Broom			Infer			Gillian		
			T [s]	Fncs contr	Res	T [s]	Fncs contr	Res	T [s]	Fncs contr	Res
circ-DLL	ML	4	6	4	ML	0.5	1/1	IE	1.2	1/0/2	OK
circ-DLL-err	DF	4	6	4	DF	0.5	1/1	IE	1.2	1/0/2	OK
circ-DLL-embedded	OK	4	9	4	OK	0.5	1/2	IE	0.6	0	PE
Linux-list-1	ML	11	56	10	ML	1.5	2/3	OK	0.6	0	PE
Linux-list-2	OK	11	42	11	OK	0.7	1/6	IE	0.6	0	PE
Linux-list-2-err	ML	11	28	11	ML	0.6	1/6	IE	0.6	0	PE
Linux-list-all	OK	23	267	21/2	OK	1.0	7/15	IE	44	8/0/9	OK
intrusive-list	OK	15	99	10/5	OK	0.7	4/3	OK	0.6	0	PE
intrusive-list-min	OK	9	45	6/2	OK	0.7	1/3	IE	0.6	0	PE
intrusive-list-smoke	OK	20	133	10/5	OK	0.9	4/3	OK	0.6	0	PE

no pointer arithmetic nor any other advanced features. It is intended to show that even in such a case our abduction rules restated wrt [6, 7] can bring some advantage. Namely, this is a consequence of that our use of the per-field separation allows us to cover more shapes of the data structures within a single contract. Indeed, as discussed already in Sect. 2, it produces a single contract for insertion into a cyclic list with one element and with more elements. Infer cannot use the same reasoning and since it primarily favours scalability, it will come with a contract for inserting into lists with at least two elements. Consequently, it then fails to analyse the top level function. As for the memory leak reported by Broom, it is a real error caused by that one of the introduced elements is not deleted.

- `circ-DLL-err` is a variation on `circ-DLL` into which we introduced a double-free error.
- `circ-DLL-embedded` is another variation on `circ-DLL` in which the list implementation from `circ-DLL` is used as a basis of a simple intrusive list in which the list structure with the linking fields from `circ-DLL` is nested into a larger data structure.
- `Linux-list-1` is our first experiment with intrusive lists in the form they are used in the Linux kernel (for some more impression about Linux lists, see Fig. 5). This particular code comes in particular from the benchmark suite of the Predator analyser [16].[21] The code contains multiple different functions for initialisation of the lists, for inserting into it, and for traversing the lists. The top-level code that is present then creates a circular Linux list nested into another circular Linux list. As can be from Fig. 5, the code involves pointer arithmetic (even in a form not supported by the C standard), and the use of nested structures leads to an application of our block splitting rules. The only function that Broom fails to handle is the function for traversing the entire list – the reason is that our so-far quite simple implementation of list abstraction fails in this case, and the otherwise correct computation diverges (which we, however, believe to be solvable in the

need such a harness. Here, we would like to stress that this indeed holds – none of the considered tools needs (nor in any way uses) the top-level function to be able to analyze the other functions. We use the harness as a model of any higher-level code using the list. Moreover, it allows us to show that the contracts that got generated for the particular functions are not complete enough, which shows up in the inability of the appropriate tool to analyse the higher-level functions.

[21] We note here that Predator can analyse the code, but – unlike Broom, Infer, or Gillian – it entirely relies on that the code is closed, i.e., it comes with a main function and has no further inputs.



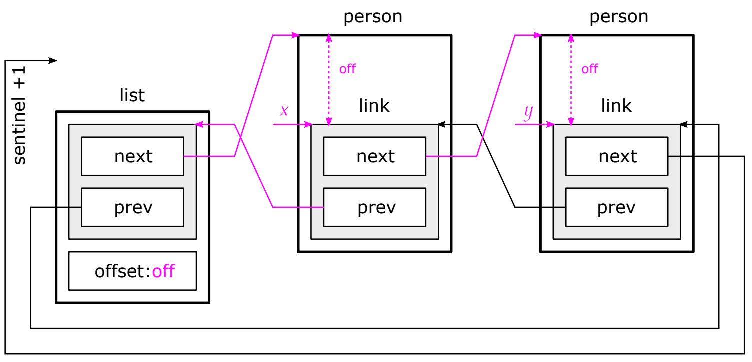
```
// moving to the next item
y = (struct my_item *)(
  (char *)x->next -
  offsetof(struct my_item,
           link)
  );
// testing, if it is head
if (head != &y->link) ...
```

Figure 5 An illustration of the Linux list data structure. The gray boxes represent the linking structure `list_head` that is nested into a user-defined structure `my_item`, whose instances the user needs to be linked into a list. List-manipulating functions know nothing about the user-defined structures: they work with the linking structures only. The user data are accessible through pointer arithmetic only. Note that the head node of the list does *not* have the user-defined envelope. The code shown on the right illustrates how the list is traversed. Note also that when one passes from the last element pointed by `x` to its successor (hence back to the head), the involved pointer arithmetic causes that the pointer `y` will be pointing out of the allocated space, which is, however, correct since it will never be dereferenced (just used for further pointer arithmetic).

close future – indeed, we can get fully inspired by the abstraction used in Predator; the concrete list abstraction used is not specific for our approach). The memory leak reported is a real one – it comes from the top-level function that does not destroy the list.

- `Linux-list-2` is a variation on the above case. It contains functions for an initialisation of a Linux list, inserting elements at its tail, and for deleting the elements. The top-level function initializes the list, inserts several elements, traverses the elements one by one, and deletes them.
- `Linux-list-2-err` is a variation on `Linux-list-2` where one of the inserted elements is not deleted and hence a memory leak is caused.
- `Linux-list-all` contains the entire collection of functions defined for working with Linux lists without any top-level function. The collection includes functions for different kinds of insertion of elements, removal of elements, swapping of elements (both within a list and between lists), moving to the end or to another list, rotation, splicing, etc. We can see that Broom produced complete contracts for many more of the functions. The contracts from Infer often do not cover cases of lists of length 0 or 1. In one of the remaining cases, Infer produced no result; and for the last one, it produced a partial result (that appears not to cover one of the branches of the function).
- `intrusive-list` is the intrusive list library[22]. See Fig. 6 for an illustration how the data structure and the code looks like. Apart from features seen already above (pointer arithmetic and a need to deal with linking fields embedded into larger structures with a need to apply block splitting), the code contains also *bit-masking*. In particular, one bit of the next pointers is used to mark pointers back to the head node, thus effectively marking the "end" of the circular list. A further intricacy of the code is that the insertion into the list touches three nodes that may be different but that may also collapse into a single node. In the case of the Linux list, we have mentioned a similar situation but with two nodes only. Having three nodes that possibly collapse is not only beyond the

[22] Described in the Patrick Wyatt's blog post "Avoiding game crashes related to linked lists", `http://www.codeofhonor.com/blog/avoiding-game-crashes-related-to-linked-lists`, on September 9th, 2012, and implemented in `https://github.com/robbiev/coh-linkedlist`.



```
// offset from
// a node pointer
// to a link structure
size_t off = (size_t) x -
  ((size_t)x->prev->next
                   & ~1);
// link of the next node
y = (link *)(
  (size_t)lnk->next
               & ~1 + off
  );
```

Figure 6 An illustration of the intrusive list data structure. The code fragment shown to the right of the figure gives the code used in the function `link_get_next` to obtain the linking structure of the next node. Note the use of the pointer arithmetic including bit-masking (to clear the bit whose bit-masking on the next pointers is used to denote the sentinel node of the list).

capabilities of Infer but also Broom if it is used with its basic abduction strategy. This is the reason why some of the contracts produced by Broom are also not complete in this case (e.g., effectively allowing insertion into a list with more than one node only). We will, however, show below that Broom can resolve the problem when using more power of the `alias-weak`/`alias-strong` rules, though for the price of quite increased runtime requirements. As for Infer, it is clearly visible that its coverage of the functions is much weaker (interestingly, we noticed that it completely ignored the bit-masking when deriving some of the contracts).

- `intrusive-list-min` contains a subset of the functions considered above (for initializing a list, inserting an element, removing an element) together with a top-level function utilising these functions. Essentially, the intention here was to create an as small as possible example of the given kind already problematic for Infer. Again, even Broom cannot handle it fully under its standard abduction strategy.
- `intrusive-list-smoke` contains the entire intrusive list library from above together with several top-level functions provided by the author to test the library. The tests use structures modelling some personal records to be linked into a list via the embedded linking structures. They create a few such records, link them into a list, traverse them (forward/backward), and destroy the list.

We now proceed to our experiments with the `alias-weak`/`alias-strong` rules. As we have said above, these rules involve a lot of guessing. Hence, if they are used to explore various possible abduction solutions based on different aliasing scenarios, the running time may grow considerably, but it may resolve situations that are otherwise not resolved. To confirm this, we have applied Broom with the strategy of using the `alias-weak`/`alias-strong` rules to explore different possible abduction solutions with different possible aliasing scenarios on the intrusive list case study. The results are shown in Table 2. The first row concerns the experiment `intrusive-list-min` discussed already above. At that time, we noted that Broom could not fully handle some of the intrusive list functions since they required it to merge three possibly independent nodes into a single one. As can be seen in Table 2, with the help of the `alias-weak`/`alias-strong` rules, Broom does fully manage even this problem (though the runtime grew a lot). The next two rows – `intrusive-list-min-2` and `intrusive-list-min-3` – are variations on the previous case where we intentionally introduced some bugs, which were correctly discovered. Finally, the last row shows a

Table 2 Experiments with `alias-weak/alias-strong` in Broom.

Name	Expected result	Fncs total	T [m]	Funcs contr	Res
intrusive-list-min	no error	9	46	9	no error found
intrusive-list-min-2	memory leak	9	47	9	memory leak
intrusive-list-min-3	double free	9	49	9	double free
intrusive-list-smoke	no error	20	505	16	no error found

significant improvement even for the entire library of intrusive lists together with its "smoke" tests.

To sum up, we believe that, despite the highly prototypical nature of Broom, the presented experiments show that the proposed approach is indeed capable of handling code that is beyond the capabilities of other currently existing approaches.

9 Conclusion and Future Works

We have presented a new SL-based bi-abduction analysis capable of analysing fragments of code that manipulates with various forms of dynamic linked lists implemented using advanced low-level pointer operations. This includes operations such as pointer arithmetic, bit-masking on pointers, block operations, dealing with blocks of in-advance-unknown size, splitting them into fields of not-fixed size, which can then be merged again, etc. Although our approach builds on a body of previous research, especially, [2, 6, 7, 16], it extends it significantly to handle the mentioned features. In particular, to be able to handle the considered kind of code, we build on a flavor of SL that uses a per-field separating conjunction instead of a per-object separating conjunction, and we also introduce a number of new abduction rules that allow us to deal with pointer arithmetic, block splitting and merging, and so on. We have implemented the proposed approach in a prototype tool Broom. Despite Broom is a very early prototype, our experiments with it allowed us to handle code fragments that are – to the best of our knowledge – out of the capabilities of currently existing analysers.

We believe that there is a lot of space for further improvements of our results in the future. First, we would like to significantly optimize Broom to make it applicable to larger code bases. Here, we are thinking of applying many of the low-level optimisations applied in other tools of a similar kind (replacing as many as possible of SMT queries by answering them using simple static rules, using incremental SMT solving, caching as much information as possible, etc.). Next, we would like to explore possibilities how to reduce the amount of non-determinism present in the abduction when the `alias-weak/strong` rules are applied. The goal is to preserve as much as possible of the power of these rules but reduce the cost of applying them. Perhaps, we could rely here partially on some pre-defined heuristics and partially even on some techniques from machine learning, which are now being applied even in SMT solvers and elsewhere. Next, we would like to significantly improve our implementation of list abstractions (inspired, e.g., by [16]) as well as numerical abstractions. Last but not least, we would also like to think of adding support for other classes of dynamic data structures than lists.

References

1 Andrew W. Appel. *Program Logics - for Certified Compilers*. Cambridge University Press, 2014.

2 J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape Analysis for Composite Data Structures. In *Proc. of CAV'07*, volume 4590 of *LNCS*. Springer, 2007.

3 A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. Accurate Invariant Checking for Programs Manipulating Lists and Arrays with Infinite Data. In *Proc. of ATVA'12*, volume 7561 of *LNCS*. Springer, 2012.

4 C. Calcagno and D. Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *Proc. of NFM'11*, volume 6617 of *LNCS*. Springer, 2011.

5 C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

6 C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. In *Proc. of POPL'09*. ACM, 2009.

7 C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. *Journal of the ACM*, 58(6), 2011.

8 Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.

9 B.-Y.E. Chang, X. Rival, and G.C. Necula. Shape Analysis with Structural Invariant Checkers. In *Proc. of SAS'07*, volume 4634 of *LNCS*. Springer, 2007.

10 W.-N. Chin, C. David, H.H. Nguyen, and S. Qin. Automated Verification of Shape, Size and Bag Properties via User-defined Predicates in Separation Logic. *Science of Computer Programming*, 77(9), 2012.

11 C. Curry, Q. Loc Le, and S. Qin. Bi-Abductive Inference for Shape and Ordering Properties. In *Proc. of ICECCS'19*. IEEE, 2019.

12 L.M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS'08*, volume 4963 of *LNCS*. Springer, 2008.

13 Stéphane Demri, Étienne Lozes, and Alessio Mansutti. The effects of adding reachability predicates in propositional separation logic. In *FoSSaCS*, volume 10803 of *Lecture Notes in Computer Science*, pages 476–493. Springer, 2018.

14 C. Drăgoi, C. Enea, and M. Sighireanu. Local Shape Analysis for Overlaid Data Structures. In *Proc. of SAS'13*, volume 7935 of *LNCS*. Springer, 2013.

15 K. Dudka, P. Peringer, and T. Vojnar. An Easy to Use Infrastructure for Building Static Analysis Tools. In *Proc. of EUROCAST'11*, volume 6927 of *LNCS*. Springer, 2011.

16 K. Dudka, P. Peringer, and T. Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In *Proc. of SAS'13*, volume 7935 of *LNCS*. Springer, 2013.

17 M. Echenim, R. Iosif, and N. Peltier. Unifying Decidable Entailments in Separation Logic with Inductive Definitions. In *Proc. of CADE'21*, volume 12699 of *LNCS*. Springer, 2021.

18 C. Enea, O. Lengál, M. Sighireanu, and T. Vojnar. Compositional Entailment Checking for a Fragment of Separation Logic. In *Proc. of APLAS'14*, volume 8858 of *LNCS*. Springer, 2014.

19 B. Fang and M. Sighireanu. Hierarchical Shape Abstraction for Analysis of Free List Memory Allocators. In *Proc. of LOPSTR'16*, volume 10184 of *LNCS*. Springer, 2016.

20 J. Heinen, T. Noll, and S. Rieger. Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. In *Proc. of TSS'09*, volume 266 of *ENTCS*. Elsevier, 2010.

21 L. Holík, O. Lengál, J. Šimáček, A. Rogalewicz, and T. Vojnar. Fully Automated Shape Analysis Based on Forest Automata. In *Proc. of CAV'13*, volume 8044 of *LNCS*. Springer, 2013.

22 L. Holík, P. Peringer, A. Rogalewicz, V. Šoková, T. Vojnar, and F. Zuleger. Low-Level Bi-Abduction, 2022. `arXiv:2205.02590`.

23 R. Iosif, A. Rogalewicz, and T. Vojnar. Deciding Entailments in Inductive Separation Logic with Tree Automata. In *Proc. of ATVA'14*, volume 8837 of *LNCS*. Springer, 2014.

24 S. Ishtiaq and P.W. O'Hearn. Separation and Information Hiding. In *Proc. of POPL'01*. ACM, 2001.

25 J.L. Jensen, M.E. Jørgensen, M.I. Schwartzbach, and N. Klarlund. Automatic Verification of Pointer Programs Using Monadic Second-order Logic. In *Proc. of PLDI'97*. ACM, 1997.

26 J. Katelaan and F. Zuleger. Beyond Symbolic Heaps: Deciding Separation Logic With Inductive Definitions. In *Proc. of LPAR'11*, volume 73 of *EPiC Series in Computing*. EasyChair, 2020.

27 Q. Loc Le. Compositional Satisfiability Solving in Separation Logic. In *Proc. of VMCAI'21*, volume 12597 of *LNCS*. Springer, 2021.

28 Q. Loc Le, C. Gherghina, S. Qin, and W.-N. Chin. Shape Analysis via Second-Order Bi-Abduction. In *Proc. of CAV'14*, volume 8559 of *LNCS*. Springer, 2014.

29 P. Maksimovic, S.-É. Ayoun, J.F. Santos, and P. Gardner. Gillian, Part II: Real-World Verification for JavaScript and C. In *Proc. of CAV'21*, volume 12760 of *LNCS*. Springer, 2021.

30 P. Maksimovic, J.F. Santos, S.-É. Ayoun, and P. Gardner. Gillian: A Multi-Language Platform for Unified Symbolic Analysis, 2021. `arXiv:2105.14769`.

31 V. Malik, M. Hruška, P. Schrammel, and T. Vojnar. Template-Based Verification of Heap-Manipulating Programs. In *Proc. of FMCAD'18*. IEEE, 2018.

32 Jens Pagel and Florian Zuleger. Strong-separation logic. In *ESOP*, volume 12648 of *Lecture Notes in Computer Science*, pages 664–692. Springer, 2021.

33 J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE, 2002.

34 M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems*, 24(3), 2002.

35 J.F. Santos, P. Maksimovic, S.-É. Ayoun, and P. Gardner. Gillian: Compositional Symbolic Execution for All, 2020. `arXiv:2001.05059`.

36 J.F. Santos, P. Maksimovic, S.-É. Ayoun, and P. Gardner. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In *Proc. of PLDI'20*. ACM, 2020.

37 T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. On Verifying Complex Properties using Symbolic Shape Analysis. In *Proc. of HAV'07*, 2007.

38 H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of CAV'08*, volume 5123 of *LNCS*. Springer, 2008.

39 K. Zee, V. Kuncak, and M.C. Rinard. Full Functional Verification of Linked Data Structures. In *Proc. of PLDI'08*. ACM, 2008.