


Deciding Boolean Separation Logic via Small Models *

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

[1] Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
idacik@fit.vut.cz

[2] Faculty of Informatics, Vienna University of Technology, Vienna, Austria

Abstract. We present a novel decision procedure for a fragment of separation logic (SL) with arbitrary nesting of separating conjunctions with boolean conjunctions, disjunctions, and guarded negations together with a support for the most common variants of linked lists. Our method is based on a model-based translation to SMT for which we introduce several optimisations—the most important of them is based on bounding the size of predicate instantiations within models of larger formulae, which leads to a much more efficient translation of SL formulae to SMT. Through a series of experiments, we show that, on the frequently used symbolic heap fragment, our decision procedure is competitive with other existing approaches, and it can outperform them outside the symbolic heap fragment. Moreover, our decision procedure can also handle some formulae for which no decision procedure has been implemented so far.

1 Introduction

In the last decade, separation logic (SL) [15, 30] has become one of the most popular formalisms for reasoning about programs working with dynamically-allocated memory, including approaches based on deductive verification [32], abstract interpretation [34], symbolic execution [31], or bi-abductive analysis [6, 12, 18]. The key ingredients of SL used in these approaches include the separating conjunction $*$, which allows modular reasoning by stating that the program heap can be decomposed into disjoint parts satisfying operands of the separating conjunction, along with inductive predicates describing shapes of data structures, such as lists, trees, or their various combinations.

The high expressive power of SL comes with the price of high complexity and even undecidability when several of its features are combined together. The existing decision procedures are usually limited to the so-called *symbolic heap fragment* that disallows any boolean structure of spatial assertions.

In this paper, we present a novel decision procedure for a fragment of SL that we call *boolean separation logic* (BSL). The fragment allows arbitrary nesting of separating conjunctions and boolean connectives of conjunction, disjunction, and a limited form of negation of the form $\varphi \wedge \neg\psi$ called *guarded negation*. To the best of our knowledge, no existing, practically applicable decision procedure supports a fragment with such a rich boolean structure and at least basic inductive predicates. The decision procedure for SL in CVC5 [29] supports arbitrary nesting of boolean connectives (including even unguarded negation, which is considered very expensive in the context of SL) but no inductive predicates. A support for conjunctions and disjunctions under separating

* The work was supported by the Czech Science Foundation project GA23-06506S. Basic research funding of the Czech team was provided by the FIT BUT internal project FIT-S-23-8151 and the ERC.CZ project LL1908. Tomáš Dacík was supported by the Brno Ph.D. Talent Scholarship funded by the Brno City Municipality.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14570, pp. 188–206, 2024.
https://doi.org/10.1007/978-3-031-57246-3_11

conjunctions is available in the backend solver of the GRASSHOPPER verifier [27,28] though not described in the papers. In our experimental evaluation, we outperform both of these approaches on some benchmarks (and can decide some formulae beyond the capabilities of both of them). We further show that adding guarded negations to BSL makes its satisfiability problem PSPACE-hard.

To motivate the usefulness of the fragment we consider, we now give several examples when SL formulae with a rich boolean structure are useful. First, in symbolic execution of heap manipulating programs, one usually needs to consider functions that involve some non-determinism—typically, at least the `malloc` statement has the non-deterministic contract $\{\mathsf{emp}\}$ `x = malloc()` $\{x \mapsto f \vee (x = \mathsf{nil} \wedge \mathsf{emp})\}$ (where f is a fresh variable) stating that when the statement is started in the empty heap, once it finishes, x is either allocated, or the allocation had failed and the heap is empty. Such contracts typically need a dedicated (and usually incomplete) treatment when no support of disjunctions is available.[3] Further, the guarded negation semantically represents the set of counterexamples of the entailment $\varphi \models \psi$, and hence allows one to reduce entailment queries to UNSAT checking. Guarded negation can also be used when one needs to obtain several models of a formula φ by joining formulae representing the already obtained models to φ using guarded negations. One can also use the guarded negation to express interesting properties such as the fact that given a list $\mathsf{sls}(x, y)$ and a pointer $y \mapsto z$, the pointer does not point back somewhere into the list closing a lasso. This can be expressed through the formula $\big(\mathsf{sls}(x, y) \wedge \neg\big(\mathsf{sls}(x, z) * \mathsf{sls}(z, y)\big)\big) * y \mapsto z$. Finally, boolean connectives can be introduced by translating quantitative separation logic into the classical SL [2].

In this work, we consider BSL with three fixed, built-in inductive predicates representing the most-common variants of lists: singly-linked (SLL), doubly-linked (DLL), and nested singly-linked (NLL). Our results can be easily extended for their variations such as nested doubly-linked lists of singly-linked lists and the like, but for the price of manually defining their semantics in the SMT encoding. We do, however, believe that our approach of bounding the sizes of models and instantiations of the individual predicates can be lifted to more complex inductive definitions and can serve as a starting point for allowing integration of SL with inductive definitions into SMT.

Contributions. Our approach to deciding BSL formulae is inspired by previous works on translation of SL to SMT. The early works [27] and [28] translate SL to intermediate theories first. Our approach is closer to the more recent approach of [16], which builds on small-model properties and axiomatizes reachability through pointer links directly. We extend the SL fragment considered in [16] by going beyond the so-called unique footprint property (under which it is much easier to obtain an efficient translation). Further, we define a more precise way to obtain global bounds on models of entire formulae, and, most importantly, we modify the translation of inductive predicates in a way that allows us to encode them succinctly by computing local bounds on their instantiations. According to our experiments, this makes the decision procedure efficient and competitive with the state-of-the-art approaches on the symbolic heap fragment (despite the increased decisive power). The claims we make in this paper are proven in [9].

[3] Note that, while the post-condition with a single disjunction might seem simple, the formulae typically start growing in the further symbolic execution.

Related work. In [3], a proof system for deciding entailments of symbolic heaps with lists was proposed. This problem was later shown to be solvable in polynomial time in [8] via graph homomorphism checking. A superposition-based calculus for the fragment was presented in [23], and a model-based approach enhancing SMT solvers was proposed in [24]. In [24], a combination of SL with SMT theories is considered but still limited to the symbolic heap fragment. A more expressive boolean structure and integration with SMT theories was developed in [27] for lists and extended for trees in [28] but still without a support for guarded negations.

Other decision procedures are focusing on more general, *user-defined* inductive predicates (usually of some restricted form). They are based, e.g., on *cyclic proof systems* (CYCLIST [5], S2S [19, 20]); lemma synthesis (SONGBIRD [33]); or automata—tree automata are used in the tools SLIDE [13] and SPEN [11], and a specialised type of automata, called *heap automata*, is used in HARRSH [17]. These procedures do, however, not support nested use of boolean connectives and separating conjunctions.

There also exist works on deciding much more expressive fragments of SL such as [10,14,21,26] but they do not lead to practically implementable decision procedures.

2 Preliminaries

Partial functions. We write $f : X \rightharpoonup Y$ to denote a *partial function* from X to Y. For a partial function f, $\mathsf{dom}(f)$ and $\mathsf{img}(f)$ denote its domain and image, respectively; $|f| = |\mathsf{dom}(f)|$ denotes its size, and $f(x) = \bot$ denotes that f is undefined for x. A restriction $f|_A$ of f to $A \subseteq X$ is defined as $f(x)$ for $x \in A$ and undefined otherwise. To represent a finite partial function f, we often use the set notation $f = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ meaning that f maps each x_i to y_i, and is undefined for other values. We call partial functions f_1 and f_2 *disjoint* if $\mathsf{dom}(f_1) \cap \mathsf{dom}(f_2) = \emptyset$ and define their *disjoint union* $f_1 \uplus f_2$ as $f_1 \cup f_2$, which is otherwise undefined.

Graphs and paths. Let $G = (V, \rightarrow_1, \ldots, \rightarrow_m)$ be a directed graph with vertices V and edges $\rightarrow = \rightarrow_1 \cup \cdots \cup \rightarrow_m$. For $1 \leq \mathsf{f} \leq m$, a sequence $\sigma = \langle v_0, v_1, \ldots, v_n \rangle \in V^+$ is a path from v_0 to v_n via $\rightarrow_\mathsf{f}$ in G, denoted as $\sigma : v_0 \rightsquigarrow_\mathsf{f} v_n$, if all elements of σ are distinct, and for all $0 \leq i < n$, it holds that $v_i \rightarrow_\mathsf{f} v_{i+1}$. By the definition, paths cannot be cyclic. The *domain* of the path σ is the set $\mathsf{dom}(\sigma) = \{v_0, v_1, ..., v_{n-1}\}$, and the length of the path is defined as $|\sigma| = |\mathsf{dom}(\sigma)| = n$.

Formulae. For a first-order formula φ, we denote by $\varphi[t/x]$ the formula obtained by simultaneously replacing all free occurrences of the variable x in φ with the term t. For a first-order model $\mathcal{M}$ and a term t, we write $t^\mathcal{M}$ to denote the evaluation of t in $\mathcal{M}$ defined as usual.

3 Separation Logic

Syntax. Let Vars be a countably infinite set of *sorted variables*. We denote by x^S a variable x of a sort $S \in \mathsf{Sort} = \{\mathbb{S}, \mathbb{D}, \mathbb{N}\}$ representing a location in an SLL, DLL, or NLL, respectively. We omit the sorts when they are not relevant or clear from the context. We further assume that there exists a distinguished, unsorted variable nil. We write $\mathsf{vars}(\varphi)$ to denote the set of all variables in φ plus nil (even when it does not appear in φ). Analogically, $\mathsf{vars}_S(\varphi)$ stands for all variables of the sort S plus nil.

$$
\begin{array}{ll}
(s,h) \models x \bowtie y & \text{iff } s(x) \bowtie s(y) \text{ and } \mathsf{dom}(h) = \emptyset \text{ for } \bowtie \in \{=, \neq\} \\
(s,h) \models x \mapsto \langle \mathsf{f}_i : f_i \rangle_{i \in I} & \text{iff } h = \{s(x) \mapsto \langle \mathsf{f}_i : s(f_i) \rangle_{i \in I}\} \\
(s,h) \models \psi_1 \bowtie \psi_2 & \text{iff } (s,h) \models \psi_1 \bowtie (s,h) \models \psi_2 \text{ for } \bowtie \in \{\wedge, \wedge\neg, \vee\} \\
(s,h) \models \psi_1 * \psi_2 & \text{iff } \exists h_1, h_2.\ h = h_1 \uplus h_2 \neq \bot \text{ and } (s,h_i) \models \psi_i \text{ for } i = 1,2 \\
 & \\
(s,h) \models \exists x.\ \psi & \text{iff there exists } \ell \text{ such that } (s[x \mapsto \ell], h) \models \psi \\
(s,h) \models \mathsf{sls}(x,y) & \text{iff } (s,h) \models x = y, \text{ or } s(x) \neq s(y) \\
 & \quad \text{and } (s,h) \models \exists n.\ x \mapsto n * \mathsf{sls}(n,y) \\
(s,h) \models \mathsf{dls}(x,y,x',y') & \text{iff } (s,h) \models x = y * x' = y', \text{ or } s(x) \neq s(y), s(x') \neq s(y'), \\
 & \quad \text{and } (s,h) \models \exists n.\ x \mapsto \langle \mathsf{n} : n, \mathsf{p} : y' \rangle * \mathsf{dls}(n,y,x',x) \\
(s,h) \models \mathsf{nls}(x,y,z) & \text{iff } (s,h) \models x = y, \text{ or } s(x) \neq s(y) \\
 & \quad \text{and } (s,h) \models \exists n, t.\ x \mapsto \langle \mathsf{n} : n, \mathsf{t} : t \rangle * \mathsf{sls}(n,z) * \mathsf{nls}(t,y,z)
\end{array}
$$

Fig. 1: The semantics of the separation logic. The existential quantifier is used for the definition of the semantics of inductive predicates and it is not a part of our fragment.

The syntax of our fragment is given by the following grammar:

$$
\begin{array}{rll}
p & ::= x^{\mathbb{S}} \mapsto \langle \mathsf{n} : n \rangle \mid x^{\mathbb{D}} \mapsto \langle \mathsf{n} : n, \mathsf{p} : p \rangle \mid x^{\mathbb{N}} \mapsto \langle \mathsf{n} : n, \mathsf{t} : t \rangle & \text{(points-to predicates)} \\
\pi & ::= \mathsf{sls}(x^{\mathbb{S}}, y^{\mathbb{S}}) \mid \mathsf{dls}(x^{\mathbb{D}}, y^{\mathbb{D}}, x_b^{\mathbb{D}}, y_b^{\mathbb{D}}) \mid \mathsf{nls}(x^{\mathbb{N}}, y^{\mathbb{N}}, z^{\mathbb{S}}) & \text{(inductive predicates)} \\
\varphi_A & ::= x = y \mid x \neq y \mid p \mid \pi & \text{(atomic formulae)} \\
\varphi & ::= \varphi_A \mid \varphi * \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \wedge \neg\varphi & \text{(formulae)}
\end{array}
$$

The *points-to* predicate $x \mapsto \langle \mathsf{f}_1 : f_1, \ldots, \mathsf{f}_n : f_n \rangle$ denotes that x is a structure whose fields f_i point to values f_i. We often write $x \mapsto n$ instead of $x \mapsto \langle \mathsf{n} : n \rangle$ and $x \mapsto \_$ if the right-hand side is not relevant. We call x the *root* of the points-to predicate. If π is an inductive predicate $\mathsf{sls}(x,y)$, $\mathsf{dls}(x,y,x',y')$, or $\mathsf{nls}(x,y,z)$, we again call x the root of π, y is the *sink* of π, and we write $\pi(x,y)$ to denote the root and the sink. We define the *sort* of the predicate π, denoted as S_π, as the sort of its root. Then, there is a one-to-one correspondence of predicates and sorts, which we often implicitly use.

Memory model. Let Loc be a countably infinite set of memory locations, and let $\mathsf{Field} = \{\mathsf{n}, \mathsf{p}, \mathsf{t}\}$ be the set of fields. A *stack* is a finite partial function $s : \mathsf{Vars} \rightharpoonup \mathsf{Loc}$. A *heap* is a finite partial function $h : \mathsf{Loc} \rightharpoonup (\mathsf{Field} \rightharpoonup \mathsf{Loc})$. For succinctness, we write $h(\ell, \mathsf{f})$ instead of $h(\ell)(\mathsf{f})$. To represent heap elements in a readable way, we write functions $\mathsf{Field} \rightharpoonup \mathsf{Loc}$ as vectors with labels, i.e., $h(\ell) = \langle \mathsf{f} : h(\ell, \mathsf{f}) \mid \mathsf{f} \in \mathsf{Field} \ \wedge \ h(\ell, \mathsf{f}) \neq \bot \rangle$ and we write $\mathsf{img}(h)$ for $\{\ell \in \mathsf{Loc} \mid \exists \ell', \mathsf{f}.\ h(\ell', \mathsf{f}) = \ell\}$. Moreover, we use $h(\ell) = n$ when $h(\ell) = \langle \mathsf{n} : n \rangle$. A *stack-heap model* is a pair (s,h) where s is stack and h is a heap such that $s(\mathsf{nil}) \neq \bot$ and $h(s(\mathsf{nil})) = \bot$. We define the set of locations of the model (s,h) as $\mathsf{locs}(s,h) = \mathsf{img}(s) \cup \mathsf{dom}(h) \cup \mathsf{img}(h)$.

Semantics. The semantics of our SL over stack-heap models is given in Fig. 1. For pure formulae, we use the so-called *precise semantics*, which additionally requires that the heap must be empty[4]. The semantics of pointer assertions, boolean connectives, and

[4] This is a common approach to avoid the atom true to be expressed as $\mathsf{nil} = \mathsf{nil}$. In our fragment, we forbid true in order not to introduce "unbounded" negations as $\neg\varphi \triangleq \mathsf{true} \wedge \neg\varphi$. Due to this change, symbolic heaps are formulae of form $* \psi_i$ where each ψ_i is an atom.

separating conjunctions is as usual. The intuition behind the semantics of the inductive predicates is as follows. An SLL segment $\mathsf{sls}(x, y)$ is either empty or represents an acyclic sequence of allocated locations starting from x and leading via the n field to y, which is not allocated. A DLL segment $\mathsf{dls}(x, y, x', y')$ is either empty with $x = y$ and $x' = y'$, or it represents an acyclic sequence that is doubly-linked via the n and p fields and leads from the first allocated location x of the segment to its last allocated location x' (x and x' may coincide) with y/y' being the n/p-successors of x'/x, respectively. Both y and y' are not allocated. An NLL segment $\mathsf{nls}(x, y, z)$ is a (possibly empty) acyclic sequence of locations starting from x and leading to y via the t (top) field in which successor of each locations starts a disjoint inner SLL to z via n.

Stack-heap graphs. We frequently identify stack-heap models with their graph representation. A stack-heap model (s, h) defines a graph $G[(s, h)] = (V, (\rightarrow_{\mathsf{f}})_{\mathsf{f} \in \mathsf{Field}})$ where $V = \mathsf{locs}(s, h)$ and $u \rightarrow_{\mathsf{f}} v$ iff $h(u, \mathsf{f}) = v$. We frequently use the fact that if there exists a path $\sigma : x \rightsquigarrow_{\mathsf{f}} y$ in a stack-heap graph, then it is uniquely determined because f-edges are given by a partial function.

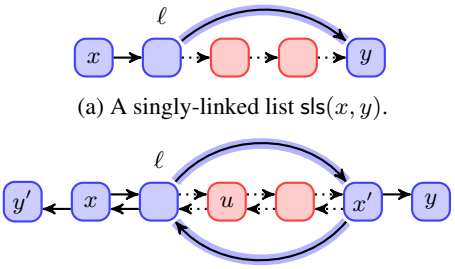
4 Small-Model Property

Small-model properties, which state that each satisfiable formula has a model of bounded size, are frequently used for various fragments of SL to prove their decidability [7] or to design decision procedures [16, 26, 29]. The latter is also the case of our translation-based decision procedure which will heavily rely on enumeration over all locations, and, for its efficiency, it is therefore necessary to obtain location bounds that are as small as possible.

The way we obtain our small-model property is inspired by the approach of [16] and by insights from the so-called *strong-separation logic* [26]. The main idea is to define a satisfiability-preserving reduction $\downarrow^s h$ which takes a heap h (referenced from a stack s), decomposes it into basic sub-heaps (which we call *chunks*), and reduces it per the sub-heaps in such a way that its size can be easily bounded by a linear expression. To define the reduction, we first need to introduce some auxiliary notions related to stack-heap models.

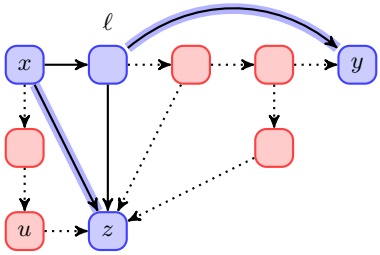
We say that a model (s, h) is *positive* if there exists φ with $(s, h) \models \varphi$. A positive model (s, h) is *atomic* if it is non-empty, and for all positive models (s, h_1) and (s, h_2), $h = h_1 \uplus h_2$ implies that $h_1 = \emptyset$ or $h_2 = \emptyset$. In other words, atomic models cannot be decomposed into two non-empty positive models. Several examples of atomic models are shown in Fig. 2. Observe that the models of dls (Figure 2b) and nls (Figure 2c) are indeed atomic as any of their decomposition, in particular the split at the location u, does not give two positive models.

A sub-heap $c \subseteq h$ is a *chunk* of a model (s, h) if c is a maximal sub-heap of h such that (s, c) is an atomic positive model. Notice that the way the definition of chunks is constructed excludes the possibility of using as a chunk a sub-heap of a heap that itself forms an atomic model. The reason is that otherwise the remaining part of the larger atomic model could not described by the available predicates. For example, in nested lists as shown in Fig. 2c, one cannot take as a chunk a part of some inner list (e.g., the pointer $u \mapsto z$) as the heap shown in the figure itself forms an atomic model. Indeed, if $u \mapsto z$ was removed, one would need a more general version of the NLL predicate to cover the remaining heap by atomic models.



(a) A singly-linked list $\mathsf{sls}(x, y)$.

(b) A doubly-linked list $\mathsf{dls}(x, y, x', y')$.



(c) A nested singly-linked list $\mathsf{nls}(x, y, z)$.

Fig. 2: An illustration of reductions of atomic models of inductive predicates. Removed heap locations are red, removed edges are dotted, and added edges are highlighted.

Lemma 1 (Chunk decomposition). *A positive model (s, h) can be uniquely decomposed into the set of its chunks, denoted $\mathsf{chunks}(s, h)$, i.e., $h = \biguplus \mathsf{chunks}(s, h)$.*

Minimal atomic models of inductive predicates. The key reason why the small-model property that we are going to state holds is that our fragment of SL cannot distinguish atomic models of the considered predicates beyond certain small sizes—namely, two for sls and nls, and three for dls. For further use, we will now state predicates describing exactly the sets of the indistinguishable lists of the different kinds.

We start with SLLs and use a disequality to exclude empty lists: $\mathsf{sls}_{\geq 1}(x, y) \triangleq \mathsf{sls}(x, y) * x \neq y$, and a guarded negation to exclude lists of length one consisting of a single pointer only: $\mathsf{sls}_{\geq 2}(x, y) \triangleq \mathsf{sls}_{\geq 1}(x, y) \wedge \neg(x \mapsto y)$. A similar predicate can be defined for NLLs too: $\mathsf{nls}_{\geq 2}(x, y, z) \triangleq \big(\mathsf{nls}(x, y, z) * x \neq y\big) \wedge \neg(x \mapsto \langle \mathsf{n} : z, \mathsf{t} : y\rangle)$.

For DLLs, we define $\mathsf{dls}_{\geq 2}(x, y, x', y') \triangleq \mathsf{dls}(x, y, x', y') * x \neq y * x \neq x'$ to exclude models that are either empty or consist of a single pointer; and $\mathsf{dls}_{\geq 3}(x, y, x', y') \triangleq \mathsf{dls}_{\geq 2}(x, y, x', y') \wedge \neg(x \mapsto \langle \mathsf{n} : x', \mathsf{p} : y'\rangle * x' \mapsto \langle \mathsf{n} : y, \mathsf{p} : x\rangle)$ to also exclude models consisting of exactly two pointers.

It holds that atomic models, and consequently also chunks, are precisely either models of single pointers or of the above predicates.

Lemma 2. *For atomic model (s, h), exactly one of the following conditions holds.*

1. $(s, h) \models x \mapsto \_$ *for some x.* *(pointer-atom)*
2. $(s, h) \models \mathsf{sls}_{\geq 2}(x, y)$ *for some x and y.* *(sls-atom)*
3. $(s, h) \models \mathsf{dls}_{\geq 3}(x, y, x', y')$ *for some x, y, x', and y'.* *(dls-atom)*
4. $(s, h) \models \mathsf{nls}_{\geq 2}(x, y, z)$ *for some x, y, and z.* *(nls-atom)*

We can now define the reduction in the way we have already sketched.

Definition 1. *The heap of a positive model (s, h) reduces to $\downarrow^s h = \biguplus_{c \in \mathsf{chunks}(s,h)} \downarrow^s c$ where the reduction of a chunk c with a root x as follows:*

- $\downarrow^s c = c$ *if* $(s, c) \models x \mapsto \_$.
- $\downarrow^s c = \{s(x) \mapsto \ell, \ell \mapsto s(y)\}$ *where* $\ell = c(s(x), \mathsf{n})$ *if* $(s, c) \models \mathsf{sls}_{\geq 2}(x, y)$ *for some y.*
- $\downarrow^s c = \{s(x) \mapsto \langle \mathsf{n} : \ell, \mathsf{p} : s(y')\rangle, \ell \mapsto \langle \mathsf{n} : s(x'), \mathsf{p} : s(x)\rangle, s(x') \mapsto \langle \mathsf{n} : s(y), \mathsf{p} : \ell\rangle\}$ *where* $\ell = c(s(x), \mathsf{n})$ *if* $(s, c) \models \mathsf{dls}_{\geq 3}(x, y, x', y')$ *for some x', y' and y.*
- $\downarrow^s c = \{s(x) \mapsto \langle \mathsf{t} : \ell, \mathsf{n} : s(z)\rangle, \ell \mapsto \langle \mathsf{t} : s(y), \mathsf{n} : s(z)\rangle\}$ *where* $\ell = c(s(x), \mathsf{t})$ *if* $(s, c) \models \mathsf{nls}_{\geq 2}(x, y, z)$ *for some y and z.*

We lift the reduction to stack-heap models as $\downarrow^X (s,h) = (s', \downarrow^{s'} h)$ where $s' = s|_X$ for some set of variables X and show that it preserves satisfiability when $X = \mathsf{vars}(\varphi)$.

Theorem 1. *For a positive model (s,h), it holds that $(s,h) \models \varphi$ iff $\downarrow^{\mathsf{vars}(\varphi)} (s,h) \models \varphi$.*

The final step to show our small-model property is to find an upper bound on the size of the reduced models. We define the size of a variable x^S, $||x^S||$, which represents its contribution to the location bound, and is defined as 2 if $S \in \{\mathbb{S}, \mathbb{N}\}$ and 1.5 if $S = \mathbb{D}$ (this corresponds to the size of a reduced chunk of sort S divided by the number of variables which are allocated in it). We further define $||\mathsf{nil}|| = 0$. The location bound of φ is then given as $\mathsf{bound}(\varphi) = 1 + \lfloor \sum_{x \in \mathsf{vars}(\varphi)} ||x|| \rfloor$ (the additional location is for nil). Analogically, the location bound for a sort S is $\mathsf{bound}_S(\varphi) = \lfloor \sum_{x \in \mathsf{vars}_S(\varphi)} ||x|| \rfloor$.

Theorem 2 (Small-model property). *If a formula φ is satisfiable, then there exists a model $(s,h) \models \varphi$ such that $|\mathsf{locs}(s,h)| \leq \mathsf{bound}(\varphi)$.*

We conjecture that the bound can be further improved, e.g., by showing that each model can be transformed to an equivalent one (indistinguishable by BSL formulae) such that the number of its chunks is bounded by the number of roots of spatial predicates in φ. We demonstrate this on the formula $\mathsf{sls}(x,y) * y \mapsto z$ and its model in which y points back into the middle of the list segment (thus splitting it into two chunks). Clearly, this model can be transformed by redirecting z outside of the list domain.

5 Translation-Based Decision Procedure

In this section, we present our translation of SL to SMT. We first present an SMT encoding of our memory model and a translation of basic predicates and boolean connectives. Then we discuss methods for efficient translation of separating conjunctions and inductive predicates with the focus on avoiding quantifiers by replacing them by small enumerations of their instantiations.

We fix an input formula φ and let $n_S = \mathsf{bound}_S(\varphi)$ for each sort $S \in \mathsf{Sort}$.

5.1 Encoding the Memory Model in SMT

To encode the heap, we use a classical approach which encodes its mapping and domain separately [16, 27, 29]. Namely, we use arrays to encode mappings and sets to encode domains. We also use the theory of datatypes to represent a finite sort of locations by a datatype $\mathsf{L} \triangleq \mathsf{loc}^{\mathsf{nil}} \mid \mathsf{loc}^{\mathbb{S}}_1 \mid \ldots \mid \mathsf{loc}^{\mathbb{S}}_{n_\mathbb{S}} \mid \mathsf{loc}^{\mathbb{D}}_1 \mid \ldots \mid \mathsf{loc}^{\mathbb{D}}_{n_\mathbb{D}} \mid \mathsf{loc}^{\mathbb{N}}_1 \mid \ldots \mid \mathsf{loc}^{\mathbb{N}}_{n_\mathbb{N}}$.

Now, we define the signature of the translation's language over the sort L. For each $x \in \mathsf{vars}(\varphi)$, we introduce a constant x of the same name—its interpretation represents the stack image $s(x)$. To represent the heap, we introduce a set symbol D representing the domain and an array symbol h_f for each field $\mathsf{f} \in \mathsf{Field}$ which represents the mapping of the partial function $\lambda \ell.\ h(\ell, \mathsf{f})$. To distinguish sorts of locations, we further introduce a set symbol D_S for each sort $S \in \mathsf{Sort}$. We define meaning of these symbols by showing how a stack-heap model can be reconstructed from a first-order model.

Definition 2 (Inverse translation). *Let $\mathcal{M}$ be a first-order model. We define its inverse translation $\mathsf{T}^{-1}_\varphi(\mathcal{M}) = (s,h)$ where $s(x) = x^\mathcal{M}$ if $x \in \mathsf{vars}(\varphi)$ and*

$$h(\ell) = \begin{cases} \langle \mathsf{n} \colon h_\mathsf{n}[\ell]^\mathcal{M} \rangle & \text{if } \ell \in (D \cap D_\mathbb{S})^\mathcal{M} \\ \langle \mathsf{n} \colon h_\mathsf{n}[\ell]^\mathcal{M}, \mathsf{p} \colon h_\mathsf{p}[\ell]^\mathcal{M} \rangle & \text{if } \ell \in (D \cap D_\mathbb{D})^\mathcal{M} \\ \langle \mathsf{n} \colon h_\mathsf{n}[\ell]^\mathcal{M}, \mathsf{t} \colon h_\mathsf{t}[\ell]^\mathcal{M} \rangle & \text{if } \ell \in (D \cap D_\mathbb{N})^\mathcal{M}. \end{cases}$$

To ensure consistency of the translation with the memory model used, we define the following axioms that a result of translation needs to satisfy:

$$\mathcal{A}_\varphi \triangleq \mathsf{nil} = \mathsf{loc}^{\mathsf{nil}} \wedge\ \mathsf{nil} \notin D\ \wedge \bigwedge_{S \in \mathsf{Sort}} \big(D_S = \{\mathsf{loc}^{\mathsf{nil}}, \mathsf{loc}_1^S, \ldots, \mathsf{loc}_{n_S}^S\}\ \wedge \bigwedge_{x \in \mathsf{vars}_S(\varphi)} x \in D_S\big).$$

The axioms ensure that nil is never allocated, that each variable is interpreted as a location of the corresponding sort and they fix the interpretation of the sets $D_{\mathbb{S}}, D_{\mathbb{D}}, D_{\mathbb{N}}$, which we will later use in the translation to assign sorts to locations.

5.2 Translation of SL to SMT

We define the translation as a function $\mathsf{T}(\varphi) = \mathcal{A}_\varphi \wedge \mathsf{T}(\varphi, D)$ where $\mathcal{A}_\varphi$ are the above defined axioms and $\mathsf{T}(\varphi, D)$ is a recursive translation function of the formula φ with the domain symbol D. The translation $\mathsf{T}(\cdot)$ together with the inverse translation of models $\mathsf{T}_\varphi^{-1}(\cdot)$ are linked by the following correctness theorem.

Theorem 3 (Translation correctness). *An SL formula φ is satisfiable iff its translation $\mathsf{T}(\varphi)$ is satisfiable. Moreover, if $\mathcal{M} \models \mathsf{T}(\varphi)$, then $\mathsf{T}_\varphi^{-1}(\mathcal{M}) \models \varphi$.*

The translation of non-inductive predicates and boolean connectives is defined as:

$$\begin{aligned}
\mathsf{T}(x \bowtie y, F) &\triangleq x \bowtie y\ \wedge\ F = \emptyset && \text{for } \bowtie \in \{=, \neq\}\\
\mathsf{T}(\psi_1 \bowtie \psi_2, F) &\triangleq \mathsf{T}(\psi_1, F) \bowtie \mathsf{T}(\psi_2, F) && \text{for } \bowtie \in \{\wedge, \vee, \wedge\neg\}\\
\mathsf{T}(x \mapsto \langle \mathsf{f}_i : f_i \rangle_{i \in I}, F) &\triangleq F = \{x\}\ \wedge \bigwedge_{i \in I} h_{\mathsf{f}_i}[x] = f_i
\end{aligned}$$

The translation of boolean connectives follows the boolean structure and propagates the domain symbol F to the operands. The translation of pointer assertions postulates content of memory cells represented by arrays and also requires the domain F to be $\{x\}$.

Translation of separating conjunctions. The semantics of separating conjunctions involves a quantification over sets (heap domains). The most direct way of translation is to use quantifiers over sets leading to decidable formulae due to the bounded location domain. This approach combined with a counterexample-guided quantifier instantiation is used in the decision procedure for a fragment of SL supported in CVC5 [29]. In some fragments, however, separating conjunctions can be translated in a way that completely avoids quantifiers. An example is the fragment of boolean combinations of symbolic heaps which has the so-called *unique footprint property* (UFP) [16, 27]—a formula ψ has a (unique) footprint in a model (s, h) with $(s, h) \models \psi * \mathsf{true}$[5], if there exists a (unique) set F such that $(s, h|_F) \models \psi$. The UFP-based approaches of [16, 27] axiomatize the footprints during translation and check operands of separating conjunctions just on the sub-heaps induced by their footprints.

However, UFP does not hold for BSL because of disjunctions. As an example, take the formula $\psi \triangleq x \mapsto y \vee \mathsf{emp}$ and the heap $h = \{x \mapsto y\}$. Both $(s, h|_{\{s(x)\}}) \models \psi$ and $(s, h|_\emptyset) \models \psi$ hold. The sets $\{s(x)\}$ and $\emptyset$ are, however, the only footprints of ψ in (s, h), and this observation can be used to generalise the idea of footprints beyond the fragment in which they are unique.

[5] Assuming the standard semantics of true which is not part of our logic.

Instead of axiomatizing the footprints, our translation builds a set of footprint terms for operands of separating conjunctions. This change can be also seen as a simplification of the former translations as it eliminates the need to deal with two kinds of formulae (the actual translation and footprint axioms), which must be treated differently during the translation. However, the precise computation of the set of all footprints of ψ in (s,h), denoted as $\mathsf{FP}_{(s,h)}(\psi)$, is as hard as satisfiability—when the set of footprints is non-empty, the formula ψ is satisfiable. Therefore, we compute just an over-approximation denoted as $\mathsf{FP}^{\#}(\psi)$. This is justified by the following lemma which gives an equivalent semantics of the separating conjunction in terms of footprints.

Lemma 3. *Let $\varphi \triangleq \psi_1 * \psi_2$ and let (s,h) be a model. Let $\mathcal{F}_1$ and $\mathcal{F}_2$ be sets of locations such that $\mathsf{FP}_{(s,h)}(\psi_i) \subseteq \mathcal{F}_i$. Then $(s,h) \models \psi_1 * \psi_2$ iff*

$$\bigvee_{F_1 \in \mathcal{F}_1} \bigvee_{F_2 \in \mathcal{F}_2} \bigwedge_{i=1,2} (s, h|_{F_i}) \models \psi_i \ \wedge\ F_1 \cap F_2 = \emptyset \ \wedge\ F_1 \cup F_2 = \mathsf{dom}(h).$$

Intuitively, to check whether a separating conjunction holds in a model, it is not necessary to check all possible splits of the heap, but only the splits induced by (possibly over-approximated) footprints of its operands. The lemma is therefore a generalisation of UFP and leads to the following definition of the translation $\mathsf{T}(\psi_1 * \psi_2, F)$:

$$\exists F_1 \in \mathcal{F}_1.\ \exists F_2 \in \mathcal{F}_2.\quad \mathsf{T}(\psi_1, F_1) \wedge \mathsf{T}(\psi_2, F_2) \wedge F_1 \cap F_2 = \emptyset \wedge F = F_1 \cup F_2.$$

Here, we use a quantifier expression of the form $\exists x \in X.\ \psi$ as a placeholder that helps us to define two methods which the translation can use for separating conjunctions:

- The method `SatEnum` computes sets of footprints $\mathcal{F}_i$ as $\mathsf{FP}^{\#}(\psi_i)$ (the computation is described below) and replaces expressions $\exists x \in X.\ \psi$ with $\bigvee_{x' \in X} \psi[x'/x]$ as in Lemma 3. This strategy is quite efficient in many practical cases when we can compute small sets of footprints $\mathcal{F}_1$ and $\mathcal{F}_2$.
- The method `SatQuantif` does not compute sets $\mathcal{F}_i$ at all and replaces $\exists x \in X.\ \psi$ simply with $\exists x.\ \psi$. This strategy is better when the existential quantifier can be later eliminated by Skolemization or when the set of footprints would be too large.

We now show how to compute the set of footprint terms $\mathsf{FP}^{\#}(\psi)$. We again postpone inductive predicates to Section 5.3. We just note that their footprints are unique. The cases of pure formulae and pointer assertions follow directly from the definition of their semantics, which requires the heap to be empty and a single pointer, respectively.

$$\mathsf{FP}^{\#}(x \bowtie y) = \{\emptyset\} \ \text{ for } \bowtie \in \{=, \neq\} \qquad\qquad \mathsf{FP}^{\#}(x \mapsto \_) = \{\{x\}\}$$

For the boolean conjunction, we can select from footprints of its operand the one with the lesser cardinality. Since negations have many footprints (consider, e.g., $\neg\mathsf{emp}$), we define the case of the guarded negation by taking footprints of its guard. The disjunction is the only case which brings non-uniqueness as we need to consider footprints of both of its operands.

$$\mathsf{FP}^{\#}(\psi_1 \wedge \neg\psi_2) = \mathsf{FP}^{\#}(\psi_1) \qquad\qquad \mathsf{FP}^{\#}(\psi_1 \vee \psi_2) = \mathsf{FP}^{\#}(\psi_1) \cup \mathsf{FP}^{\#}(\psi_2)$$

$$\mathsf{FP}^{\#}(\psi_1 \wedge \psi_2) = \text{ if } |\mathsf{FP}^{\#}(\psi_1)| \leq |\mathsf{FP}^{\#}(\psi_2)| \text{ then } \mathsf{FP}^{\#}(\psi_1) \text{ else } \mathsf{FP}^{\#}(\psi_2)$$

Finally, we define footprints of the separating conjunction by taking the union $F_1 \cup F_2$ for each pair (F_1, F_2) of footprints of its operands. Notice that here $F_1 \cup F_2$ represents an SMT term, therefore we cannot replace it with a disjoint union which is not available in the classical set theories in SMT. We can, however, use heuristics and filter out terms for which we can statically determine that interpretations of F_1 and F_2 are not disjoint.

$$\mathsf{FP}^{\#}(\psi_1 * \psi_2) = \{F_1 \cup F_2 \mid F_1 \in \mathsf{FP}^{\#}(\psi_1) \text{ and } F_2 \in \mathsf{FP}^{\#}(\psi_2)\}$$

We state the correctness of the footprint computation in the following lemma.

Lemma 4. *Let $\mathcal{M}$ be a first-order model with $\mathcal{M} \models \mathsf{T}(\varphi)$ and let $(s, h) = \mathsf{T}_{\varphi}^{-1}(\mathcal{M})$. Then we have $\mathsf{FP}_{(s,h)}(\varphi) \subseteq \{F^{\mathcal{M}} \mid F \in \mathsf{FP}^{\#}(\varphi)\}$.*

5.3 Translation of Inductive Predicates

To translate inductive predicates, we express them in terms of reachability and paths in the heaps. While unbounded reachability cannot be expressed in first-order logic, we can efficiently express bounded *linear* reachability in our encoding. The linearity means that each path uses only a single field (which is not the case, e.g., for paths in trees). All predicates in this section are parametrised with an interval $[m, n]$ which bounds the length of the considered paths. When we do not state the bounds explicitly, we assume conservative bounds $[0, \mathsf{bound}_S(\varphi)]$ for a path starting from a root of a sort S. We show how to compute more precise bounds in Section 6. We start with the translation of reachability:

$$\mathsf{reach}^{=n}(h, x, y) \triangleq h^n[x] = y \quad \mathsf{reach}^{[m,n]}(h, x, y) \triangleq \bigvee\nolimits_{m \le i \le n} \mathsf{reach}^{=i}(h, x, y)$$

Here, the predicate $\mathsf{reach}^{=n}(h, x, y)$ expresses that x can reach y via a field represented by the array h in exactly n steps. Similarly, $\mathsf{reach}^{[m,n]}$ expresses reachability in m to n steps. Besides reachability, we will need a macro $\mathsf{path}_C(h, x, y)$ expressing the domain of a path from x to y, or the empty set if such a path does not exists:

$$\begin{aligned}
\mathsf{path}_C^{=n}(h, x, y) &\triangleq \bigcup\nolimits_{0 \le i < n} C(h^i[x]) \\
\mathsf{path}_C^{[m,n]}(h, x, y) &\triangleq \mathsf{if}\ (\mathsf{reach}^{=m}(h, x, y))\ \mathsf{then}\ (\mathsf{path}_C^{=m}(h, x, y)) \\
&\quad \cdots\ \mathsf{else\ if}\ (\mathsf{reach}^{=n}(h, x, y))\ \mathsf{then}\ (\mathsf{path}_C^{=n}(h, x, y))\ \mathsf{else}\ (\emptyset)
\end{aligned}$$

The additional parameter C is a function applied to each element of the path that can be used to define nested paths. We define a simple path $\mathsf{path}_S^{[m,n]}(h, x, y) \triangleq \mathsf{path}_C^{[m,n]}(h, x, y)$ with $C \triangleq \lambda \ell.\ \{\ell\}$ and a nested path as $\mathsf{path}_N^{[m,n]}(h_1, h_2, x, y, z) \triangleq \mathsf{path}_C^{[m,n]}(h_1, x, y)$ with $C \triangleq \lambda \ell.\ \mathsf{path}_S(h_2, \ell, z)$. In the case of the nested path, the array h_1 represents the top-level path from x to y, and h_2 represents nested paths terminating in the common location z. Now we can define footprints of inductive predicates using path terms as follows:

$$\begin{aligned}
\mathsf{FP}^{\#}(\pi(x, y)) &= \{\mathsf{path}_S(h_{\mathsf{n}}, x, y)\} && \text{for } \pi \in \{\mathsf{sls}, \mathsf{dls}\} \\
\mathsf{FP}^{\#}(\mathsf{nls}(x, y, z)) &= \{\mathsf{path}_N(h_{\mathsf{t}}, h_{\mathsf{n}}, x, y, z)\}
\end{aligned}$$

The common part of the translation $\mathsf{T}(\pi(x,y),F)$ postulates the existence of a top-level path from x to y and a domain F based on this path (formalised in the formula main_path below); and ensures that all locations have the correct sort (through the formula typing). For DLLs, we add an invariant which ensures that its locations are correctly doubly-linked (the back_links formula), and we further need a special treatment of the cases when the list is empty as well as a special treatment for its roots and sinks (cf. the formula boundaries). For NLLs, we add an invariant stating that an inner list starts from each location in its top-level path (the inner_lists formula) and that those inner paths are disjoint (the disjoint formula)[6].

- $\mathsf{T}(\mathsf{sls}(x,y),F) \triangleq \mathsf{main\_path} \wedge \mathsf{typing}$ where

  $\mathsf{main\_path} \triangleq \mathsf{reach}(h_\mathsf{n},x,y) \wedge F = \mathsf{path}_S(h_\mathsf{n},x,y)$ and $\mathsf{typing} \triangleq F \subseteq D_\mathbb{S}$.

- $\mathsf{T}(\mathsf{dls}(x,y,x',y'),F) \triangleq \mathsf{empty} \vee \mathsf{nonempty}$ where

  $\mathsf{empty} \triangleq x = y \wedge x' = y' \wedge F = \emptyset$,

  $\mathsf{nonempty} \triangleq x \neq y \wedge x' \neq y' \wedge \mathsf{main\_path} \wedge \mathsf{boundaries} \wedge \mathsf{typing} \wedge \mathsf{back\_links}$,

  $\mathsf{main\_path} \triangleq \mathsf{reach}(h_\mathsf{n},x,y) \wedge F = \mathsf{path}_S(h_\mathsf{n},x,y)$,

  $\mathsf{boundaries} \triangleq h_\mathsf{p}[x] = y' \wedge h_\mathsf{n}[x'] = y \wedge x' \in F \wedge y' \notin F$,

  $\mathsf{typing} \triangleq F \subseteq D_\mathbb{D}$,

  $\mathsf{back\_links} \triangleq \forall \ell.\ (\ell \in F \wedge \ell \neq x') \rightarrow h_\mathsf{p}[h_\mathsf{n}[\ell]] = \ell$.

- $\mathsf{T}(\mathsf{nls}(x,y,z),F) \triangleq \mathsf{main\_path} \wedge \mathsf{typing} \wedge \mathsf{inner\_lists} \wedge \mathsf{disjoint}$ where

  $\mathsf{main\_path} \triangleq \mathsf{reach}(h_\mathsf{t},x,y) \wedge F = \mathsf{path}_N(h_\mathsf{t},h_\mathsf{n},x,y,z)$,

  $\mathsf{typing} \triangleq \mathsf{path}_S(h_\mathsf{t},x,y) \subseteq D_\mathbb{N} \wedge F \setminus \mathsf{path}_S(h_\mathsf{t},x,y) \subseteq D_\mathbb{S}$,

  $\mathsf{inner\_lists} \triangleq \forall \ell.\ \ell \in F \cap D_\mathbb{N} \rightarrow \mathsf{reach}(h_\mathsf{n},h[\ell],z)$,

  $\mathsf{disjoint} \triangleq \forall \ell_1,\ell_2.\big(\{\ell_1,\ell_2\} \subseteq F \wedge \ell_1 \neq \ell_2 \wedge h_\mathsf{n}[\ell_1] = h_\mathsf{n}[\ell_2]\big) \rightarrow h_\mathsf{n}[\ell_1] \notin F$.

Path quantifiers. Invariants of paths are naturally expressed using universal quantifiers. For quantifiers, however, we cannot directly take advantage of bounds on path lengths. Therefore, similarly as for separating conjunctions, we use the idea of replacing quantifiers by small enumerations of their instances, which is efficient when we can compute small enough bounds on the paths. For example, if we know that the length of an f-path with a root x is at most two, it is enough to instantiate its invariant for x, $h_\mathsf{f}[x]$, and $h_\mathsf{f}^2[x]$. This idea is formalised using expressions $\mathbb{P}^{\leq n}_{(h,x)}\ \ell.\ \psi$, which we call *path quantifiers* and which state that ψ holds for all locations of the path with the length n starting from x via the array h:

$$\mathbb{P}^{\leq n}_{(h,x)}\ \ell.\ \psi \ \triangleq\ \bigwedge\nolimits_{0 \leq i \leq n}\ \psi[h^i[x]/\ell].$$

If we need to quantify over nested paths, we need to use two path quantifiers (one for the top-level path and one for the nested paths). The quantifiers in the last conjunct of the NLL translation can be rewritten as $\mathbb{P}_{(h_t,x)}\ \ell'_1.\ \mathbb{P}_{(h_t,x)}\ \ell'_2.\ \mathbb{P}_{(h_n,\ell'_1)}\ \ell_1.\ \mathbb{P}_{(h_n,\ell'_2)}\ \ell_2$. In this expression, ℓ'_1 and ℓ'_2 range over locations in the top-level list, and ℓ_1 and ℓ_2 range over locations in the nested paths starting from ℓ'_1 and ℓ'_2, respectively.

[6] In the consequent of the disjoint formula, we could also write $h_\mathsf{n}[\ell_1] = z$ instead of $h_\mathsf{n}[\ell_1] \notin F$, but the latter leads to better performance of SMT solvers.

5.4 Complexity

This section briefly discusses the complexity of the proposed decision procedure as well as the complexity lower bound for the satisfiability problem in the considered fragment of SL. We will use $\mathsf{SAT}(\omega_1, \ldots, \omega_n)$ to denote the satisfiability problem for a sub-fragment constructed of atomic formulae and the connectives ω_i and $\mathsf{SAT}(\overline{\omega_1, \ldots, \omega_n})$ to denote the fragment where none of the connectives ω_i appear.

Theorem 4. *The procedure* `SatQuantif` *produces formula of polynomial size, and, for* $\mathsf{SAT}(\overline{\wedge\neg})$*, it runs in* NP*. The procedure* `SatEnum` *runs in* NP *for* $\mathsf{SAT}(\overline{\vee})$*.*

Proof (sketch). When not considering the instantiation of quantifiers over footprints, both `SatQuantif` and `SatEnum` produce a formula $\mathsf{T}(\varphi)$ of a polynomial size dominated by the translation of inductive predicates. For the variant of the translation of inductive predicates using universal quantifiers over locations, the size is $\mathcal{O}(n^3)$ for SLLs and DLLs (dominated by the $\mathcal{O}(n^3)$ size of the path_S term), and $\mathcal{O}(n^5)$ for NLLs (dominated by path_N). If the input formula does not contain guarded negations, then all quantifiers can be eliminated using Skolemization. The translated formulae are then in a theory decidable in NP (e.g., when sets are encoded as extended arrays [22]).

The procedure `SatEnum` can produce exponentially large formulae because of the footprint enumeration. This can be prevented if the input formula does not contain disjunctions, in which case the footprints of all sub-formulae are unique, i.e., singleton sets. The translated formulae are then again in a theory decidable in NP. □

Theorem 5. $\mathsf{SAT}(\mapsto, \wedge\neg, \wedge, \vee, *)$ *is* PSPACE*-complete.*

Proof (sketch). Membership in PSPACE was proved in [26] for a more expressive fragment. For the hardness part, we build on the reduction from QBF used in [7]. In this reduction, the boolean value of a variable is represented by the corresponding SL variable being allocated (always pointing to nil for simplicity). The fact that x is false is expressed using a negative points-to predicate stating that x is not allocated. The existential quantifier is expressed using the separating conjunction, and the universal quantifier is obtained using the (unguarded) negation. (For details, see [7].)

We show that this reduction can be done without the unguarded negation and the negative points-to assertion, using the guarded negation instead. The key observation is that, for a QBF formula with variables X, we can express that all variables in X can have arbitrary boolean values as $\mathsf{arbitrary}[X] \triangleq *_{x \in X}(x \mapsto \mathsf{nil} \vee \mathsf{emp})$. In the context of variables X, we can then express negation as $\neg F \triangleq \mathsf{arbitrary}[X] \wedge \neg F$ and the truth values of a variable x as $\neg x \triangleq \mathsf{arbitrary}[X \setminus \{x\}]$ and $x \triangleq \mathsf{arbitrary}[X] * x \mapsto \mathsf{nil}$. The rest of the reduction then easily follows [7]. □

6 Optimised Bound Computation

In many practical cases, the main source of complexity is the translation of inductive predicates, which heavily depends on the possible lengths of paths between locations. We now propose how to bound the length of these paths based on the so-called *SL-graphs* which are graph representations of constraints imposed by SL formulae. SL-graphs were originally used for representation and deciding of symbolic heaps with lists in [8]. Here, we use their generalised form which captures must-relations holding in all models of a given formula. Note that the nodes of the graphs are implicitly given by the domains of the involved relations, which themselves can be viewed as edges.

Definition 3. *An SL-graph of φ is a tuple $G[\varphi] = (\circledminus, \circledcirc, (\hookrightarrow_{\mathsf{f}}, \rightsquigarrow_{\mathsf{f}}, \circledast_{\mathsf{f}})_{\mathsf{f} \in \mathsf{Field}})$ where:*

- $\circledminus \subseteq \mathsf{vars}(\varphi) \times \mathsf{vars}(\varphi)$ *is an equivalence relation called must-equality,*
- $\circledcirc \subseteq \mathsf{vars}(\varphi) \times \mathsf{vars}(\varphi)$ *is a symmetric relation called must-disequality,*
- $\hookrightarrow_{\mathsf{f}} \subseteq \mathsf{vars}(\varphi) \times \mathsf{vars}(\varphi)$ *is a must-*f*-pointer relation,*
- $\rightsquigarrow_{\mathsf{f}} \subseteq \mathsf{vars}(\varphi) \times \mathsf{vars}(\varphi)$ *is an irreflexive must-*f*-path relation,*
- $\circledast_{\mathsf{f}} \subseteq \mathsf{vars}(\varphi)^2 \times \mathsf{vars}(\varphi)^2$ *is a symmetric relation called must-*f*-path-disjointness.*

Except $\circledast_{\mathsf{f}}$, the components of $G[\varphi]$ represent atomic formulae—equalities, disequalities, pointers, and paths (i.e., list segments)—holding within all models of φ. The fact that $(x_1, y_1) \circledast_{\mathsf{f}} (x_2, y_2)$ states that, in all models of φ, the domains of f-paths from x_1 to y_1 and from x_2 to y_2 are disjoint.

To compute the SL-graph $G[\varphi]$, we define some auxiliary notation. We define $G_\emptyset$ to be an SL-graph where all the relations are empty. We write $G \lhd \{x_i \bowtie_i y_i\}_{i \in I}$ to denote the SL-graph G' which is the same as G with the elements $x_i \bowtie_i y_i$ for $i \in I$ added to the corresponding relations. We use $\sqcup$ and $\sqcap$ as a component-wise union and intersection of SL-graphs, respectively. We define the disjoint union of SL-graphs as:

$$\begin{aligned} G_1 \uplus G_2 = (G_1 \sqcup G_2) \\ \lhd \{x \circledcirc y \mid x \in \mathsf{alloc}(G_1), y \in \mathsf{alloc}(G_2), \text{ and } (x \text{ is not } \mathsf{nil} \text{ or } y \text{ is not } \mathsf{nil})\} \\ \lhd \{e_1 \circledast_{\mathsf{f}} e_2 \mid \mathsf{f} \in \mathsf{Field}, e_1 \in \mathsf{paths}_{\mathsf{f}}(G_1), \text{ and } e_2 \in \mathsf{paths}_{\mathsf{f}}(G_2)\}. \end{aligned}$$

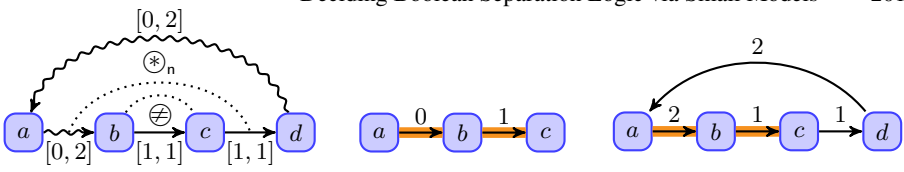
Here, $\mathsf{paths}_{\mathsf{f}}(G)$ is defined as $\hookrightarrow_{\mathsf{f}} \cup \rightsquigarrow_{\mathsf{f}}$, and the set of must-allocated variables is $\mathsf{alloc}(G) = \{x \mid \exists y, \mathsf{f}.\ x \hookrightarrow_{\mathsf{f}} y \text{ or } (x \rightsquigarrow_{\mathsf{f}} y \text{ and } x \circledcirc y)\} \cup \{\mathsf{nil}\}$ (nil is added for technical reasons). We further assume that all operations on SL-graphs ($\lhd$, $\sqcup$, $\sqcap$, and $\uplus$) preserve relational properties (symmetry, transitivity, etc.) of the components of SL-graphs by computing the corresponding closures after the operation is performed. We compute the SL-graph $G[\varphi]$ as follows.

$$\begin{array}{rl} G[x = y] = G_\emptyset \lhd \{x \circledminus y\} & G[x \mapsto \langle \mathsf{f}_i : f_i \rangle_{i \in I}] = G_\emptyset \lhd \{x \hookrightarrow_{\mathsf{f}_i} f_i\}_{i \in I} \\ G[x \neq y] = G_\emptyset \lhd \{x \circledcirc y\} & G[\mathsf{sls}(x, y)] = G_\emptyset \lhd \{x \rightsquigarrow_{\mathsf{n}} y\} \\ G[\psi_1 \wedge \neg\psi_2] = G[\psi_1] & G[\mathsf{dls}(x, y, x', y')] = G_\emptyset \lhd \{x \rightsquigarrow_{\mathsf{n}} y, x' \rightsquigarrow_{\mathsf{p}} y'\} \\ G[\psi_1 \wedge \psi_2] = G[\psi_1] \sqcup G[\psi_2] & G[\mathsf{nls}(x, y, z)] = G_\emptyset \lhd \{x \rightsquigarrow_{\mathsf{n}} z, x \rightsquigarrow_{\mathsf{t}} y\} \\ G[\psi_1 \vee \psi_2] = G[\psi_1] \sqcap G[\psi_2] & G[\psi_1 * \psi_2] = G[\psi_1] \uplus G[\psi_2] \end{array}$$

Observe that we only approximate dls and nls. After the construction is finished, we apply the following rules for matching of pointers and for detection of inconsistencies.

$$\frac{x_1 \hookrightarrow_{\mathsf{f}} y_1 \quad x_2 \hookrightarrow_{\mathsf{f}} y_2 \quad x_1 \circledminus x_2}{y_1 \circledminus y_2}\ (\mapsto\text{-match}) \qquad \frac{x \circledminus y \quad x \circledcirc y}{\varphi \text{ is unsat}}\ (\text{contradiction})$$

Tighter location bounds. Using SL-graphs, we can slightly improve the location bound from Section 4 by considering equivalence classes of $\circledminus$ instead of individual variables (this can be also used to refine the later described path bound computation) and by defining $||x|| = 1$ if x is a must-pointers, i.e., $x \hookrightarrow_{\mathsf{f}} y$ for some f and y.



(a) Fragment of SL-graph $G[\varphi]$. (b) Graph G^{ℓ}_{σ}. (c) Graph G^{u}_{σ}.

Fig. 3: An illustration of the bound computation for the path σ from a to c on a fragment of SL-graph of $\varphi \triangleq \big(\mathsf{sls}(a,b) * b \mapsto c * c \mapsto d * \mathsf{sls}(d,a)\big) \wedge \neg\big(\mathsf{sls}(a,c) * \mathsf{sls}(c,a)\big)$. The highlighted edges denote the paths used to determine the bound $[1,3]$.

Path bounds. We now fix an f-path σ from x^S to y and show how to compute an interval $[\ell, u]$ that gives bounds on its length. The computation of the path bounds runs in two steps. In the first step, we compute an initial bound $[\ell^0_e, u^0_e]$ for each edge $e \in \mathsf{paths}_{\mathsf{f}}(G)$. If e is a pointer edge, its bound is given as $[1,1]$. For a path edge $e = (a,b)$, we define $\ell^0_e = 1$ if $a \not\circledast b$ and 0 otherwise; while u^0_e is defined as $\mathsf{bound}_S(\varphi) - \sum_{v \in V} ||v||$ where $V = \{v \in \mathsf{vars}_S(\varphi) \mid v \text{ is not } x \text{ and } \exists u.\ (v,u) \circledast_{\mathsf{f}} (x,y)\}$. This way, we exclude from the computation of the initial upper bound the source v of each path disjoint with σ and all locations possibly allocated in a chunk with the root v. Note that it can be the case that the actual size of this chunk has a lesser size than $||v||$, but this means that we were too conservative when computing the global location bound and can decrease the path bound by the same number anyway.

In the second phase, we compute the bounds of the path σ using initial bounds from the first step. The computation is based on two weighted directed graphs derived from the SL-graph G: G^{u}_{σ} for the upper bound and G^{ℓ}_{σ} for the lower bound (in both cases, the vertices are implicitly given as $\mathsf{vars}(\varphi)$, and the edge weight of an edge e is given by u^0_e and ℓ^0_e computed in the previous step, respectively):

$$
\begin{aligned}
G^{\mathrm{u}}_{\sigma} &= \{a \to b \mid (a,b) \in \mathsf{paths}_{\mathsf{f}}(G)\},\\
G^{\ell}_{\sigma} &= \{a \to b \mid (a \hookrightarrow_{\mathsf{f}} b \text{ and } a \not\circledast y) \text{ or}\\
&\qquad\qquad (a \rightsquigarrow_{\mathsf{f}} b \text{ and } \exists w.\ \mathsf{nonempty}(y,w) \text{ and } (y,w) \circledast_{\mathsf{f}} (a,b)\}.
\end{aligned}
$$

Here, the condition $\mathsf{nonempty}(y,w)$ states that a directed SL-graph edge (y,w) is non-empty which holds if either $y \hookrightarrow_{\mathsf{f}} w$, or when $y \rightsquigarrow_{\mathsf{f}} w$ and $y \not\circledast w$.

Intuitively, the upper bound u is computed as the length of the shortest path from x to y in G^{u}_{σ}. Since f-paths are uniquely determined, we know that no path can be longer than the shortest one, and thus u is indeed a correct upper bound. The lower bound ℓ is computed as the length of the longest path starting from x (ending anywhere) in G^{ℓ}_{σ}. By construction, G^{ℓ}_{σ} contains only those edges for which one can prove that they cannot contain y in their domains. A path from x of a length ℓ therefore implies that x cannot reach y in less than ℓ steps, and thus ℓ is indeed a correct lower bound.

Example. We demonstrate the path bound computation in Fig. 3, which shows a fragment of the SL-graph of a formula φ (it shows only those $\circledast_{\mathsf{n}}$ edges that are relevant in our example) and the graphs G^{ℓ}_{σ} and G^{u}_{σ} for the path σ from a to c. We have that $||b|| = ||c|| = 1$ and $||a|| = ||d|| = 2$. This gives us the location bound, which is 6. In the first phase, we compute the initial bound $[0,2]$ for paths of the predicates $\mathsf{sls}(a,b)$ and $\mathsf{sls}(d,a)$ because both of them are disjoint with all the other paths in $G[\varphi]$. In the second phase, we get the bound for σ equal to $[1,3]$ instead of the default bound $[0,6]$.

7 Experimental Evaluation

We have implemented the proposed decision procedure in a new solver called ASTRAL[7]. ASTRAL is written in OCaml and can use multiple backend SMT solvers. With the encoding presented in Section 5, it can use either CVC5 supporting set theory directly [1] or Z3 supporting it by a reduction to the extended theory of arrays [22]. We have also developed an alternative encoding in which both locations and location sets are represented as bitvectors. The bitvector encoding differs only in expressing set operations on the level of bitvectors with additional axioms ensuring that all locations "can fit" into sets encoded by the bitvectors (for details, see [9]). With the bitvector encoding, a backend solver only needs to support theories of bitvectors and arrays, which are both standard and supported by many other SMT solvers. Another advantage is that the quantification on bitvectors seems to perform significantly better than on sets.

In our experiments, if we do not say explicitly which encoding and solver is used, we use the bitvector encoding and BITWUZLA [25] as the backend solver, which we found to be the best performing combination. We set a limit for the method `SatEnum` to 64 footprints. If this limited is exceeded, we dynamically switch to `SatQuantif`. We use path quantifiers when the path bound is at most half of the domain bound. These are design choices that can be revisited in the future.

All experiments were run on a machine with 2.5 GHz Intel Core i5-7300HQ CPU and 16 GiB RAM, running Ubuntu 18.04. The timeout was set to 60 s and the memory limit to 1 GB. Our experiments were conducted using BENCHEXEC [4], a framework for reliable benchmarking.

7.1 Entailments of Symbolic Heaps

In the first part of our evaluation, we focus on formulae from the symbolic heap fragment which is frequently used by verification tools and for which there exist many dedicated solvers. We therefore do not expect to outperform the best existing tools but rather to obtain a comparison with other translation-based decision procedures.

In Table 1a, we provide results for the category QF_SHLID_ENTL (entailments with SLLs). We divide the category into two subsets: verification conditions (which are simpler) and more complex artificially generated formulae "*bolognesa*" and "*clones*" from [23]. During the experiments, we found out that several "cloned" entailments contain root variables on the right-hand side of the entailment that do not appear on the left-hand side, making the entailment trivially invalid when its left-hand side is satisfiable. For a few hard clone instances, this makes a problem for ASTRAL as it cannot use the path bound computation as such roots do not appear in the SL-graph. We have therefore implemented a heuristic that detects entailments $\varphi \models \psi$ that can be reduced to satisfiability of φ. Since this is a benchmark-specific heuristic, we present also the version without this heuristic (ASTRAL*) in Table 1a. The optimised version of ASTRAL is able to solve all the formulae being faster than other translation-based solvers GRASSHOPPER[8] and SLOTH. For illustration, the table further contains the second best solver in the latest edition of SL-COMP, S2S[9].

[7] https://github.com/TDacik/Astral

[8] Since GRASSHOPPER is not an solver but a verification tool, we encode the entailment checking as a verification of an empty program.

[9] We had technical issues running the winner ASTERIX [24]. The difference between those tools is, however, negligible.

Table 1: Experimental results for formulae from SL-COMP. The columns are: solved instances (OK), out of time/memory (RO), instances on which ASTRAL wins—ASTRAL can solve it and the other solver not or ASTRAL solves it faster (WIN), instances solved in the time limits of 0.1 s and 1 s, and the total time for solved instances in seconds.

(a) Results for the category QF_SHLS_ENTL.

	Verification conditions (86)						bolognesa+clones (210)					
Solver	OK	RO	WIN	<0.1	≤1	Total time	OK	RO	WIN	<0.1	≤1	Total time
ASTRAL	86	0	-	84	86	4.62	210	0	-	68	169	202.91
ASTRAL*	86	0	42	83	86	4.64	195	15	88	64	150	408.48
GRASSHOPPER	86	0	70	52	86	8.65	203	7	148	60	87	1229.35
S2S	86	0	5	86	86	2.08	210	0	3	203	210	8.18
SLOTH	64	3	86	0	28	235.28	70	140	210	0	50	149.42

(b) Results for a subset of the category QF_SHLID_ENTL.

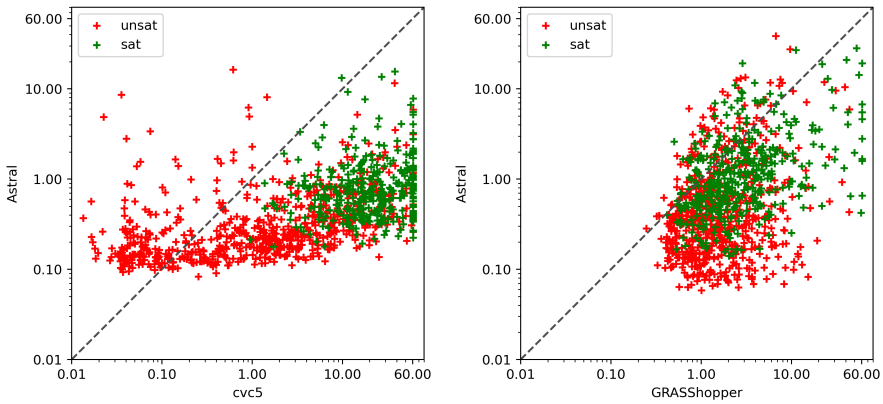
	Doubly-linked lists (17)						Nested singly-linked lists (19)					
Solver	OK	RO	WIN	<0.1	≤1	Total time	OK	RO	WIN	<0.1	≤1	Total time
ASTRAL	17	0	-	11	17	2.72	19	0	-	3	9	86.93
GRASSHOPPER	17	0	16	3	15	7.53	-	-	-	-	-	-
HARRSH	17	0	17	0	0	95.18	14	5	18	0	0	183.01
S2S	17	0	0	17	17	0.15	19	0	0	19	19	0.43
SONGBIRD	11	5	14	5	9	13.39	11	5	8	4	11	1.38

In Table 1b, we provide results for a subset of the category QF_SHLID_ENTL (entailments with linear inductive definitions from which we selected DLLs and NLLs) for ASTRAL and three best-performing solvers competing in the latest edition of SL-COMP—S2S, SONGBIRD (in the version with automated lemma synthesis called SLS), and HARRSH. We also include GRASSHOPPER which supports DLLs only. Except S2S which solves almost all formulae virtually immediately, ASTRAL is the only one able to solve all the formulae in the given time limit.

7.2 Experiments on Formulae Outside of the Symbolic Heap Fragment

For formulae outside of the symbolic heap fragment and its top-level boolean closure, there are currently no existing benchmarks. For now, we therefore limit ourselves to randomly generated but extensive sets of formulae. In the future, we would like to develop a program analyser using symbolic execution over BSL and make more careful experiments on realistic formulae.

We first focus on the fragment with guarded negations but without inductive predicates, on which we can compare ASTRAL with CVC5. We have prepared a set of 1000 entailments of the form $\varphi \models \psi$ which are generated as random binary trees with depth 8 over 8 variables with the only atoms being pointer assertions. To reduce the number of trivial instances, we only generated formulae for which $\mathsf{vars}(\psi) \subseteq \mathsf{vars}(\varphi)$ and ASTRAL cannot deduce contradiction from their SL-graphs. To avoid any suspicion that the difference is caused by better performance of the backend solver rather than the design of our translation, we used ASTRAL with the CVC5 backend and direct set



(a) Comparison with CVC5 (b) Comparison with GRASSHOPPER

Fig. 4: A comparison of ASTRAL with CVC5 and GRASSHOPPER on randomly generated formulae. Times are in seconds, axes are logarithmic. The timeout was set to 60 s.

encoding (with BITWUZLA and bitvector encoding, our results would be even better). The results are given in Fig. 4a and suggest that our treatment of guarded negations really brings a better performance—ASTRAL can solve all the instances and almost all of them under 10 seconds. On the other hand, CVC5 timed out in 61 cases and is usually slower than ASTRAL, in particular on satisfiable formulae which represent invalid entailments.

In the second experiment, we compared our solver with GRASSHOPPER on the fragment which it supports, i.e., arbitrary nesting of conjunctions and disjunctions. We again generated 1000 entailments, this time with depth 6, 6 variables and with atoms being singly-linked lists (with 20 % probability) or pointer-assertions. The results are given in Fig. 4b. ASTRAL ran out of memory in 5 cases, and GRASSHOPPER timed out in 10 cases. In summary, ASTRAL is faster on more than 80 % of the formulae with an almost 3 times lesser running time.

Finally, to illustrate that ASTRAL can indeed handle formulae out of the fragments of all the other mentioned tools, we apply it on an entailment query that involves the formula mentioned at the end of the introduction: $((\mathsf{sls}(x,y) \wedge \neg(\mathsf{sls}(x,z) * \mathsf{sls}(z,y))) * y \mapsto z) \models \mathsf{sls}(x,z)$, converted to an unsatisfiability query. ASTRAL resolves the query in 0.12 s. Note that without the requirement $\neg(\mathsf{sls}(x,z) * \mathsf{sls}(z,y))$, the entailment does not hold as a cycle may be closed in the heap.

8 Conclusions and Future Work

We have presented a novel decision procedure based on a small-model property and translation to SMT. Our experiments have shown very promising results, especially for formulae with rich boolean structure for which our decision procedure outperforms other approaches (apart from being able to solve more formulae).

In the future, we would like to extend our approach with some class of user-defined inductive predicates, with more complex spatial connectives such as septractions and/or magic wands, consider a lazy and/or interactive translation instead of the current eager approach, and try ASTRAL within some SL-based program analyser.

References

1. Bansal, K., Barrett, C., Reynolds, A., Tinelli, C.: A New Decision Procedure for Finite Sets and Cardinality Constraints in SMT. In: IJCAR (2017)
2. Batz, K., Fesefeldt, I., Jansen, M., Katoen, J.P., Keßler, F., Matheja, C., Noll, T.: Foundations for Entailment Checking in Quantitative Separation Logic. In: ESOP (2022)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: A Decidable Fragment of Separation Logic. In: FSTTCS 2004. LNCS, vol. 3328 (2004)
4. Beyer, D., Löwe, S., Wendler, P.: Reliable Benchmarking: Requirements and Solutions. International Journal on Software Tools for Technology Transfer **21** (2017)
5. Brotherston, J., Gorogiannis, N., Petersen, R.L.: A Generic Cyclic Theorem Prover. In: APLAS. LNCS, vol. 7705 (2012)
6. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. Journal of the ACM **58**(6) (2011)
7. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In: FST TCS (2001)
8. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable Reasoning in a Fragment of Separation Logic. In: CONCUR. LNCS, vol. 3901 (2011)
9. Dacík, T., Rogalewicz, A., Vojnar, T., Zuleger, F.: Deciding Boolean Separation Logic via Small Models. Tech. rep. (10 2023), https://zenodo.org/records/10012893
10. Echenim, M., Iosif, R., Peltier, N.: The Bernays-Schönfinkel-Ramsey Class of Separation Logic with Uninterpreted Predicates. ACM Transactions on Computational Logic **21** (2019)
11. Enea, C., Lengál, O., Sighireanu, M., Vojnar, T.: Compositional Entailment Checking for a Fragment of Separation Logic. In: APLAS (2014)
12. Holík, L., Peringer, P., Rogalewicz, A., Šoková, V., Vojnar, T., Zuleger, F.: Low-level bi-abduction. In: ECOOP 2022. LIPIcs, vol. 222, pp. 19:1–19:30 (2022)
13. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding Entailments in Inductive Separation Logic with Tree Automata. In: ATVA (2014)
14. Iosif, R., Zuleger, F.: Expressiveness results for an inductive logic of separated relations. In: Pérez, G.A., Raskin, J. (eds.) CONCUR. LIPIcs, vol. 279, pp. 20:1–20:20 (2023). https://doi.org/10.4230/LIPICS.CONCUR.2023.20, https://doi.org/10.4230/LIPIcs.CONCUR.2023.20
15. Ishtiaq, S., O'Hearn, P.: Separation and Information Hiding. In: Proc. of POPL'01. ACM (2001)
16. Katelaan, J., Jovanovic, D., Weissenbacher, G.: A Separation Logic with Data: Small Models and Automation. In: IJCAR (2018)
17. Katelaan, J., Matheja, C., Noll, T., Zuleger, F.: Harrsh: A Tool for Unied Reasoning about Symbolic-Heap Separation Logic. In: LPAR-22 Workshop and Short Paper Proceedings. vol. 9 (2018)
18. Le, Q.L., Gherghina, C., Qin, S., Chin, W.N.: Shape Analysis via Second-Order Bi-Abduction. In: Proc. of CAV'14. LNCS, vol. 8559. Springer (2014)
19. Le, Q.L.: Compositional Satisfiability Solving in Separation Logic. In: VMCAI. LNCS, vol. 12597 (2021)
20. Le, Q.L., Le, X.B.D.: An Efficient Cyclic Entailment Procedure in a Fragment of Separation Logic. In: FoSSaCS (2023)
21. Matheja, C., Pagel, J., Zuleger, F.: A Decision Procedure for Guarded Separation Logic Complete Entailment Checking for Separation Logic with Inductive Definitions. ACM Trans. Comput. Logic **24**(1) (2023)
22. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD (2009)

23. Navarro Pérez, J.A., Rybalchenko, A.: Separation Logic + Superposition Calculus = Heap Theorem Prover. In: PLDI (2011)
24. Navarro Pérez, J.A., Rybalchenko, A.: Separation Logic Modulo Theories. In: APLAS. LNCS, vol. 8301 (2013)
25. Niemetz, A., Preiner, M.: Bitwuzla. In: CAV. LNCS, vol. 13965 (2023)
26. Pagel, J., Zuleger, F.: Strong-separation logic. ACM Trans. Program. Lang. Syst. **44**(3), 16:1–16:40 (2022). https://doi.org/10.1145/3498847, https://doi.org/10.1145/3498847
27. Piskac, R., Wies, T., Zufferey, D.: Automating Separation Logic Using SMT. In: CAV (2013)
28. Piskac, R., Wies, T., Zufferey, D.: Automating Separation Logic with Trees and Data. In: CAV (2014)
29. Reynolds, A., Iosif, R., King, T.: A Decision Procedure for Separation Logic in SMT. In: ATVA (2016)
30. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science (2002)
31. Santos, J., Maksimovic, P., Ayoun, S.E., Gardner, P.: Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In: Proc. of PLDI'20. ACM (2020)
32. Summers, A.J., Müller, P.: Automating deductive verification for weak-memory programs (extended version). Int. J. Softw. Tools Technol. Transf. **22**(6), 709–728 (2020). https://doi.org/10.1007/s10009-020-00559-y
33. Ta, Q.T., Le, T.C., Khoo, S.C., Chin, W.N.: Automated Lemma Synthesis in Symbolic-Heap Separation Logic. In: POPL (2018)
34. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable Shape Analysis for Systems Code. In: Proc. of CAV'08. LNCS, vol. 5123. Springer (2008)