Strong-Separation Logic

JENS PAGEL and FLORIAN ZULEGER, TU Wien, Austria

Most automated verifiers for separation logic are based on the symbolic-heap fragment, which disallows both the magic-wand operator and the application of classical Boolean operators to spatial formulas. This is not surprising, as support for the magic wand quickly leads to undecidability, especially when combined with inductive predicates for reasoning about data structures. To circumvent these undecidability results, we propose assigning a more restrictive semantics to the separating conjunction. We argue that the resulting logic, strong-separation logic, can be used for symbolic execution and abductive reasoning just like "standard" separation logic, while remaining decidable even in the presence of both the magic wand and inductive predicates (we consider a list-segment predicate and a tree predicate)—a combination of features that leads to undecidability for the standard semantics.

1 INTRODUCTION

Separation logic [Reynolds 2002] is one of the most successful formalisms for the analysis and verification of programs making use of dynamic resources such as heap memory and access permissions [Berdine et al. 2011; Bornat et al. 2005; Calcagno et al. 2015, 2011; Dudka et al. 2011; Jacobs et al. 2011; O'Hearn 2007]. At the heart of the success of separation logic (SL) is the *separating conjunction*, $*$, which supports concise statements about the disjointness of resources. In this article, we will focus on separation logic for describing the heap in single-threaded heap-manipulating programs. In this setting, the formula $\varphi * \psi$ can be read as "the heap can be split into two disjoint parts, such that φ holds for one part and ψ for the other."

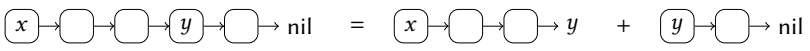
Our article starts from the following observation: The standard semantics of $*$ allows splitting a heap into two arbitrary sub-heaps. The magic-wand operator $-\!\!*$, which is the adjoint of $*$, then allows adding arbitrary heaps. This arbitrary splitting and adding of heaps makes reasoning about SL formulas difficult, and quickly renders separation logic undecidable when inductive predicates for data structures are considered. For example, [Demri et al. 2018] recently showed that adding only the singly-linked list-segment predicate to propositional separation logic (i.e., with $*$, $-\!\!*$ and classical connectives $\wedge$, $\vee$, $\neg$) leads to undecidability.

Most SL specifications used in automated verification do not, however, make use of arbitrary heap compositions. For example, the widely used symbolic-heap fragments of separation logic considered, e.g., in [Berdine et al. 2004, 2005; Cook et al. 2011; Iosif et al. 2013, 2014], have the following property: a symbolic heap satisfies a separating conjunction, if and only if one can split the model at locations that are the values of some program variables.

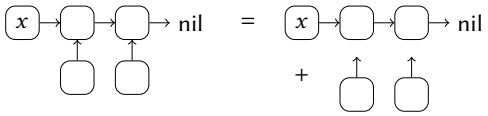
Motivated by this observation, we propose a more restrictive separating conjunction that allows splitting the heap only at locations that are the values of some program variables. We call the resulting logic *strong-separation logic*. Strong-separation logic (SSL) shares many properties with standard separation-logic semantics; for example, the models of our logic form a separation algebra. Because the *frame rule* and other standard SL inference rules continue to hold for SSL, SSL is suitable for deductive Hoare-style verification à la [Ishtiaq and O'Hearn 2001a; Reynolds 2002], symbolic execution [Berdine et al. 2005], as well as abductive reasoning [Calcagno et al. 2015, 2011]. At the same time, SSL has the advantage to be decidable (in PSPACE) for a logic that combines the

Authors' address: Jens Pagel, jens@pagel.codes; Florian Zuleger, florian.zuleger@tuwien.ac.at, TU Wien, Vienna, Austria.

0164-0925/2022/2-ART $15.00
https://doi.org/10.1145/3498847



(a) A model of $\texttt{ls}(x, y) * \texttt{ls}(y, \text{nil})$ in both the standard semantics and our semantics.



(b) A model of $\texttt{ls}(x, \text{nil}) * \texttt{t}$ in the standard semantics.

Fig. 1. Two models and their decomposition into disjoint submodels. The dangling arrows represent dangling pointers.

singly-linked list-segment predicate, classical negation and the magic wand, which is undecidable over standard semantics [Demri et al. 2018]; moreover, the PSPACE complexity matches the complexity of the same fragment without the singly-linked list-segment predicate over standard semantics [Calcagno et al. 2001].

We now give a more detailed introduction to the contributions of this article.

The standard semantics of the separating conjunction. To be able to justify our changed semantics of $*$, we need to introduce a bit of terminology. As standard in separation logic, we interpret SL formulas over *stack–heap pairs*. A *stack* is a mapping of the program variables to memory locations. A *heap* is a finite partial function between memory locations; if a memory location l is mapped to location l', we say the heap contains a *pointer* from l to l'. A memory location l is *allocated* if there is a pointer of the heap from l to some location l'. We call a location *dangling* if it is the target of a pointer but not allocated; a pointer is dangling if its target location is dangling.

Dangling pointers arise naturally in compositional specifications, i.e., in formulas that employ the separating conjunction $*$: In the standard semantics of separation logic, a stack–heap pair (s, h) satisfies a formula $\varphi * \psi$, if it is possible to split the heap h into two disjoint parts h_1 and h_2 such that (s, h_1) satisfies φ and (s, h_2) satisfies ψ. Here, disjoint means that the allocated locations of h_1 and h_2 are disjoint; however, the targets of the pointers of h_1 and h_2 do not have to be disjoint.

We illustrate this in Fig. 1a, where we show a graphical representation of a stack–heap pair (s, h) that satisfies the formula $\texttt{ls}(x, y) * \texttt{ls}(y, \text{nil})$. Here, $\texttt{ls}$ denotes the list-segment predicate. As shown in Fig. 1a, h can be split into two disjoint parts h_1 and h_2 such that (s, h_1) is a model of $\texttt{ls}(x, y)$ and (s, h_2) is a model of $\texttt{ls}(y, \text{nil})$. Now, h_1 has a dangling pointer with target $s(y)$ (displayed by the arrow to y), while no pointer is dangling in the heap h.

In what sense is the standard semantics too permissive? The standard semantics of $*$ allows splitting a heap into two arbitrary sub-heaps, which may result in the introduction of arbitrary dangling pointers into the sub-heaps. We note, however, that the introduction of dangling pointers is *not* arbitrary when splitting the models of $\texttt{ls}(x, y) * \texttt{ls}(y, \text{nil})$; there is only one way of splitting the models of this formula, namely at the location of program variable y. The formula $\texttt{ls}(x, y) * \texttt{ls}(y, \text{nil})$ belongs to a certain variant of the symbolic-heap fragment of separation logic, and all formulas of this fragment have the property that their models can only be split at locations that are the values of some variables.

Standard SL semantics also allows the introduction of dangling pointers without the use of variables. Fig. 1b shows a model of $\texttt{ls}(x, \text{nil}) * \texttt{t}$—assuming the standard semantics. Here, the formula $\texttt{t}$ (for *true*) stands for any arbitrary heap. In particular, this includes heaps with arbitrary dangling pointers into the list segment $\texttt{ls}(x, \text{nil})$. This power of introducing arbitrary dangling pointers is what is used by [Demri et al. 2018] for their undecidability proof of propositional separation logic with the singly-linked list-segment predicate.

Strong-separation logic. In this article, we want to explicitly *disallow* the *implicit* sharing of dangling locations when composing heaps. We propose to parameterize the separating conjunction by the stack and exclusively allow the union of heaps that only share locations that are pointed to by the stack. For example, the model in Fig. 1b is *not* a model of $\mathtt{ls}(x, \mathsf{nil}) * \mathtt{t}$ in our semantics because of the dangling pointers in the sub-heap that satisfies t. *Strong-separation logic* (SSL) is the logic resulting from this restricted definition of the separating conjunction.

Why should I care? We argue that SSL is a promising proposal for automated program verification:

1) We show that the memory models of strong-separation logic form a *separation algebra* [Calcagno et al. 2007], which guarantees the soundness of the standard *frame rule* of SL [Reynolds 2002]. Consequently, SSL can be potentially be used instead of standard SL in a wide variety of (semi-)automated analyzers and verifiers, including Hoare-style verification [Reynolds 2002], symbolic execution [Berdine et al. 2005], and bi-abductive shape analysis [Calcagno et al. 2011].

2) To date, most automated reasoners for separation logic have been developed for *symbolic-heap separation logic* [Berdine et al. 2004, 2005; Calcagno et al. 2011; Iosif et al. 2013, 2014; Katelaan et al. 2019; Katelaan and Zuleger 2020; Pagel et al. 2020]. In these fragments of separation logic, assertions about the heap can exclusively be combined via $*$; neither the magic wand $-\!\!*$ nor classical Boolean connectives are permitted. We show that the strong semantics agrees with the standard semantics on symbolic heaps. For this reason, symbolic-heap SL specifications remain unchanged when switching to strong-separation logic.

3) We establish that the satisfiability and entailment problem for full propositional separation logic with a singly-linked list-segment predicate and a tree predicate is decidable in our semantics (in PSpace)—in stark contrast to the aforementioned undecidability result obtained by [Demri et al. 2018] assuming the standard semantics.

4) The standard Hoare-style approach to verification requires discharging verification conditions (VCs), which amounts to proving for loop-free pieces of code that a pre-condition implies some post-condition. Discharging VCs can be automated by calculi that symbolically execute the pre-condition forward resp. the post-condition backward, and then using an entailment checker for proving the implication. For SL, symbolic execution calculi can be formulated using the magic wand resp. the septraction operator. However, these operators have proven to be difficult for automated procedures: "VC-generators do not work especially well with separation logic, as they introduce magic-wand $-\!\!*$ operators which are difficult to eliminate." [Appel 2014, p. 131] In contrast, we demonstrate that SSL can overcome the described difficulties. We formulate a forward symbolic execution calculus for a simple heap-manipulating programming language using SSL. In conjunction with our entailment checker, see 3), our calculus gives rise to a fully-automated procedure for discharging VCs of loop-free code segments.

5) Computing solutions to the *abduction problem* is an integral building block of Facebook's Infer analyzer [Calcagno et al. 2015], required for a scalable and fully-automated shape analysis [Calcagno et al. 2011]. We show how to compute explicit representations of optimal, i.e., *logically weakest* and *spatially minimal*, solutions to the abduction problem for the separation logic considered in this paper. The result is of theoretical interest, as explicit representations for optimal solutions to the abduction problem are hard to obtain [Calcagno et al. 2011; Gorogiannis et al. 2011].

Contributions. Our main contributions are as follows:

(1) We propose and motivate *strong-separation logic* (SSL), a new semantics for separation logic.
(2) We present a PSpace decision procedure for strong-separation logic with points-to assertions, a list-segment predicate, a tree predicate, as well as spatial and Boolean operators, i.e., $*$, $-\!\!*$, $\wedge$, $\vee$, $\neg$—a logic that is undecidable when assuming the standard semantics [Demri et al. 2018].

(3) We present symbolic execution rules for SSL, which allow us to discharge verification conditions fully automatically.
(4) We show how to compute explicit representations of optimal solutions to the abduction problem for the SSL considered in (2).

We strongly believe that these results motivate further research on SSL (e.g., going beyond the singly-linked list-segment predicate, implementing our decision procedure and integrating it into fully-automated analyzers).

Journal version. This journal version substantially extends the conference version of this paper [Pagel and Zuleger 2021] in several regards:

(1) We have added a *tree predicate* to the considered separation logic, while the only data-structure predicate in the conference version of this paper was the list-segment predicate. We show that all our decidability and complexity results continue to hold for the extended logic. For didactic reasons, we still first introduce a separation logic that only has the list-segment predicate and develop our decision procedure for this restricted logic. After that we extend our results to trees in a separate section (Section 3.8).
(2) We present an operational semantics for the program statements considered in the program verification section (Section 4) and prove the correctness of our symbolic execution calculus with regard to this semantics. The operational semantics and the proof of correctness were left out in the conference version for space reasons.
(3) We improve the exposition of the section on normal forms and abduction (Section 5) by adding the missing proofs, improving the formula that characterizes abstract memory states, and adding the result that the normal form transformation is a *closure operator* (see Theorem 5.4).
(4) For the result on the closure operator to hold, we had to adapt and improve the definition of the chunk size of a formula (see Section 3.5). The new definition gives strictly smaller bounds on the number of chunks than the definition from the conference version. This improvement is not only helpful for the result on the closure operator, but will also have practical impact in future implementations of our decision procedure.
(5) We give all proofs that were left out in the conference version due to space reasons.

Related work. The undecidability of separation logic was established already in [Calcagno et al. 2001]. Since then, decision problems for a large number of fragments and variants of separation logic have been studied. Most of this work has been on symbolic-heap separation logic or other variants of the logic that neither support the magic wand nor the use of negation below the $*$ operator. While entailment in the symbolic-heap fragment with inductive definitions is undecidable in general [Antonopoulos et al. 2014], there are decision procedures for variants with built-in lists and/or trees [Berdine et al. 2004; Cook et al. 2011; Pérez and Rybalchenko 2013; Piskac et al. 2013, 2014], support for defining variants of linear structures [Gu et al. 2016] or tree structures [Iosif et al. 2014; Tatsuta and Kimura 2015] or graphs of bounded tree width [Iosif et al. 2013; Katelaan et al. 2019]. The expressive heap logics Strand [Madhusudan et al. 2011] and Dryad [Qiu et al. 2013] also have decidable fragments, as have some other separation logics that allow combining shape and data constraints. Besides the already mentioned work [Piskac et al. 2013, 2014], these include [Katelaan et al. 2018; Le et al. 2017].

Among the aforementioned works, the graph-based decision procedures of [Cook et al. 2011] and [Katelaan et al. 2018] are most closely related to our approach. Note however, that neither of these works supports reasoning about magic wands or negation below the separating conjunction.

In contrast to symbolic-heap SL, separation logics with the *magic wand* quickly become undecidable. Propositional separation logic with the magic wand, but without inductive data structures, was shown to be decidable in PSpace in the early days of SL research [Calcagno et al. 2001]. Support for this fragment was added to CVC4 a few years ago [Reynolds et al. 2016]. Some tools have "lightweight" support for the magic wand involving

heuristics and user annotations, in part motivated by the lack of decision procedures [Blom and Huisman 2015; Schwerhoff and Summers 2015].

There is a significant body of work studying first-order SL with the magic wand and unary points-to assertions, but without a list predicate. This logic was first shown to be undecidable in [Brochenin et al. 2012]; a result that has since been refined, showing e.g. that while satisfiability is still in PSpace if we allow one quantified variable [Demri et al. 2014], two variables already lead to undecidability, even without the separating conjunction [Demri and Deters 2014]. [Echenim et al. 2019] have recently addressed the satisfiability problem of SL with $\exists^*\forall^*$ quantifier prefix, separating conjunction, magic wand, and full Boolean closure, but no inductive definitions. The logic was shown to be undecidable in general (contradicting an earlier claim [Reynolds et al. 2017]), but decidable in PSpace under certain restrictions.

Above, we have focussed on the related work with regard to automated decision procedures. Here, we also mention several projects that target general and powerful frameworks rather than automation. Iris [Jung et al. 2018], FCSL [Sergey et al. 2015] and TaDA [da Rocha Pinto et al. 2014] provide frameworks for the verification of fine-grained concurrent programs, supporting higher-order functions, concurrency, ownership and rely-guarantee reasoning. The separation logics employed in these frameworks are parameterized by the underlying separation algebras resp. resource monoids, which can be specified by the user. Iris [Jung et al. 2018] and FCSL [Sergey et al. 2015] have been formalized in the Coq Proof assistant ensuring the soundness of the meta-theory. Because the cited approaches provide versatile and expressive frameworks, the involved logics are typically not decidable and proofs need to be done manually (resp. interactively making use of Coq proof tactics), whereas we propose in this paper a specific separation logic and establish decision procedures and complexity results for the considered logic. We further mention that our logic is a *classical* separation logic allowing to prove the absence of memory leaks. In contrast, Iris uses an *intuitionistic* semantics[1], which does not allow proving the absence of resources; this design choice has been made for principle reasons because the later modality supported by Iris does not allow to incorporate the law of excluded middle [Jung et al. 2018]. Relatedly, TaDA [da Rocha Pinto et al. 2014] employs predicates that are upwards-closed sets of worlds, i.e., an intuitionistic semantics. At the present stage, it is difficult to determine whether the TaDA framework could be adapted to classical semantics. We finally mention the flow framework [Krishna et al. 2018, 2020], which identifies separation algebras that can be used for reasoning about global graph properties such as reachabiltiy, acyclicity,etc., in a modular way. The goal of the flow framework was to identify the mathematical foundations for such reasoning while leaving the (promising) automation of flow-based proofs for future work. With regard to automation we remark that the general framework will likely not admit decidability results without putting further restrictions on the considered separation algebras[2].

Outline. In Section 2, we introduce two semantics of propositional separation logic, the standard semantics and our new *strong-separation* semantics. We show the decidability of the satisfiability and entailment problems of SSL with lists and trees in Section 3 (we first show the decidability for SSL with lists but without trees, and then extend our results to trees in Section 3.8). We present symbolic execution rules for SSL in Section 4. We show how to compute explicit representations of optimal solutions to the abduction problem in Section 5. We conclude in Section 6.

[1]Assertions in intuitionistic separation logic satisfy the following monotonicity property: an assertion that is true for some portion of the heap remains true for any extension of the heap [Reynolds 2002]. The classical version of separation logic does not impose this monotonicity property and can therefore be used to reason about explicit storage deallocation.

[2]For comparison we also quote from the extended version [Krishna et al. 2019] why the separating implication was omitted from the logic: "Most presentations of SL also include the separating implication connective $-\!\!*$. However, logics including $-\!\!*$ are harder to automate and usually undecidable. By omitting $-\!\!*$ we emphasize that we do not require it to perform flow-based reasoning." (We recall that in this paper we establish decidability results for a separation logic that includes $-\!\!*$).

$$\begin{array}{lcl} \tau & ::= & \mathbf{emp} \mid x \mapsto y \mid \mathtt{ls}(x,y) \mid x = y \mid x \neq y \\ \varphi & ::= & \tau \mid \varphi * \varphi \mid \varphi \mathrel{-\!\!\circledast} \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \end{array}$$

Fig. 2. The syntax of separation logic with list segments.

2 STRONG- AND WEAK-SEPARATION LOGIC

2.1 Preliminaries

We denote by $|X|$ the cardinality of the set X. Let f be a (partial) function. Then, $\mathrm{dom}(f)$ and $\mathrm{img}(f)$ denote the domain and image of f, respectively. We write $|f| := |\mathrm{dom}(f)|$ and $f(x) = \bot$ for $x \notin \mathrm{dom}(f)$. We frequently use set notation to define and reason about partial functions: $f := \{x_1 \mapsto y_1, \ldots, x_k \mapsto y_k\}$ is the partial function that maps x_i to y_i, $1 \leq i \leq k$, and is undefined on all other values; $f^{-1}(b)$ is the set of all elements a with $f(a) = b$; we write $f \cup g$ resp. $f \cap g$ for the union resp. intersection of partial functions f and g, provided that $f(a) = g(a)$ for all $a \in \mathrm{dom}(f) \cap \mathrm{dom}(g)$; similarly, $f \subseteq g$ holds if $\mathrm{dom}(f) \subseteq \mathrm{dom}(g)$. Given a partial function f, we denote by $f[x/v]$ the updated partial function in which x maps to v, i.e.,

$$f[x/v](y) = \begin{cases} v, & \text{if } y = x, \\ f(y) & \text{otherwise,} \end{cases}$$

where we use $v = \bot$ to express that the updated function $f[x/v]$ is undefined for x.

Sets and ordered sequences are denoted in boldface, e.g., $\mathbf{x}$. To list the elements of a sequence, we write $\langle x_1, \ldots, x_k \rangle$.

We assume a linearly-ordered infinite set of variables **Var** with $\mathsf{nil} \in \mathbf{Var}$ and denote by $\max(\mathbf{v})$ the maximal variable among a set of variables $\mathbf{v}$ according to this order. In Fig. 2, we define the syntax of the separation-logic fragment we study in this article. The atomic formulas of our logic are the *empty-heap predicate* **emp**, *points-to assertions* $x \mapsto y$, the *list-segment predicate* $\mathtt{ls}(x, y)$, equalities $x = y$ and disequalities $x \neq y$[3]; in all these cases, $x, y \in \mathbf{Var}$. (We note that for the moment our separation logic does not include a tree predicate. We defer this extension to Section 3.8.) Formulas are closed under the classical Boolean operators $\wedge, \vee, \neg$ as well as under the *separating conjunction* $*$ and the existential magic wand, also called *septraction*, $\mathrel{-\!\!\circledast}$ (see e.g. [Brochenin et al. 2012]). We collect the set of all SL formulas in **SL**. We also consider derived operators and formulas, in particular the *separating implication* (or *magic wand*), $\mathrel{-\!\!*}$, defined by $\varphi \mathrel{-\!\!*} \psi := \neg(\varphi \mathrel{-\!\!\circledast} \neg\psi)$.[4] We also use *true*, defined as $\mathsf{t} := \mathbf{emp} \vee \neg\mathbf{emp}$. Finally, for $\Phi = \{\varphi_1, \ldots, \varphi_n\}$, we set $\mathbin{\ast}\Phi := \varphi_1 * \varphi_2 * \cdots * \varphi_n$, if $n > 1$, and $\mathbin{\ast}\Phi := \mathbf{emp}$, if $n = 0$. By $\mathrm{fvs}(\varphi)$ we denote the set of (free) variables of φ. We define the *size* of the formula φ as $|\varphi| = 1$ for atomic formulas φ, $|\varphi_1 \times \varphi_2| := |\varphi_1| + |\varphi_2| + 1$ for $\times \in \{\wedge, \vee, *, \mathrel{-\!\!\circledast}\}$ and $|\neg\varphi_1| := |\varphi_1| + 1$.

2.2 Two Semantics of Separation Logic

Memory model. **Loc** is an infinite set of *heap locations*. A *stack* is a partial function $s \colon \mathbf{Var} \rightharpoonup \mathbf{Loc}$. A *heap* is a partial function $h \colon \mathbf{Loc} \rightharpoonup \mathbf{Loc}$. A *model* is a stack–heap pair (s, h) with $\mathsf{nil} \in \mathrm{dom}(s)$ and $s(\mathsf{nil}) \notin \mathrm{dom}(h)$. We let $\mathrm{locs}(h) := \mathrm{dom}(h) \cup \mathrm{img}(h)$. A location ℓ is *dangling* if $\ell \in \mathrm{img}(h) \setminus \mathrm{dom}(h)$. We write **S** for the set of all stacks and **H** for the set of all heaps.

[3]Note that $x \neq y$ is *not* equivalent to $\neg(x = y)$ in our separation logic, as we require the heap to be empty for all models of $x \neq y$.
[4]As $\mathrel{-\!\!*}$ can be defined via $\mathrel{-\!\!\circledast}$ and $\neg$ and vice-versa, the expressivity of our logic does not depend on which operator we choose. We have chosen $\mathrel{-\!\!\circledast}$ because we can include this operator in the positive fragment considered later on.

$$
\begin{array}{lll}
(s,h) \models \mathbf{emp} & \text{iff} & \mathrm{dom}(h) = \emptyset \\
(s,h) \models x = y & \text{iff} & \mathrm{dom}(h) = \emptyset \text{ and } s(x) = s(y) \\
(s,h) \models x \neq y & \text{iff} & \mathrm{dom}(h) = \emptyset \text{ and } s(x) \neq s(y) \\
(s,h) \models x \mapsto y & \text{iff} & h = \{s(x) \mapsto s(y)\} \\
(s,h) \models \mathtt{ls}(x,y) & \text{iff} & \mathrm{dom}(h) = \emptyset \text{ and } s(x) = s(y) \text{ or there exist } n \geq 1, \ell_0, \ldots, \ell_n \text{ with} \\
 & & \quad h = \{\ell_0 \mapsto \ell_1, \ldots, \ell_{n-1} \mapsto \ell_n\}, s(x) = \ell_0 \text{ and } s(y) = \ell_n \\
(s,h) \models \varphi_1 \wedge \varphi_2 & \text{iff} & (s,h) \models \varphi_1 \text{ and } (s,h) \models \varphi_2 \\
(s,h) \models \neg\varphi & \text{iff} & (s,h) \not\models \varphi \\
(s,h) \models^{\mathsf{wk}} \varphi_1 * \varphi_2 & \text{iff} & \text{there exist } h_1, h_2 \text{ with } h = h_1 + h_2, (s,h_1) \models^{\mathsf{wk}} \varphi_1, \text{ and } (s,h_2) \models^{\mathsf{wk}} \varphi_2 \\
(s,h) \models^{\mathsf{wk}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2 & \text{iff} & \text{exist } h_1, \text{ with } (s,h_1) \models^{\mathsf{wk}} \varphi_1, h + h_1 \neq \bot \text{ and } (s, h + h_1) \models^{\mathsf{wk}} \varphi_2 \\
(s,h) \models^{\mathsf{st}} \varphi_1 * \varphi_2 & \text{iff} & \text{there exists } h_1, h_2 \text{ with } h = h_1 \uplus^s h_2, (s,h_1) \models^{\mathsf{st}} \varphi_1, \text{ and } (s,h_2) \models^{\mathsf{st}} \varphi_2 \\
(s,h) \models^{\mathsf{st}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2 & \text{iff} & \text{exists } h_1 \text{ with } (s,h_1) \models^{\mathsf{st}} \varphi_1, h \uplus^s h_1 \neq \bot \text{ and } (s, h \uplus^s h_1) \models^{\mathsf{st}} \varphi_2
\end{array}
$$

Fig. 3. The standard, "weak" semantics of separation logic, $\models^{\mathsf{wk}}$, and the "strong" semantics, $\models^{\mathsf{st}}$. We write $\models$ when there is no difference between $\models^{\mathsf{wk}}$ and $\models^{\mathsf{st}}$.

Two notions of disjoint union of heaps. We write $h_1 + h_2$ for the union of disjoint heaps, i.e.,

$$h_1 + h_2 := \begin{cases} h_1 \cup h_2, & \text{if } \mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset \\ \bot, & \text{otherwise.} \end{cases}$$

This standard notion of disjoint union is commonly used to assign semantics to the separating conjunction and magic wand. It requires that h_1 and h_2 are domain-disjoint, but does not impose any restrictions on the *images* of the heaps. In particular, the dangling pointers of h_1 may alias arbitrarily with the domain of h_2 and vice-versa.

Let s be a stack. We write $h_1 \uplus^s h_2$ for the disjoint union of h_1 and h_2 that restricts aliasing of dangling pointers to the locations in stack s. This yields an infinite family of union operators: one for each stack. Formally,

$$h_1 \uplus^s h_2 := \begin{cases} h_1 + h_2, & \text{if } (\mathrm{dom}(h_1) \cap \mathrm{img}(h_2)) \cup (\mathrm{dom}(h_2) \cap \mathrm{img}(h_1)) \subseteq \mathrm{img}(s) \\ \bot, & \text{otherwise.} \end{cases}$$

Intuitively, $h_1 \uplus^s h_2$ is the (disjoint) union of heaps whose dangling pointers may only point to the domain of the other heap in case the targets of these dangling pointers are in the image of s. Note that if $h_1 \uplus^s h_2$ is defined then $h_1 + h_2$ is defined, but not vice-versa.

Just like the standard disjoint union +, the operator $\uplus^s$ gives rise to a separation algebra, i.e., a cancellative, commutative partial monoid [Calcagno et al. 2007]:

Lemma 2.1. *Let s be a stack and let u be the empty heap (i.e., $\mathrm{dom}(u) = \emptyset$). The triple $(\mathbf{H}, \uplus^s, u)$ is a separation algebra.*

Proof. Trivially, the operation $\uplus^s$ is commutative and associative with unit u. Let $h \in \mathbf{H}$. Consider $h_1, h_2 \in \mathbf{H}$ such that $h \uplus^s h_1 = h \uplus^s h_2 \neq \bot$. Since the domain of h is disjoint from the domains of h_1 and h_2, it follows that for all x, $h_1(x) = h_2(x)$ and thus $h_1 = h_2$. As h_1 and h_2 were chosen arbitrarily, we obtain that the function $h \uplus^s (\cdot)$ is injective. Consequently, the monoid is cancellative. □

Weak- and strong-separation logic. Both + and $\uplus^s$ can be used to give a semantics to the separating conjunction and septraction. We denote the corresponding model relations $\models^{\mathsf{wk}}$ and $\models^{\mathsf{st}}$ and define them in Fig. 3. Where the two semantics agree, we simply write $\models$.

Fig. 4. Two models of $(\mathtt{ls}(a, \mathsf{nil}) * \mathsf{t}) \wedge (\mathtt{ls}(b, \mathsf{nil}) * \mathsf{t})$ for a stack with domain a, b and a stack with domain a, b, c.

In both semantics, **emp** only holds for the empty heap, and $x = y$ holds for the empty heap when x and y are interpreted by the same location[5]. Points-to assertions $x \mapsto y$ are precise, i.e., only hold in singleton heaps. (Following [Ishtiaq and O'Hearn 2001b], it is, of course, possible to express intuitionistic points-to assertions by $x \mapsto y * \mathsf{t}$). The list segment predicate $\mathtt{ls}(x, y)$ holds in possibly-empty lists of pointers from $s(x)$ to $s(y)$. The semantics of Boolean connectives are standard. The semantics of the separating conjunction, $*$, and septraction, $-\!\circledast$, differ based on the choice of + vs. $\uplus^s$ for combining disjoint heaps. In the former case, denoted $\models^{\mathsf{wk}}$, we get the standard semantics of separation logic (cf. [Reynolds 2002]). In the latter case, denoted $\models^{\mathsf{st}}$, we get a semantics that imposes stronger requirements on sub-heap composition: Sub-heaps may only overlap at locations that are stored in the stack.

Because the semantics $\models^{\mathsf{st}}$ imposes stronger constraints, we will refer to the standard semantics $\models^{\mathsf{wk}}$ as the *weak* semantics of separation logic and to the semantics $\models^{\mathsf{st}}$ as the *strong* semantics of separation logic. Moreover, we use the terms *weak-separation logic* (WSL) and *strong-separation logic* (SSL) to distinguish between SL with the semantics $\models^{\mathsf{wk}}$ and $\models^{\mathsf{st}}$.

Example 2.2. Let $\varphi := a \neq b * (\mathtt{ls}(a, \mathsf{nil}) * \mathsf{t}) \wedge (\mathtt{ls}(b, \mathsf{nil}) * \mathsf{t})$. In Fig. 4, we show two models of φ. On the left, we assume that a, b are the only program variables, whereas on the right, we assume that there is a third program variable c.

Note that the latter model, where the two lists overlap, is possible in SSL *only* because the lists come together at the location labeled by c. If we removed variable c from the stack, the model would no longer satisfy φ according to the strong semantics, because $\uplus^s$ would no longer allow splitting the heap at that location. Conversely, the model would still satisfy φ with standard semantics.

This is a feature rather than a bug of SSL: Without having a variable c the stack-heap pair on the right of Fig. 4 is not a model of φ. However, an SSL user is able to explicitly allow such models by adding a (ghost) variable c to the set of program variables.

Isomorphism. For later use, we state that SL formulas cannot distinguish isomorphic models:

Definition 2.3. Let $(s, h), (s', h')$ be models. (s, h) and (s', h') are *isomorphic*, $(s, h) \cong (s', h')$, if there exists a bijection $\sigma\colon (\mathrm{locs}(h) \cup \mathrm{img}(s)) \to (\mathrm{locs}(h') \cup \mathrm{img}(s'))$ such that (1) for all x, $s'(x) = \sigma(s(x))$ and (2) $h' = \{\sigma(l) \mapsto \sigma(h(l)) \mid l \in \mathrm{dom}(h)\}$.

Lemma 2.4. *Let $(s, h), (s', h')$ be models with $(s, h) \cong (s', h')$ and let $\varphi \in$ SL. Then $(s, h) \models^{\mathsf{st}} \varphi$ iff $(s', h') \models^{\mathsf{st}} \varphi$.*

Proof. See appendix. □

(Lemma 2.4.) We prove the claim by induction on the structure of the formula φ. Clearly, the claim holds for the base cases **emp**, $x \mapsto y$, $\mathtt{ls}(x, y)$, $x = y$ and $x \neq y$. Further, the claim immediately follows from the induction assumption for the cases $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ and $\neg\varphi$. It remains to consider the cases $\varphi_1 * \varphi_2$ and $\varphi_1 -\!\circledast \varphi_2$. Let (s, h) and (s', h') be two stack-heap pairs with $(s, h) \cong (s', h')$.

[5]Usually $x = y$ is defined to hold for *all* heaps, not just the empty heap, when x and y are interpreted by the same location; however, this choice does not change the expressivity of the logic: the formula $(x = y) * \mathsf{t}$ expresses the standard semantics. Our choice is needed for the results on the positive fragment considered in Section 2.3.

We will show that $(s,h) \models^{\mathsf{st}} \varphi_1 * \varphi_2$ implies $(s',h') \models^{\mathsf{st}} \varphi_1 * \varphi_2$; the other direction is completely symmetric. We assume that $(s,h) \models^{\mathsf{st}} \varphi_1 * \varphi_2$. Then, there are h_1, h_2 with $h_1 \uplus^s h_2 = h$ and $(s,h_i) \models^{\mathsf{st}} \varphi_i$ for $i = 1,2$. We consider the bijection σ that witnesses the isomorphism between (s,h) and (s',h'). Let h'_1 resp. h'_2 be the sub-heap of h' restricted to $\sigma(\mathrm{dom}(h_1))$ resp. $\sigma(\mathrm{dom}(h_2))$. It is easy to verify that $h'_1 \uplus^s h'_2 = h'$ and $(s,h_i) \cong (s',h'_i)$ for $i = 1,2$. Hence, we can apply the induction assumption and get that $(s',h'_i) \models^{\mathsf{st}} \varphi_i$ for $i = 1,2$. Because of $h'_1 \uplus^s h'_2 = h'$ we get $(s',h') \models^{\mathsf{st}} \varphi_1 * \varphi_2$.

We will show that $(s,h) \models^{\mathsf{st}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2$ implies $(s',h') \models^{\mathsf{st}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2$; the other direction is completely symmetric. We assume that $(s,h) \models^{\mathsf{st}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2$. Hence there is a heap h_0 with $(s,h_0) \models^{\mathsf{st}} \varphi_1$ and $(s, h_0 \uplus^s h) \models^{\mathsf{st}} \varphi_2$. We note that in particular $h_0 \uplus^s h \neq \bot$. We consider the bijection σ that witnesses the isomorphism between (s,h) and (s',h'). Let $L \subseteq \mathbf{Loc}$ be some subset of locations with $L \cap (\mathrm{locs}(h') \cup \mathrm{img}(s')) = \emptyset$ and $|L| = \mathrm{locs}(h_0) \setminus (\mathrm{locs}(h) \cup \mathrm{img}(s))$. We can extend σ to some bijective function $\sigma' : (\mathrm{locs}(h_0) \cup \mathrm{locs}(h) \cup \mathrm{img}(s)) \to (L \cup \mathrm{locs}(h') \cup \mathrm{img}(s'))$. Then, σ' induces a heap h'_0 such that $(s,h_0) \cong (s',h'_0)$, $h'_0 \uplus^s h' \neq \bot$ and $(s, h_0 \uplus^s h) \cong (s', h'_0 \uplus^s h')$. By induction assumption we get that $(s',h'_0) \models^{\mathsf{st}} \varphi_1$ and $(s', h'_0 \uplus^s h') \models^{\mathsf{st}} \varphi_2$. Hence, $(s',h') \models^{\mathsf{st}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2$.

Satisfiability and Semantic Consequence. We define the notions of satisfiability and semantic consequence parameterized by a finite set of variables $\mathbf{x} \subseteq \mathbf{Var}$. For a formula φ with $\mathrm{fvs}(\varphi) \subseteq \mathbf{x}$, we say that φ is *satisfiable* w.r.t. $\mathbf{x}$ if there is a model (s,h) with $\mathrm{dom}(s) = \mathbf{x}$ such that $(s,h) \models^{\mathsf{st}} \varphi$. We say that φ *entails* ψ w.r.t. $\mathbf{x}$, in signs $\varphi \models^{\mathsf{st}}_{\mathbf{x}} \psi$, if $(s,h) \models^{\mathsf{st}} \varphi$ then also $(s,h) \models^{\mathsf{st}} \psi$ for all models (s,h) with $\mathrm{dom}(s) = \mathbf{x}$.

2.3 Correspondence of Strong and Weak Semantics on Positive Formulas

We call an SL formula φ *positive* if it does not contain $\neg$. Note that, in particular, this implies that φ does *not* contain the magic wand $\mathrel{-\!\!*}$ or the atom t.

In models of positive formulas, all dangling locations are labeled by variables:

Lemma 2.5. *Let φ be positive and $(s,h) \models^{\mathsf{wk}} \varphi$. Then,* $(\mathrm{img}(h) \setminus \mathrm{dom}(h)) \subseteq \mathrm{img}(s)$.

Proof. We prove the following stronger statement by structural induction on φ: For every model $(s,h) \models^{\mathsf{wk}} \varphi$ we have that

(1) $(\mathrm{img}(h) \setminus \mathrm{dom}(h)) \subseteq \mathrm{img}(s)$,
(2) every *join point* is labelled by a variable, i.e., $\left|h^{-1}(\ell)\right| \geq 2$ implies that $\ell \in \mathrm{img}(s)$, and
(3) every *source* is labelled by a variable, i.e., $(\mathrm{dom}(h) \setminus \mathrm{img}(h)) \subseteq \mathrm{img}(s)$.

The proof is straightforward except for the $\mathrel{-\!\!\circledast}$ case: Assume $(s,h) \models^{\mathsf{wk}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2$. Then there is a h_0 with $(s,h_0) \models^{\mathsf{wk}} \varphi_1$ and $(s, h_0 + h) \models^{\mathsf{wk}} \varphi_2$. By induction assumption the claim holds for (s,h_0) and (s,h_0+h). We note that every join point of h is also a join point of $h_0 + h$ and hence labelled by a variable. We now verify that every pointer that is dangling in h is either also dangling in $h_0 + h$ or is a join point in $h_0 + h$ or is pointing to a source of h_0; in all cases the target of the dangling pointer is labelled by a variable. Finally, a source of h is either also a source of $h_0 + h$ or is pointed to by a dangling pointer of h_0; in both cases the source is labelled by a variable. □

As every location shared by heaps h_1 and h_2 in $h_1 + h_2$ is either dangling in h_1 or in h_2 (or in both), the operations $+$ and $\uplus^s$ coincide on models of positive formulas:

Lemma 2.6. *Let $(s,h_1) \models^{\mathsf{wk}} \varphi_1$ and $(s,h_2) \models^{\mathsf{wk}} \varphi_2$ for positive formulas φ_1, φ_2. Then $h_1 + h_2 \neq \bot$ iff $h_1 \uplus^s h_2 \neq \bot$.*

Proof. If $h_1 \uplus^s h_2 \neq \bot$, then $h_1 + h_2 \neq \bot$ by definition.

Conversely, assume $h_1 + h_2 \neq \bot$. We need to show that $\mathrm{locs}(h_1) \cap \mathrm{locs}(h_2) \subseteq \mathrm{img}(s)$. To this end, let $\ell \in \mathrm{locs}(h_1) \cap \mathrm{locs}(h_2)$. Then there exists an $i \in \{1,2\}$ such that $\ell \in \mathrm{img}(h_i) \setminus \mathrm{dom}(h_i)$—otherwise ℓ would be in $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2)$ and $h_1 + h_2 = \bot$. By Lemma 2.5, we thus have $\ell \in \mathrm{img}(s)$. □

Since the semantics coincide on atomic formulas by definition and on $*$ by Lemma 2.5, we can easily show that they coincide on all positive formulas:

Lemma 2.7. *Let φ be a positive formula and let (s,h) be a model. Then, $(s,h) \models^{\text{wk}} \varphi$ iff $(s,h) \models^{\text{st}} \varphi$.*

Proof. We proceed by structural induction on φ. If φ is atomic, there is nothing to show. For $\varphi = \varphi_1 * \varphi_2$ and $\varphi = \varphi_1 \circledast \varphi_2$, the claim follows from the induction hypotheses and Lemma 2.6. For $\varphi = \varphi_1 \wedge \varphi_2$ and $\varphi = \varphi_1 \vee \varphi_2$, the claim follows immediately from the induction hypotheses and the semantics of $\wedge$, $\vee$.

□

Lemma 2.7 implies that the two semantics coincide on the popular *symbolic-heap fragment* of separation logic.[6] Further, by negating Lemma 2.7, we have that $\{(s,h) \mid (s,h) \models^{\text{wk}} \varphi\} \neq \left\{(s,h) \mid (s,h) \models^{\text{st}} \varphi\right\}$ implies that φ contains negation, either explicitly or in the form of a magic wand or t.

We remark that formula φ in Example 2.2 uses only t but not $\neg, \mathrel{-\!\!*}$. Hence, adding t to the positive fragment is already sufficient to invalidate Lemma 2.7; because t can be defined from $\neg$ resp. $\mathrel{-\!\!*}$, we cannot add either operator to the positive fragment without invalidating Lemma 2.7. Moreover, Lemma 2.7 does not hold under intuitionistic semantics: Recall that the meaning of a formula ζ under intuitionistic semantics is equivalent to the meaning of $\zeta * \mathsf{t}$ under classic semantics [Reynolds 2002]. Hence, the meaning of the formula $\psi := a \neq b * (\mathtt{ls}(a, \mathsf{nil}) \wedge (\mathtt{ls}(b, \mathsf{nil}))$ under intuitionistic semantics is equivalent to formula φ in Example 2.2 under classical semantics. As ψ is from the positive fragment, Lemma 2.7 does not hold under intuitionistic semantics.

3 DECIDING THE SSL SATISFIABILITY PROBLEM

The goal of this section is to develop a decision procedure for SSL:

Theorem 3.1. *Let $\varphi \in \mathbf{SL}$ and let $\mathbf{x} \subseteq \mathbf{Var}$ be a finite set of variables with $\mathrm{fvs}(\varphi) \subseteq \mathbf{x}$. It is decidable in PSpace (in $|\varphi|$ and $|\mathbf{x}|$) whether there exists a model (s,h) with $\mathrm{dom}(s) = \mathbf{x}$ and $(s,h) \models^{\text{st}} \varphi$.*

Our approach is based on abstracting stack–heap models by *abstract memory states* (AMS), which have two key properties, which together imply Theorem 3.1:

Refinement (Theorem 3.19). If (s_1,h_1) and (s_2,h_2) abstract to the same AMS, then they satisfy the same formulas. That is, the AMS abstraction *refines* the satisfaction relation of SSL.

Computability (Theorem 3.42, Lemmas 3.44 and 3.46). For every formula φ, we can compute (in PSpace) the set of all AMSs of all models of φ; then, φ is satisfiable if this set is nonempty.

The AMS abstraction is motivated by the following insights:

(1) The operator $\uplus^s$ induces a unique decomposition of the heap into at most $|s|$ minimal *chunks* of memory that cannot be further decomposed.
(2) To decide whether $(s,h) \models^{\text{st}} \varphi$ holds, it is sufficient to know for each chunk of (s,h) a) which atomic formulas the chunk satisfies and b) which variables (if any) are allocated in the chunk.
(3) We equip the AMS abstract domain with a composition operator $\bullet$ such that AMS abstraction is a homomorphism with regard to $\uplus^s$ and $\bullet$ (see Lemma 3.28); moreover, given a model (s,h) that abstracts to the composition of two AMSs $\mathcal{A}_1 \bullet \mathcal{A}_2$, we can always find a decomposition $h = h_1 \uplus^s h_2$, such that (s,h_i) abstracts to $\mathcal{A}_i$ (see Lemma 3.29). These two properties are the key for proving the Refinement Theorem. We remark that the homomorphism and decomposition properties also were essential for proving the decidability and complexity results for the separation logics considered in [Katelaan et al. 2019; Katelaan and

[6] Strictly speaking, this only holds for the symbolic-heap fragment of the separation logic studied in this paper, i.e., for symbolic-heaps composed of points-to predicates, list predicates and tree predicates (see Section 3.8). We consider the logic in [Iosif et al. 2013], which proposes symbolic heaps of bounded treewidth, as an interesting direction for future work.

Zuleger 2020; Pagel et al. 2020]. Interestingly, the homomorphism and decomposition property have been identified in concurrent work as natural properties for reasoning about framing and parallel composition in separation logic [Farka et al. 2021][7].

We proceed as follows. In Sec 3.1, we make precise the notion of memory chunks. In Sec. 3.2, we define *abstract memory states* (AMS), an abstraction of models that retains for every chunk precisely the information from point (2) above. We will prove the *refinement theorem* in 3.3. We will show in Sections 3.4–3.6 that we can compute the AMS of the models of a given formula φ, which allows us to decide satisfiability and entailment problems for SSL. Finally, we prove the PSpace-completeness result in Sec. 3.7.

3.1 Memory Chunks

We will abstract a model (s, h) by abstracting every *chunk* of h, which is a *minimal* nonempty sub-heap of (s, h) that can be split off of h according to the strong-separation semantics.

Definition 3.2 (Sub-heap). Let (s, h) be a model. We say that h_1 is a *sub-heap* of h, in signs $h_1 \sqsubseteq h$, if there is some heap h_2 such that $h = h_1 \uplus^s h_2$. We collect all sub-heaps in the set $\mathsf{subHeaps}(s, h)$.

Sub-heaps are closed under taking intersection and unions:

Proposition 3.3. *Let (s, h) be a model and let h_1, h_2 be sub-heaps of h. Then, $h_1 \cap h_2$ and $h_1 \cup h_2$ are also sub-heaps of h.*

Proof. By definition of sub-heaps, there are some heaps h_1', h_2' such that $h = h_1 \uplus^s h_1'$ and $h = h_2 \uplus^s h_2'$. We prove the claim for $(h_1 \cap h_2)$. The proof for $(h_1 \cup h_2)$ is analogous. We will now argue that $(h_1 \cap h_2) \uplus^s (h_1' \cup h_2') = h$. Let us consider some $\ell \in \mathrm{dom}(h_1 \cap h_2) \cap \mathrm{img}(h_1' \cup h_2')$. Because of $\ell \in \mathrm{img}(h_1' \cup h_2')$ we have $\ell \in \mathrm{img}(h_i')$ for some $i \in \{1, 2\}$. Then, because of $\ell \in \mathrm{dom}(h_1 \cap h_2)$ we also have $\ell \in \mathrm{dom}(h_i)$. Because of $h = h_i \uplus^s h_i'$ we get that $\ell \in \mathrm{img}(s)$ from the definition of $\uplus^s$. The proof that $\ell \in \mathrm{img}(h_1 \cap h_2) \cap \mathrm{dom}(h_1' \cup h_2')$ implies $\ell \in \mathrm{img}(s)$ is analogous. □

The following proposition is an immediate consequence of Proposition 3.3:

Proposition 3.4. *Let (s, h) be a model. Then, $(\mathsf{subHeaps}(s, h), \sqsubseteq, \sqcup, \sqcap, \neg)$ is a Boolean algebra with greatest element h and smallest element $\emptyset$, where*

- $(s, h_1) \sqcup (s, h_2) := (s, h_1 \cup h_2)$,
- $(s, h_1) \sqcap (s, h_2) := (s, h_1 \cap h_2)$, *and*
- $\neg(s, h_1) := (s, h_1')$, *where $h_1' \in \mathsf{subHeaps}(s, h)$ is the unique sub-heap with $h = h_1 \uplus^s h_1'$.*

The fact that the sub-models form a Boolean algebra allows us to make the following definition[8]:

Definition 3.5 (Chunk). Let (s, h) be a model. A *chunk of* (s, h) is an atom of the Boolean algebra $(\mathsf{subHeaps}(s, h), \sqsubseteq, \sqcup, \sqcap, \neg)$. We collect all chunks of (s, h) in the set $\mathsf{chunks}(s, h)$.

Because every element of a Boolean algebra can be uniquely decomposed into atoms, we obtain that every heap can be fully decomposed into its chunks:

Proposition 3.6. *Let (s, h) be a model and let $\mathsf{chunks}(s, h) = \{h_1, \ldots, h_n\}$ be its chunks. Then, $h = h_1 \uplus^s h_2 \uplus^s \cdots \uplus^s h_n$.*

[7] In [Farka et al. 2021] *decomposability* is termed *invertibility*.

[8] It is an interesting question for future work to relate the chunks considered in this paper to the atomic building blocks used in SL symbolic executions engines. Likewise, it would be interesting to build a symbolic execution engine based on the chunks resp. on the AMS abstraction proposed in this paper.

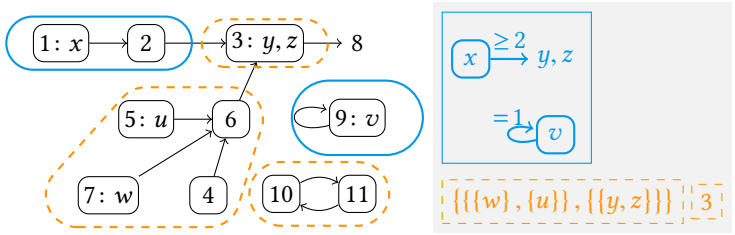


Fig. 5. Graphical representation of a model consisting of five chunks (left, see Ex. 3.7) and its induced AMS (right, see Ex. 3.13).

Example 3.7. Let $s = \{x \mapsto 1, y \mapsto 3, u \mapsto 5, z \mapsto 3, w \mapsto 7, v \mapsto 9\}$ and $h = \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 8, 4 \mapsto 6, 5 \mapsto 6, 6 \mapsto 3, 7 \mapsto 6, 9 \mapsto 9, 10 \mapsto 11, 11 \mapsto 10\}$. The model (s, h) is illustrated in Fig. 5. This time, we include the identities of the locations in the graphical representation; e.g., $3 : y, z$ represents location 3, $s(y) = 3$, $s(z) = 3$. The model consists of five chunks, $h_1 := \{1 \mapsto 2, 2 \mapsto 3\}$, $h_2 := \{9 \mapsto 9\}$, $h_3 := \{4 \mapsto 6, 5 \mapsto 6, 6 \mapsto 3, 7 \mapsto 6\}$, $h_4 := \{3 \mapsto 8\}$, and $h_5 := \{10 \mapsto 11, 11 \mapsto 10\}$.

We distinguish two types of chunks: those that satisfy SSL atoms and those that don't.

Definition 3.8 (Positive and Negative chunk). Let $h_c \subseteq h$ be a chunk of (s, h). h_c is a *positive chunk* if there exists an atomic formula τ such that $(s, h_c) \models^{\text{pt}} \tau$. Otherwise, h_c is a *negative chunk*. We collect the respective chunks in $\text{chunks}^+(s, h)$ and $\text{chunks}^-(s, h)$.

Example 3.9. Recall the chunks h_1 through h_5 from Ex. 3.7. h_1 and h_2 are positive chunks (blue in Fig. 5), h_3 to h_5 are negative chunks (orange).

Negative chunks fall into three (not mutually-exclusive) categories:

Garbage. Chunks with locations that are inaccessible via stack variables.

Unlabeled dangling pointers. Chunks with an unlabeled sink, i.e., a dangling location that is not in $\text{img}(s)$ and thus *cannot* be "made non-dangling" via composition using $\uplus^s$.

Overlaid list segments. Overlaid list segments that cannot be separated via $\uplus^s$ because they are joined at locations that are not in $\text{img}(s)$.

Example 3.10 (Negative chunks). The chunk h_3 from Example 3.7 contains garbage, namely the location 4 that cannot be reached via stack variables, *and* two overlaid list segments (from 5 to 3 and 7 to 3). The chunk h_4 has an unlabeled dangling pointer. The chunk h_5 contains only garbage.

3.2 Abstract Memory States

In *abstract memory states* (AMSs), we retain for every chunk enough information to (1) determine which atomic formulas the chunk satisfies, and (2) keep track of which variables are allocated within each chunk.

Definition 3.11. A quadruple $\mathcal{A} = \langle V, E, \rho, \gamma \rangle$ is an *abstract memory state*, if

(1) V is a *partition* of some finite set of variables, i.e., $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ for some non-empty disjoint finite sets $\mathbf{v}_i \subseteq \mathbf{Var}$,
(2) $E \colon V \rightharpoonup V \times \{=1, \geq 2\}$ is a partial function such that there is no $\mathbf{v} \in \text{dom}(E)$ with $\text{nil} \in \mathbf{v}$[9],
(3) ρ consists of disjoint subsets of V such that every $R \in \rho$ is disjoint from $\text{dom}(E)$ and there is no $\mathbf{v} \in R$ with $\text{nil} \in \mathbf{v}$,

[9]The edges of an AMS represent either a single pointer (case "$=1$") or a list segment of at least length two (case "≥ 2").

(4) γ is a natural number, i.e., $\gamma \in \mathbb{N}$.

We call V the *nodes*, E the *edges*, ρ the *negative-allocation constraint* and γ the *garbage-chunk count* of $\mathcal{A}$. We call the AMS $\mathcal{A} = \langle V, E, \rho, \gamma \rangle$ *garbage-free* if $\rho = \emptyset$ and $\gamma = \emptyset$.

We collect the set of all AMSs in **AMS**. The *size* of $\mathcal{A}$ is given by $|\mathcal{A}| := |V| + \gamma$. Finally, the *allocated variables* of an AMS are given by $\mathbf{alloc}(\mathcal{A}) := \mathrm{dom}(E) \cup \bigcup \rho$.

Every model induces an AMS, defined in terms of the following auxiliary definitions. The equivalence class of variable x w.r.t. stack s is $[x]^s_= := \{y \mid s(y) = s(x)\}$; the set of all equivalence classes of s is $\mathsf{cls}_=(s) := \{[x]^s_= \mid x \in \mathrm{dom}(s)\}$. We now define the edges induced by a model (s, h): For every equivalence class $[x]^s_= \in \mathsf{cls}_=(s)$, we set

$$\mathsf{edges}(s,h)([x]^s_=) := \begin{cases} \langle [y]^s_=, =1 \rangle & \text{there are } y \in \mathrm{dom}(s) \text{ and } h_c \in \mathsf{chunks}^+(s,h) \\ & \text{with } (s, h_c) \models^{\mathsf{st}} x \mapsto y \\ \langle [y]^s_=, \geq 2 \rangle & \text{there are } y \in \mathrm{dom}(s) \text{ and } h_c \in \mathsf{chunks}^+(s,h) \\ & \text{with } (s, h_c) \models^{\mathsf{st}} \mathtt{ls}(x,y) \wedge \neg x \mapsto y \\ \bot, & \text{otherwise.} \end{cases}$$

Finally, we denote the sets of variables allocated in negative chunks by

$$\mathsf{alloc}^-(s,h) := \{\{[x]^s_= \mid s(x) \in \mathrm{dom}(h_c)\} \mid h_c \in \mathsf{chunks}^-(s,h)\} \setminus \{\emptyset\},$$

where (equivalence classes of) variables that are allocated in the same negative chunk are grouped together in a set.

Now we are ready to define the *induced AMS* of a model.

Definition 3.12. Let (s, h) be a model. Let $V := \mathsf{cls}_=(s)$, $E := \mathsf{edges}(s,h)$, $\rho := \mathsf{alloc}^-(s,h)$ and $\gamma := |\mathsf{chunks}^-(s,h)|$. Then $\mathsf{ams}(s,h) := \langle V, E, \rho, \gamma \rangle$ is the *induced AMS* of (s, h).

Example 3.13. The induced AMS of the model (s, h) from Ex. 3.7 is illustrated on the right-hand side of Fig. 5. The blue box depicts the graph (V, E) induced by the positive chunks h_1, h_2; the negative chunks that allocate variables are abstracted to the set $\rho = \{\{\{w\}, \{u\}\}, \{\{y, z\}\}\}$ (note that the variables w and u are allocated in the chunk h_3 and the aliasing variables y, z are allocated in h_4); and the garbage-chunk count is 3.

Observe that the induced AMS is indeed an AMS:

Proposition 3.14. *Let (s, h) be a model. Then* $\mathsf{ams}(s,h) \in$ **AMS**.

The reverse also holds: Every AMS is the induced AMS of at least one model; in fact, even of a model of linear size.

Lemma 3.15 (Realizability of AMS). *Let $\mathcal{A} = \langle V, E, \rho, \gamma \rangle$ be an AMS. There exists a model (s, h) with* $\mathsf{ams}(s,h) = \mathcal{A}$ *whose size is linear in the size of $\mathcal{A}$.*

Proof. For simplicity, we assume $\mathbf{Loc} = \mathbb{N}$; this allows us to add locations.

Let $n := |V|$. We fix some injective function $t \colon V \to \{1, \ldots, n\}$ from nodes to natural numbers. We set $s := \bigcup_{x \in v, v \in V} \{x \mapsto t(v)\}$ and define h as the (disjoint) union of

- $\bigcup_{E(v) = \langle v', =1 \rangle} \{t(v) \mapsto t(v')\}$
- $\bigcup_{E(v) = \langle v', \geq 2 \rangle} \{t(v) \mapsto n + t(v), n + t(v) \mapsto t(v')\}$
- $\left(\bigcup_{v \in \mathbf{r}, \mathbf{r} \in \rho} \{t(v) \mapsto 2n + t(\max(\mathbf{r}))\}\right) \cup \left(\bigcup_{\mathbf{r} \in \rho} \{2n + t(\max(\mathbf{r})) \mapsto 2n + t(\max(\mathbf{r}))\}\right)$
- $\bigcup_{l \in \{3n+1, \ldots, 3n+\gamma\}} \{l \mapsto l\}$

It is easy to verify that $\mathsf{ams}(s,h) = \mathcal{A}$ and that $|h| \in O(|\mathcal{A}|)$. □

The following lemma demonstrates that we only need the ρ and γ components in order to be able to deal with negation and/or the magic wand:

Lemma 3.16 (Models of Positive Formulas Abstract to Garbage-free AMS). *Let (s,h) be a model. If $(s,h) \models \varphi$ for a positive formula φ, then $\mathsf{ams}(s,h)$ is garbage-free.*

Proof. The lemma can be proved by a straight-forward induction on φ, using that every heap fully decomposes into its chunks. □

We abstract SL formulas by the set of AMS of their models:

Definition 3.17. Let s be a stack. The **SL** *abstraction* w.r.t. s, $\alpha_s : \mathbf{SL} \to 2^{\mathbf{AMS}}$, is given by

$$\alpha_s(\varphi) := \{\mathsf{ams}(s,h) \mid h \in \mathbf{H}, \text{ and } (s,h) \models^{\mathsf{st}} \varphi\}. \qquad \triangle$$

Because AMSs do not retain any information about heap locations, just about aliasing, abstractions do not differ for stacks with the same equivalence classes:

Lemma 3.18. *Let s, s' be stacks with $\mathsf{cls}_=(s) = \mathsf{cls}_=(s')$. Then $\alpha_s(\varphi) = \alpha_{s'}(\varphi)$ for all formulas φ.*

Proof. Let $\mathcal{A} \in \alpha_s(\varphi)$. Then there exists a heap h such that $\mathsf{ams}(s,h) = \mathcal{A}$ and $(s,h) \models^{\mathsf{st}} \varphi$. Let h' be such that $(s,h) \cong (s',h')$. By Lemma 2.4, $(s',h') \models^{\mathsf{st}} \varphi$. Moreover, $\mathsf{ams}(s',h') = \mathcal{A}$. Consequently, $\mathcal{A} \in \alpha_{s'}(\varphi)$. The other direction is proved analogously. □

3.3 The Refinement Theorem for SSL

The main goal of this section is to show the following *refinement theorem:*

Theorem 3.19 (Refinement Theorem). *Let φ be a formula and let (s,h_1), (s,h_2) be models with $\mathsf{ams}(s,h_1) = \mathsf{ams}(s,h_2)$. Then $(s,h_1) \models^{\mathsf{st}} \varphi$ iff $(s,h_2) \models^{\mathsf{st}} \varphi$.*

We will prove this theorem step by step, characterizing the AMS abstraction of all atomic formulas and of the composed models before proving the refinement theorem. In the remainder of this section, we fix some model (s,h).

Abstract Memory States of Atomic Formulas. The empty-heap predicate **emp** is only satisfied by the empty heap, i.e., by a heap that consists of zero chunks:

Lemma 3.20. $(s,h) \models \mathbf{emp}$ *iff* $\mathsf{ams}(s,h) = \langle \mathsf{cls}_=(s), \emptyset, \emptyset, 0\rangle$

Proof. $(s,h) \models^{\mathsf{st}} \mathbf{emp}$ iff $h = \emptyset$ iff $\mathsf{chunks}(s,h) = \emptyset$ iff $\mathsf{ams}(s,h) = \langle \mathsf{cls}_=(s), \emptyset, \emptyset, 0\rangle$. □

Lemma 3.21. *(1)* $(s,h) \models x = y$ *iff* $\mathsf{ams}(s,h) = \langle \mathsf{cls}_=(s), \emptyset, \emptyset, 0\rangle$ *and* $[x]^s_= = [y]^s_=$.
(2) $(s,h) \models x \neq y$ *iff* $\mathsf{ams}(s,h) = \langle \mathsf{cls}_=(s), \emptyset, \emptyset, 0\rangle$ *and* $[x]^s_= \neq [y]^s_=$.

Proof. We only show the first claim, as the proof of the second claim is completely analogous. $(s,h) \models x = y$ iff ($s(x) = s(y)$ and $h = \emptyset$) iff ($[x]^s_= = [y]^s_=$ and $(s,h) \models \mathbf{emp}$) iff, by Lemma 3.20, ($[x]^s_= = [y]^s_=$ and $\mathsf{ams}(s,h) = \langle \mathsf{cls}_=(s), \emptyset, \emptyset, 0\rangle$). □

Models of points-to assertions consist of a single positive chunk of size 1:

Lemma 3.22. *Let $E = \{[x]^s_= \mapsto \langle [y]^s_=, = 1\rangle\}$. $(s,h) \models x \mapsto y$ iff $\mathsf{ams}(s,h) = \langle \mathsf{cls}_=(s), E, \emptyset, 0\rangle$.*

Proof. If $(s,h) \models^{\mathsf{st}} x \mapsto y$ then $h = \{s(x) \mapsto s(y)\}$. In particular, it then holds that h is a positive chunk. Consequently, $\mathsf{edges}(s,h) = E$. It follows that $\mathsf{ams}(s,h) = \langle \mathsf{cls}_=(s), E, \emptyset, 0\rangle$.

Conversely, assume $\mathsf{ams}(s,h) = \langle \mathsf{cls}_=(s), E, \emptyset, 0\rangle$. Then, (s,h) consists of a single positive chunk and no negative chunks. Further, by the definition of $\mathsf{edges}(s,h)$ we have that this single positive chunk satisfies $(s,h) \models x \mapsto y$. □

Intuitively, the list segment $\mathtt{ls}(x, y)$ is satisfied by models (s, h) that consist of zero or more positive chunks, corresponding to a (possibly empty) list from some equivalence class $[x]^s_=$ to $[y]^s_=$ via (zero or more) intermediate equivalence classes $[x_1]^s_=, \ldots, [x_n]^s_=$. We will use this intuition to define abstract lists; this notion allows us to characterize the AMSs arising from abstracting lists.

Definition 3.23. Let $\mathcal{A} = \langle V, E, \rho, \gamma \rangle \in \mathbf{AMS}$ and $x, y \in \mathbf{Var}$. We say $\mathcal{A}$ is an *abstract list* w.r.t. x and y, in signs $\mathcal{A} \in \mathbf{AbstLists}(x, y)$, iff

(1) $\rho = \emptyset$ and $\gamma = 0$, and
(2) we can pick nodes $\mathbf{v}_1, \ldots, \mathbf{v}_n \in V$ and labels $\iota_1, \ldots, \iota_{n-1} \in \{=1, \geq 2\}$ such that $x \in \mathbf{v}_1$, $y \in \mathbf{v}_n$ and $E = \{\mathbf{v}_i \mapsto \langle \mathbf{v}_{i+1}, \iota_i \rangle \mid 1 \leq i < n\}$.

Lemma 3.24. $(s, h) \models \mathtt{ls}(x, y)$ *iff* $\mathsf{ams}(s, h) \in \mathbf{AbstLists}(x, y)$.

Proof. Assume $(s, h) \models \mathtt{ls}(x, y)$. By the semantics, there exist locations $\ell_0, \ldots, \ell_n$, $n \geq 1$, with $s(x) = \ell_0$, $s(y) = \ell_n$ and $h = \{\ell_0 \mapsto \ell_1, \ldots, \ell_{n-1} \mapsto \ell_n\}$. Let $j_1, \ldots, j_k$ those indices among $1, \ldots, n$ with $\ell_{j_i} \in \mathsf{img}(s)$. (In particular, $j_1 = 1$ and $j_k = n$.) Then for each j_i, the restriction of h to $\ell_{j_i}, \ell_{j_i+1}, \ldots, \ell_{j_{i+1}-1}$ is a positive chunk that either satisfies a points-to assertion or a list-segment predicate. Hence, $\mathsf{edges}(s, h)(s^{-1}(\ell_{j_i})) = \left\langle s^{-1}(\ell_{j_{i+1}}), \iota_i \right\rangle$ for all $1 \leq i < k$, for some $\iota_i \in \{=1, \geq 2\}$. Thus, $\mathsf{ams}(s, h) \in \mathbf{AbstLists}(x, y)$.

Assume $\mathsf{ams}(s, h) \in \mathbf{AbstLists}(x, y)$. Then, there are equivalence classes $[x_1]^s_=, \ldots, [x_n]^s_= \in \mathsf{cls}_=(s)$ and labels $\iota_1, \ldots, \iota_{n-1} \in \{=1, \geq 2\}$ such that $x \in [x_1]^s_=$, $y \in [x_n]^s_=$ and $\mathsf{edges}(s, h) = \{[x_i]^s_= \mapsto \langle [x_{i+1}]^s_=, \iota_i \rangle \mid 1 \leq i < n\}$. By the definition of $\mathsf{edges}(s, h)$, we have that there are positive chunks h_i of h such that $(s, h_i) \models^{\mathsf{st}} x_i \mapsto x_{i+1}$ or $(s, h_i) \models^{\mathsf{st}} \mathtt{ls}(x_i, x_{i+1})$. In particular, we have $h_i = \{\ell_{i,1} \mapsto \ell_{i,2}, \ldots, \ell_{i,k_i-1} \mapsto \ell_{i,k_i}\}$, $s(x_i) = \ell_{i,1}$ and $s(x_{i+1}) = \ell_{i,k_i}$ for some locations $\ell_{i,j}$. Because h does not have negative chunks, we get that h fully decomposes into its positive chunks. Hence, the locations $\ell_{i,j}$ witness that $(s, h) \models \mathtt{ls}(x, y)$. □

Abstract Memory States of Models composed by the Union Operator. Our next goal is to lift the union operator $\uplus^s$ to the abstract domain **AMS**. We will define an operator $\bullet$ with the following property:

$$\text{if } h_1 \uplus^s h_2 \neq \bot \text{ then } \mathsf{ams}(s, h_1 \uplus^s h_2) = \mathsf{ams}(s, h_1) \bullet \mathsf{ams}(s, h_2).$$

AMS composition is a partial operation defined only on *compatible* AMS. Compatibility enforces (1) that the AMSs were obtained for equivalent stacks (i.e., for stacks s, s' with $\mathsf{cls}_=(s) = \mathsf{cls}_=(s')$), and (2) that there is no double allocation.

Definition 3.25 (Compatibility of AMSs). AMSs $\mathcal{A}_1 = \langle V_1, E_1, \rho_1, \gamma_1 \rangle$ and $\mathcal{A}_2 = \langle V_2, E_2, \rho_2, \gamma_2 \rangle$ are *compatible* iff (1) $V_1 = V_2$ and (2) $\mathbf{alloc}(\mathcal{A}_1) \cap \mathbf{alloc}(\mathcal{A}_2) = \emptyset$.

Note that if $h_1 \uplus^s h_2$ is defined, then $\mathsf{ams}(s, h_1)$ and $\mathsf{ams}(s, h_2)$ are compatible. The converse is not true, because $\mathsf{ams}(s, h_1)$ and $\mathsf{ams}(s, h_2)$ may be compatible even if $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) \neq \emptyset$.

AMS composition is defined in a point-wise manner on compatible AMSs and undefined otherwise.

Definition 3.26 (AMS composition). Let $\mathcal{A}_1 = \langle V_1, E_1, \rho_1, \gamma_1 \rangle$ and $\mathcal{A}_2 = \langle V_2, E_2, \rho_2, \gamma_2 \rangle$ be two AMS. The *composition* of $\mathcal{A}_1, \mathcal{A}_2$ is then given by

$$\mathcal{A}_1 \bullet \mathcal{A}_2 := \begin{cases} \langle V_1, E_1 \cup E_2, \rho_1 \cup \rho_2, \gamma_1 + \gamma_2 \rangle, & \text{if } \mathcal{A}_1, \mathcal{A}_2 \text{ compatible} \\ \bot, & \text{otherwise.} \end{cases}$$

Lemma 3.27. *Let s be a stack and let h_1, h_2 be heaps. If $h_1 \uplus^s h_2 \neq \bot$ then* $\mathsf{ams}(s, h_1) \bullet \mathsf{ams}(s, h_2) \neq \bot$.

Proof. Since the same stack s underlies both abstractions, we have $V_1 = V_2$. Furthermore, $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset$ implies that $\mathbf{alloc}(\mathcal{A}_1) \cap \mathbf{alloc}(\mathcal{A}_2) = \emptyset$. □

We next show that $\mathsf{ams}(s, h_1 \uplus^s h_2) = \mathsf{ams}(s, h_1) \bullet \mathsf{ams}(s, h_2)$ whenever $h_1 \uplus^s h_2$ is defined:

Lemma 3.28 (Homomorphism of composition). *Let $(s, h_1), (s, h_2)$ be models with $h_1 \uplus^s h_2 \neq \bot$. Then,* $\mathsf{ams}(s, h_1 \uplus^s h_2) = \mathsf{ams}(s, h_1) \bullet \mathsf{ams}(s, h_2)$.

Proof. The result follows easily from the observation that

$$\mathsf{chunks}(s, h_1 \uplus^s h_2) = \mathsf{chunks}(s, h_1) \cup \mathsf{chunks}(s, h_2),$$

which, in turn, is an immediate consequence of Proposition 3.6. □

To show the refinement theorem, we need one additional property of AMS composition. If an AMS $\mathcal{A}$ of a model (s, h) can be decomposed into two smaller AMS $\mathcal{A} = \mathcal{A}_1 \bullet \mathcal{A}_2$, it is also possible to decompose the heap h into smaller heaps h_1, h_2 with $\mathsf{ams}(s, h_i) = \mathcal{A}_i$:

Lemma 3.29 (Decomposability of AMS). *Let* $\mathsf{ams}(s, h) = \mathcal{A}_1 \bullet \mathcal{A}_2$. *There exist h_1, h_2 with $h = h_1 \uplus^s h_2$,* $\mathsf{ams}(s, h_1) = \mathcal{A}_1$ *and* $\mathsf{ams}(s, h_2) = \mathcal{A}_2$.

Proof. It can be verified from the definition of AMS and the definition of the composition operator $\bullet$ that the following property holds: Let $h_c \in \mathsf{chunks}(s, h)$ be a chunk. Then, either there exists an $\mathcal{A}_1'$ such that $\mathcal{A}_1 = \mathsf{ams}(s, h_c) \bullet \mathcal{A}_1'$ or there exists an $\mathcal{A}_2'$ such that $\mathcal{A}_2 = \mathsf{ams}(s, h_c) \bullet \mathcal{A}_2'$.
The claim can then by proven an induction of the number of chunks $|\mathsf{chunks}(s, h)|$. □

These results suffice to prove the Refinement Theorem stated at the beginning of this section; see the appendix for a proof.

(Theorem 3.19.) Let $\mathcal{A} = \langle V, E, \rho, \gamma \rangle$ be the AMS with $\mathsf{ams}(s, h_1) = \mathcal{A} = \mathsf{ams}(s, h_2)$. We proceed by induction on the structure of φ. We only prove that $(s, h_1) \models^{\mathsf{st}} \varphi$ implies that $(s, h_2) \models^{\mathsf{st}} \varphi$, as the other direction is completely analogous.

Assume that the claim holds for all subformulas of φ and assume that $(s, h_1) \models^{\mathsf{st}} \varphi$.

Case emp, $x = y$, $x \neq y$, $x \mapsto y$, $\mathtt{ls}(x, y)$. Immediate consequence of Lemmas 3.20, 3.21, 3.22 and 3.24.

Case $\varphi_1 * \varphi_2$. By the semantics of $*$, there exist $h_{1,1}, h_{1,2}$ with $h_1 = h_{1,1} \uplus^s h_{1,2}$, $(s, h_{1,1}) \models^{\mathsf{st}} \varphi_1$, and $(s, h_{1,2}) \models^{\mathsf{st}} \varphi_2$. Let $\mathcal{A}_1 := \mathsf{ams}(s, h_{1,1})$ and $\mathcal{A}_2 := \mathsf{ams}(s, h_{1,2})$. By Lemma 3.28, $\mathsf{ams}(s, h_1) = \mathcal{A}_1 \bullet \mathcal{A}_2 = \mathsf{ams}(s, h_2)$. We can thus apply Lemma 3.29 to $\mathsf{ams}(s, h_2)$, $\mathcal{A}_1$, and $\mathcal{A}_2$ to obtain heaps $h_{2,1}, h_{2,2}$ with $h_2 = h_{2,1} \uplus^s h_{2,2}$, $\mathsf{ams}(s, h_{2,1}) = \mathcal{A}_1$ and $\mathsf{ams}(s, h_{2,2}) = \mathcal{A}_2$. We can now apply the induction hypotheses for $1 \leq i \leq 2$, φ_i, $h_{1,i}$ and $h_{2,i}$, and obtain that $(s, h_{2,i}) \models^{\mathsf{st}} \varphi_i$. By the semantics of $*$, we then have $(s, h_2) = (s, h_{2,1} \uplus^s h_{2,2}) \models^{\mathsf{st}} \varphi_1 * \varphi_2$.

Case $\varphi_1 \mathrel{-\!\!\circledast} \varphi_2$. Since $(s, h_1) \models^{\mathsf{st}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2$, there exists a heap h_0 with $(s, h_0) \models^{\mathsf{st}} \varphi_1$ and $(s, h_1 \uplus^s h_0) \models^{\mathsf{st}} \varphi_2$. We can assume w.l.o.g. that $h_2 \uplus^s h_0 \neq \bot$—if this is not the case, simply replace h_0 with a heap h_0' with $(s, h_0) \cong (s, h_0')$, $h_1 \uplus^s h_0' \neq \bot$ and $h_2 \uplus^s h_0' \neq \bot$; then, $(s, h_1 \uplus^s h_0') \models^{\mathsf{st}} \varphi_2$ by Lemma 2.4. We have that $\mathsf{ams}(s, h_1 \uplus^s h_0) = \mathsf{ams}(s, h_1) \bullet \mathsf{ams}(s, h_0) = \mathsf{ams}(s, h_2) \bullet \mathsf{ams}(s, h_0) = \mathsf{ams}(s, h_2 \uplus^s h_0)$ (by assumption and Lemma 3.28). It therefore follows from the induction hypothesis for φ_2, $(s, h_1 \uplus^s h_0)$, and $(s, h_2 \uplus^s h_0)$ that $(s, h_2 \uplus^s h_0) \models^{\mathsf{st}} \varphi_2$. Thus, $(s, h_2) \models^{\mathsf{st}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2$.

Case $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$. By the semantics of $\wedge$ resp. $\vee$, we have $(s, h_1) \models^{\mathsf{st}} \varphi_1$ and/or $(s, h_1) \models^{\mathsf{st}} \varphi_2$. We apply the induction hypotheses for φ_1 and φ_2 to obtain $(s, h_2) \models^{\mathsf{st}} \varphi_1$ and/or $(s, h_2) \models^{\mathsf{st}} \varphi_2$. By the semantics of $\wedge$ resp. $\vee$, we then have $(s, h_2) \models^{\mathsf{st}} \varphi_1 \wedge \varphi_2$ resp. $(s, h_2) \models^{\mathsf{st}} \varphi_1 \vee \varphi_2$.

Case $\neg\varphi_1$. By the semantics of $\neg$, we have $(s, h_1) \not\models^{\mathsf{st}} \varphi_1$. By the induction hypothesis for φ_1 we then obtain $(s, h_2) \not\models^{\mathsf{st}} \varphi_1$. By the semantics of $\neg$, we have $(s, h_2) \models^{\mathsf{st}} \neg\varphi_1$.

Corollary 3.30. *Let (s, h) be a model and φ be a formula. $(s, h) \models^{\mathsf{st}} \varphi$ iff* $\mathsf{ams}(s, h) \in \alpha_s(\varphi)$.

Proof. Assume $\mathcal{A} := \mathsf{ams}(s,h) \in \alpha_s(\varphi)$. By definition of α_s there is a model (s,h') with $(s,h') \models^{\mathsf{st}} \varphi$ and $\mathsf{ams}(s,h') = \mathcal{A}$. By applying Theorem 3.19 to φ, (s,h) and (s,h'), we then get that $(s,h) \models^{\mathsf{st}} \varphi$. □

3.4 Recursive Equations for Abstract Memory States

In this section, we derive recursive equations that reduce the set of AMS $\alpha_s(\varphi)$ for arbitrary compound formulas to the set of AMS of the constituent formulas of φ. In the next sections, we will show that we can actually evaluate these equations, thus obtaining an algorithm for computing the abstraction of arbitrary formulas.

Lemma 3.31. $\alpha_s(\varphi_1 \wedge \varphi_2) = \alpha_s(\varphi_1) \cap \alpha_s(\varphi_2)$.

Proof. Let (s,h) be a model. $(s,h) \models^{\mathsf{st}} \varphi_1 \wedge \varphi_2$ iff $(s,h) \models^{\mathsf{st}} \varphi_1$ and $(s,h) \models^{\mathsf{st}} \varphi_2$ iff $\mathsf{ams}(s,h) \in \alpha_s(\varphi_1)$ and $\mathsf{ams}(s,h) \in \alpha_s(\varphi_2)$ iff $\mathsf{ams}(s,h) \in \alpha_s(\varphi_1) \cap \alpha_s(\varphi_2)$. □

Lemma 3.32. $\alpha_s(\varphi_1 \vee \varphi_2) = \alpha_s(\varphi_1) \cup \alpha_s(\varphi_2)$.

Proof. Let (s,h) be a model. $(s,h) \models^{\mathsf{st}} \varphi_1 \vee \varphi_2$ iff $((s,h) \models^{\mathsf{st}} \varphi_1$ or $(s,h) \models^{\mathsf{st}} \varphi_2)$ iff $(\mathsf{ams}(s,h) \in \alpha_s(\varphi_1)$ or $\mathsf{ams}(s,h) \in \alpha_s(\varphi_2))$ iff $\mathsf{ams}(s,h) \in \alpha_s(\varphi_1) \cup \alpha_s(\varphi_2)$. □

Lemma 3.33. $\alpha_s(\neg\varphi_1) = \{\mathsf{ams}(s,h) \mid h \in \mathbf{H}\} \setminus \alpha_s(\varphi_1)$.

Proof. Let (s,h) be a model. $(s,h) \models^{\mathsf{st}} \neg\varphi_1$ iff it is not the case that $(s,h) \models^{\mathsf{st}} \varphi_1$ iff it is not the case that $\mathsf{ams}(s,h) \in \alpha_s(\varphi_1)$ iff it is the case that $\mathsf{ams}(s,h) \in \{\mathsf{ams}(s,h) \mid h \in \mathbf{H}\} \setminus \alpha_s(\varphi_1)$. □

The Separating Conjunction. In Section 3.3, we defined the composition operation, $\bullet$, on pairs of AMS. We now lift this operation to sets of AMS $\mathbf{A}_1, \mathbf{A}_2$:

$$\mathbf{A}_1 \bullet \mathbf{A}_2 := \{\mathcal{A}_1 \bullet \mathcal{A}_2 \mid \mathcal{A}_1 \in \mathbf{A}_1, \mathcal{A}_2 \in \mathbf{A}_2, \mathcal{A}_1 \bullet \mathcal{A}_2 \neq \bot\}\,.$$

Lemma 3.28 implies that α_s is a homomorphism from formulas and $*$ to sets of AMS and $\bullet$:

Lemma 3.34. *For all* φ_1, φ_2, $\alpha_s(\varphi_1 * \varphi_2) = \alpha_s(\varphi_1) \bullet \alpha_s(\varphi_2)$.

Proof. See appendix. □

(Lemma 3.34.) Let $\mathcal{A} \in \alpha_s(\varphi_1 * \varphi_2)$. There then exists a heap h such that $(s,h) \models^{\mathsf{st}} \varphi_1 * \varphi_2$ and $\mathsf{ams}(s,h) = \mathcal{A}$. By the semantics of $*$, we can split h into $h_1 \uplus^s h_2$ with $(s,h_i) \models^{\mathsf{st}} \varphi_i$ (and thus $\mathsf{ams}(s,h_i) \in \alpha_s(\varphi_i)$). By Lemma 3.28, $\mathcal{A} = \mathsf{ams}(s,h_1) \bullet \mathsf{ams}(s,h_2)$ for h_1, h_2 as above. Consequently, $\mathcal{A} \in \alpha_s(\varphi_1) \bullet \alpha_s(\varphi_2)$ by definition of $\bullet$.

Conversely, let $\mathcal{A} \in \alpha_s(\varphi_1) \bullet \alpha_s(\varphi_2)$. By definition of $\bullet$, there then exist $\mathcal{A}_i \in \alpha_s(\varphi_i)$ such that $\mathcal{A} = \mathcal{A}_1 \bullet \mathcal{A}_2$. Let h_1, h_2 be witnesses of that, i.e., $(s,h_i) \models^{\mathsf{st}} \varphi_i$ with $\mathsf{ams}(s,h_i) = \mathcal{A}_i$. Assume w.l.o.g. that $h_1 \uplus^s h_2 \neq \bot$. (Otherwise, replace h_2 with an h_2' such that $(s,h_2) \cong (s,h_2')$ and $h_1 \uplus^s h_2' \neq \bot$; by Lemma 2.4 we then have $(s,h_2') \models^{\mathsf{st}} \varphi_2$.) By the semantics of $*$, $(s, h_1 \uplus^s h_2) \models^{\mathsf{st}} \varphi_1 * \varphi_2$. Therefore, $\mathsf{ams}(s, h_1 \uplus^s h_2) \in \alpha_s(\varphi_1 * \varphi_2)$. By Lemma 3.28, $\mathsf{ams}(s, h_1 \uplus^s h_2) = \mathcal{A}$. The claim follows.

The septraction operator. We next define an *abstract septraction operator* $\mathbin{-\!\!\bullet}$ that relates to $\bullet$ in the same way that $\mathbin{-\!\!\circledast}$ relates to $*$. For two sets of AMS $\mathbf{A}_1, \mathbf{A}_2$ we set:

$$\mathbf{A}_1 \mathbin{-\!\!\bullet} \mathbf{A}_2 := \{\mathcal{A} \in \mathbf{AMS} \mid \text{ there exists } \mathcal{A}_1 \in \mathbf{A}_1 \text{ s.t. } \mathcal{A} \bullet \mathcal{A}_1 \in \mathbf{A}_2\}$$

Then, α_s is a homomorphism from formulas and $\mathbin{-\!\!\circledast}$ to sets of AMS and $\mathbin{-\!\!\bullet}$:

Lemma 3.35. *For all* φ_1, φ_2, $\alpha_s(\varphi_1 \mathbin{-\!\!\circledast} \varphi_2) = \alpha_s(\varphi_1) \mathbin{-\!\!\bullet} \alpha_s(\varphi_2)$.

Proof. See appendix. □

(Lemma 3.35.) Let $\mathcal{A} \in \alpha_s(\varphi_1 \mathrel{-\!\!\circledast} \varphi_2)$. Then there exists a model (s,h) with $\mathsf{ams}(s,h) = \mathcal{A}$ and $(s,h) \overset{\text{st}}{\models} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2$. Consequently, there exists a heap h_1 such that $h \uplus^s h_1 \neq \bot$, $(s,h_1) \overset{\text{st}}{\models} \varphi_1$ and $(s, h \uplus^s h_1) \overset{\text{st}}{\models} \varphi_2$. By definition of α_s, we then have $\mathsf{ams}(s,h_1) \in \alpha_s(\varphi_1)$ and $\mathsf{ams}(s, h \uplus^s h_1) \in \alpha_s(\varphi_2)$. By Lemma 3.28, $\mathsf{ams}(s, h \uplus^s h_1) = \mathsf{ams}(s,h) \bullet \mathsf{ams}(s,h_1)$. In other words, we have for $\mathcal{A} = \mathsf{ams}(s,h)$ and $\mathcal{A}_1 := \mathsf{ams}(s,h_1)$ that $\mathcal{A}_1 \in \alpha_s(\varphi_1)$ and $\mathcal{A} \bullet \mathcal{A}_1 \in \alpha_s(\varphi_2)$. By definition of $\mathrel{-\!\!\bullet}$, we hence have $\mathcal{A} \in \alpha_s(\varphi_1) \mathrel{-\!\!\bullet} \alpha_s(\varphi_2)$.

Conversely, let $\mathcal{A} \in \alpha_s(\varphi_1) \mathrel{-\!\!\bullet} \alpha_s(\varphi_2)$. Then there exists an $\mathcal{A}_1 \in \alpha_s(\varphi_1)$ such that $\mathcal{A} \bullet \mathcal{A}_1 \in \alpha_s(\varphi_2)$. Let h, h_1 be heaps with $\mathsf{ams}(s,h) = \mathcal{A}$, $\mathsf{ams}(s,h_1) = \mathcal{A}_1$ and $(s,h_1) \overset{\text{st}}{\models} \varphi_1$. Assume w.l.o.g. that $h \uplus^s h_1 \neq \bot$. (Otherwise, replace h_1 with an h_1' such that $(s,h_1) \cong (s,h_1')$ and $h \uplus^s h_1' \neq \bot$; by Lemma 2.4 we then have $(s,h_1') \overset{\text{st}}{\models} \varphi_1$.)

By Lemma 3.28, we then have $\mathsf{ams}(s, h \uplus^s h_1) = \mathcal{A} \bullet \mathcal{A}_1$. By Cor. 3.30, this allows us to conclude that $(s, h \uplus^s h_1) \overset{\text{st}}{\models} \varphi_2$. Consequently, $(s,h) \overset{\text{st}}{\models} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2$, implying $\mathcal{A} \in \alpha_s(\varphi_1 \mathrel{-\!\!\circledast} \varphi_2)$.

3.5 Refining the Refinement Theorem: Bounding Garbage

Even though we have now characterized the set $\alpha_s(\varphi)$ for every formula φ, we do not yet have a way to implement AMS computation: While $\alpha_s(\varphi)$ is finite if φ is a spatial atom, the set is infinite in general; see the cases $\alpha_s(\neg\varphi)$ and $\alpha_s(\varphi_1 \mathrel{-\!\!\circledast} \varphi_2)$. However, we note that for a fixed stack s only the garbage-chunk count γ of an AMS $\langle V, E, \rho, \gamma \rangle \in \alpha_s(\varphi)$ can be of arbitrary size, while the size of the nodes V, the edges E and the negative-allocation constraint ρ is bounded by $|s|$. Fortunately, to decide the satisfiability of any fixed formula φ, it is *not* necessary to keep track of arbitrarily large garbage-chunk counts.

We introduce the *chunk size* $\lceil\varphi\rceil$ of a formula φ, which provides an upper bound on the number of *negative chunks* that may be necessary to satisfy and/or falsify the formula; $\lceil\varphi\rceil$ is defined as follows:

- $\lceil \mathbf{emp} \rceil = \lceil x \mapsto y \rceil = \lceil \mathtt{ls}(x,y) \rceil = \lceil x = y \rceil = \lceil x \neq y \rceil := 0$
- $\lceil \varphi * \psi \rceil := \lceil \varphi \rceil + \lceil \psi \rceil$
- $\lceil \varphi \mathrel{-\!\!\circledast} \psi \rceil := \lceil \psi \rceil$
- $\lceil \varphi \vee \psi \rceil := \max(\lceil \varphi \rceil, \lceil \psi \rceil)$
- $\lceil \varphi \wedge \psi \rceil := \begin{cases} 0, & \text{if } \lceil\varphi\rceil = 0 \text{ or } \lceil\psi\rceil = 0 \\ \max(\lceil \varphi \rceil, \lceil \psi \rceil), & \text{otherwise.} \end{cases}$
- $\lceil \neg\varphi \rceil := \max\{1, \lceil\varphi\rceil\}$.

Observe that $\lceil\varphi\rceil \leq |\varphi|$ for all φ. Intuitively, the chunk bound $\lceil\varphi\rceil$ of a formula φ establishes two pieces of information: (1) For $\lceil\varphi\rceil = 0$, we have that every model of φ does not contain negative chunks. (2) For $\lceil\varphi\rceil \geq 1$, we have that if there is a model of φ then there is also a model with at most $\lceil\varphi\rceil$ negative chunks, and for every model with at least $\lceil\varphi\rceil$ negative chunks we can add an arbitrary number of negative chunks (without allocated variables) and still satisfy φ. We now formally state these two facts:

Lemma 3.36. *Let φ be a formula with $\lceil\varphi\rceil = 0$ and let (s,h) be a model of φ. Then, the AMS of all models of φ have a garbage-chunk count of* 0.

Proof. See appendix. □

(Lemma 3.36.) We proceed by structural induction on φ. Let (s,h) be a stack-heap pair with $(s,h) \overset{\text{st}}{\models} \varphi$.
By Lemmas 3.20, 3.21, 3.22 and 3.24 we have that the AMS of all models of φ have a garbage-chunk count of 0.
Let h_1, h_2 be such that $h = h_1 \uplus^s h_2$, $(s,h_1) \overset{\text{st}}{\models} \varphi_1$, and $(s,h_2) \overset{\text{st}}{\models} \varphi_2$. By the definition of the garbage-chunk count, we must have $\lceil\varphi_1\rceil = \lceil\varphi_2\rceil = 0$. Hence, the claim follows from the induction assumption.
Let h_0 be such that $(s,h_0) \overset{\text{st}}{\models} \varphi_1$ and $(s, h \uplus^s h_0) \overset{\text{st}}{\models} \varphi_2$. Because of $0 = \lceil \varphi_1 \mathrel{-\!\!\circledast} \varphi_2 \rceil = \lceil \varphi_2 \rceil$, it thus follows from the induction hypothesis that $(s, h \uplus^s h_0) \overset{\text{st}}{\models} \varphi_2$ does not contain negative chunks. This implies, that (s,h) does not contain negative chunks.

We then have $(s, h_1) \models^{\mathsf{st}} \varphi_1$ or $(s, h_1) \models^{\mathsf{st}} \varphi_2$. The claim then follows from the induction assumption because we must have $\lceil \varphi_1 \rceil = \lceil \varphi_2 \rceil = 0$.
We then have $(s, h_1) \models^{\mathsf{st}} \varphi_1$ and $(s, h_1) \models^{\mathsf{st}} \varphi_2$. The claim then follows from the induction assumption because we have $\lceil \varphi_1 \rceil = 0$ or $\lceil \varphi_2 \rceil = 0$.
Because of the assumption $\lceil \varphi \rceil = 0$, this case is not possible.

For stating the second fact, we generalize the refinement theorem, Theorem 3.19, to models whose AMS differ in their garbage-chunk count, provided both garbage-chunk counts exceed the non-zero chunk size of the formula:

Theorem 3.37 (Refined Refinement Theorem). *Let φ be a formula with $\lceil \varphi \rceil = k \geq 1$. Let $m \geq k$, $n \geq k$ and let $(s, h_1), (s, h_2)$ be models with $\mathsf{ams}(s, h_1) = \langle V, E, \rho, m \rangle$, $\mathsf{ams}(s, h_2) = \langle V, E, \rho, n \rangle$. Then, $(s, h_1) \models^{\mathsf{st}} \varphi$ iff $(s, h_2) \models^{\mathsf{st}} \varphi$.*

Proof. See appendix. □

(Theorem 3.37.) We proceed by structural induction on φ. We only prove that $(s, h_1) \models^{\mathsf{st}} \varphi$ implies $(s, h_2) \models^{\mathsf{st}} \varphi$, as the proof of the other direction is very similar.

Case emp, $x = y$, $x \neq y$, $x \mapsto y$, $\mathtt{ls}(x, y)$. By Lemmas 3.20, 3.21, 3.22 and 3.24 we have that the AMS of all models of φ have a garbage-chunk count of 0. Thus, $(s, h_1) \not\models^{\mathsf{st}} \varphi$ and $(s, h_2) \not\models^{\mathsf{st}} \varphi$.

Case $\varphi_1 * \varphi_2$. Assume $(s, h_1) \models^{\mathsf{st}} \varphi_1 * \varphi_2$. Let $h_{1,1}, h_{1,2}$ be such that $h_1 = h_{1,1} \uplus^s h_{1,2}$, $(s, h_{1,1}) \models^{\mathsf{st}} \varphi_1$, and $(s, h_{1,2}) \models^{\mathsf{st}} \varphi_2$. Let $\mathcal{A}_1 = \langle V_1, E_1, \rho_1, m_1 \rangle := \mathsf{ams}(h_{1,1})$ and $\mathcal{A}_2 = \langle V_2, E_2, \rho_2, m_2 \rangle := \mathsf{ams}(h_{1,2})$. Since $k = \lceil \varphi_1 \rceil + \lceil \varphi_2 \rceil$, it follows that, either $m_1 \geq \lceil \varphi_1 \rceil$ or $m_2 \geq \lceil \varphi_2 \rceil$ (or both). We can assume w.l.o.g. that $m_1 \geq \lceil \varphi_1 \rceil$. We set $\mathcal{A}'_1 := \langle V_1, E_1, \rho_1, n - \min\{\lceil \varphi_2 \rceil, m_2\} \rangle$ and $\mathcal{A}'_2 := \langle V_2, E_2, \rho_2, \min\{\lceil \varphi_2 \rceil, m_2\} \rangle$. Observe that $\mathsf{ams}(s, h_2) = \mathcal{A}'_1 \bullet \mathcal{A}'_2$. There thus exist by Lemma 3.29 heaps $h_{2,1}, h_{2,2}$ such that $(s, h_2) = h_{2,1} \uplus^s h_{2,2}$, $\mathsf{ams}(s, h_{2,1}) = \mathcal{A}'_1$ and $\mathsf{ams}(s, h_{2,1}) = \mathcal{A}'_2$. As both $m_1 \geq \lceil \varphi_1 \rceil$ and $n - \min\{\lceil \varphi_2 \rceil, m_2\} \geq k - \min\{\lceil \varphi_2 \rceil, m_2\} \geq \lceil \varphi_1 \rceil$, we have by the induction hypothesis for φ_1 that $(s, h_{2,1}) \models^{\mathsf{st}} \varphi_1$. Additionally, we have $h_{2,2} \models^{\mathsf{st}} \varphi_2$ by Theorem 3.19 (for $m_2 < \lceil \varphi_2 \rceil$) or by the induction hypothesis (for $m_2 \geq \lceil \varphi_2 \rceil$). Consequently, $(s, h_2) \models^{\mathsf{st}} \varphi_1 * \varphi_2$.

Case $\varphi_1 \mathrel{-\!\!\circledast} \varphi_2$. Assume $(s, h_1) \models^{\mathsf{st}} \varphi_1 \mathrel{-\!\!\circledast} \varphi_2$. Let h_0 be such that $(s, h_0) \models^{\mathsf{st}} \varphi_1$ and $(s, h_1 \uplus^s h_0) \models^{\mathsf{st}} \varphi_2$. We can assume w.l.o.g. that $h_2 \uplus^s h_0 \neq \bot$—if this is not the case, simply replace h_0 with a heap h'_0 with $(s, h_0) \cong (s, h'_0)$, $h_1 \uplus^s h'_0 \neq \bot$ and $h_2 \uplus^s h'_0 \neq \bot$; then, $(s, h_1 \uplus^s h'_0) \models^{\mathsf{st}} \varphi_2$ by Lemma 2.4. We set $\mathcal{A}_2 = \mathsf{ams}(s, h_1 \uplus^s h_0)$ and $\mathcal{A}'_2 = \mathsf{ams}(s, h_2 \uplus^s h_0)$. By Lemma 3.28 we have $\mathsf{ams}(s, h_1 \uplus^s h_0) = \mathsf{ams}(s, h_1) \bullet \mathsf{ams}(s, h_0)$ and $\mathsf{ams}(s, h_2 \uplus^s h_0) = \mathsf{ams}(s, h_2) \bullet \mathsf{ams}(s, h_0)$. Hence, $\mathcal{A}_2 = \langle V_2, E_2, \rho_2, m' \rangle$ and $\mathcal{A}'_2 = \langle V_2, E_2, \rho_2, n' \rangle$ for some V_2, E_2, ρ_2 and $m', n' \geq k = \lceil \varphi_1 \mathrel{-\!\!\circledast} \varphi_2 \rceil = \lceil \varphi_2 \rceil$. It thus follows from the induction hypothesis for φ_2 that $(s, h_0 \uplus^s h_2) \models^{\mathsf{st}} \varphi_2$.

Case $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$. We then have $(s, h_1) \models^{\mathsf{st}} \varphi_1$ and/or $(s, h_1) \models^{\mathsf{st}} \varphi_2$. By definition of $\lceil \varphi_1 \wedge \varphi_2 \rceil$ resp. $\lceil \varphi_1 \vee \varphi_2 \rceil$, it follows that $n, m \geq \max(\lceil \varphi_1 \rceil, \lceil \varphi_2 \rceil) \geq \lceil \varphi_i \rceil$. We therefore conclude from the induction hypothesis that $(s, h_2) \models^{\mathsf{st}} \varphi_1$ and/or $(s, h_2) \models^{\mathsf{st}} \varphi_2$. Thus, $(s, h_2) \models^{\mathsf{st}} \varphi_1 \wedge \varphi_2$ resp. $(s, h_2) \models^{\mathsf{st}} \varphi_1 \vee \varphi_2$.

Case $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$. We then have $(s, h_1) \models^{\mathsf{st}} \varphi_1$ and/or $(s, h_1) \models^{\mathsf{st}} \varphi_2$. By definition of $\lceil \varphi_1 \wedge \varphi_2 \rceil$ resp. $\lceil \varphi_1 \vee \varphi_2 \rceil$, it follows that $n, m \geq \max(\lceil \varphi_1 \rceil, \lceil \varphi_2 \rceil) \geq \lceil \varphi_i \rceil$. We therefore conclude from the induction hypothesis that $(s, h_2) \models^{\mathsf{st}} \varphi_1$ and/or $(s, h_2) \models^{\mathsf{st}} \varphi_2$. Thus, $(s, h_2) \models^{\mathsf{st}} \varphi_1 \wedge \varphi_2$ resp. $(s, h_2) \models^{\mathsf{st}} \varphi_1 \vee \varphi_2$.

Case $\neg \varphi_1$. Assume $(s, h_1) \models^{\mathsf{st}} \neg \varphi_1$. Consequently, $(s, h_1) \not\models^{\mathsf{st}} \varphi_1$. Since $m, n \geq \lceil \neg \varphi_1 \rceil = \lceil \varphi_1 \rceil$, it follows by induction that $(s, h_2) \not\models^{\mathsf{st}} \varphi_1$. Then, $(s, h_2) \models^{\mathsf{st}} \neg \varphi_1$.

This implies that φ is satisfiable over stack s iff φ is satisfiable by a heap that contains at most $\lceil \varphi \rceil$ negative chunks:

Corollary 3.38. *Let φ be an formula with $\lceil \varphi \rceil = k$. Then φ is satisfiable over stack s iff there exists a heap h such that (1) $\mathsf{ams}(s, h) = (V, E, \rho, \gamma)$ for some $\gamma \leq k$ and (2) $(s, h) \models^{\mathsf{st}} \varphi$.*

$$
\begin{aligned}
\mathrm{abst}_s(\mathbf{emp}) &:= \{\langle \mathrm{cls}_=(s), \emptyset, \emptyset, 0\rangle\} \\
\mathrm{abst}_s(x = y) &:= \text{if } s(x) = s(y) \text{ then } \{\langle \mathrm{cls}_=(s), \emptyset, \emptyset, 0\rangle\} \text{ else } \emptyset \\
\mathrm{abst}_s(x \neq y) &:= \text{if } s(x) \neq s(y) \text{ then } \{\langle \mathrm{cls}_=(s), \emptyset, \emptyset, 0\rangle\} \text{ else } \emptyset \\
\mathrm{abst}_s(x \mapsto y) &:= \{\langle \mathrm{cls}_=(s), \{[x]^s_= \mapsto [y]^s_=\}, \emptyset, 0\rangle\} \\
\mathrm{abst}_s(\mathtt{ls}(x,y)) &:= \mathbf{AbstLists}(x,y) \cap \mathbf{AMS}_{0,s} \\
\mathrm{abst}_s(\varphi_1 * \varphi_2) &:= \mathbf{AMS}_{\lceil \varphi_1 * \varphi_2 \rceil, s} \\
&\qquad \cap (\mathrm{lift}_{\lceil \varphi_1 \rceil \nearrow \lceil \varphi_1 * \varphi_2 \rceil}(\mathrm{abst}_s(\varphi_1)) \bullet \mathrm{lift}_{\lceil \varphi_2 \rceil \nearrow \lceil \varphi_1 * \varphi_2 \rceil}(\mathrm{abst}_s(\varphi_2))) \\
\mathrm{abst}_s(\varphi_1 \mathrel{-\!\!\circledast} \varphi_2) &:= \mathbf{AMS}_{\lceil \varphi_1 \mathrel{-\!\!\circledast} \varphi_2 \rceil, s} \cap (\mathrm{abst}_s(\varphi_1) \mathrel{-\!\!\bullet} \mathrm{lift}_{\lceil \varphi_2 \rceil \nearrow \lceil \varphi_1 * \varphi_2 \rceil}(\mathrm{abst}_s(\varphi_2))) \\
\mathrm{abst}_s(\varphi_1 \wedge \varphi_2) &:= \begin{cases} \mathrm{abst}_s(\varphi_1) \cap \mathrm{abst}_s(\varphi_2), & \text{if } \lceil \varphi_1 \rceil = 0 \text{ or } \lceil \varphi_2 \rceil = 0 \\ \mathrm{lift}_{\lceil \varphi_1 \rceil \nearrow \lceil \varphi_1 \wedge \varphi_2 \rceil}(\mathrm{abst}_s(\varphi_1)) \cap & \\ \qquad \mathrm{lift}_{\lceil \varphi_2 \rceil \nearrow \lceil \varphi_1 \wedge \varphi_2 \rceil}(\mathrm{abst}_s(\varphi_2)), & \text{otherwise} \end{cases} \\
\mathrm{abst}_s(\varphi_1 \vee \varphi_2) &:= \mathrm{lift}_{\lceil \varphi_1 \rceil \nearrow \lceil \varphi_1 \vee \varphi_2 \rceil}(\mathrm{abst}_s(\varphi_1)) \cup \mathrm{lift}_{\lceil \varphi_2 \rceil \nearrow \lceil \varphi_1 \vee \varphi_2 \rceil}(\mathrm{abst}_s(\varphi_2)) \\
\mathrm{abst}_s(\neg \varphi_1) &:= \mathbf{AMS}_{\lceil \neg \varphi_1 \rceil, s} \setminus \mathrm{abst}_s(\varphi_1)
\end{aligned}
$$

Fig. 6. Computing the abstract memory states of the models of φ with stack s.

Proof. Assume φ is satisfiable and let (s,h) be a model with $(s,h) \models^{\mathrm{st}} \varphi$. Let $\mathcal{A} = \langle V, E, \rho, \gamma\rangle := \mathrm{ams}(s,h)$. If $\gamma \leq k$, there is nothing to show. Otherwise, let $\mathcal{A}' := \langle V, E, \rho, k\rangle$. By Lemma 3.15, we can choose a heap h' with $\mathrm{ams}(s,h') = \mathcal{A}'$. By Theorem 3.37, $(s,h') \models^{\mathrm{st}} \varphi$. □

3.6 Deciding SSL by AMS Computation

In light of Cor. 3.38, we can decide the SSL satisfiability problem by means of a function $\mathrm{abst}_s(\varphi)$ that computes the (finite) intersection of the (possibly infinite) set $\alpha_s(\varphi)$ and the (finite) set $\mathbf{AMS}_{k,s} := \{\langle V, E, \rho, \gamma\rangle \in \mathbf{AMS} \mid V = \mathrm{cls}_=(s) \text{ and } \gamma \leq k\}$ for $k = \lceil \varphi \rceil$. We define $\mathrm{abst}_s(\varphi)$ in Fig. 6. For atomic predicates we only need to consider garbage-chunk-count 0, whereas the cases $*$, $\mathrel{-\!\!\circledast}$, $\wedge$ and $\vee$ require *lifting* the bound on the garbage-chunk count from m to $n \geq m$.

Definition 3.39. Let $m, n \in \mathbb{N}$ with $m \leq n$ and let $\mathcal{A} = \langle V, E, \rho, \gamma\rangle \in \mathbf{AMS}$. The *bound-lifting* of $\mathcal{A}$ from m to n is

$$
\mathrm{lift}_{m \nearrow n}(\mathcal{A}) := \begin{cases} \{\mathcal{A}\} & \text{if } m = 0 \text{ or } \gamma < m \\ \{\langle V, E, \rho, k\rangle \mid m \leq k \leq n\} & \text{if } m \neq 0 \text{ and } \gamma = m. \end{cases}
$$

We generalize bound-lifting to sets of AMS: $\mathrm{lift}_{m \nearrow n}(\mathbf{A}) := \bigcup_{\mathcal{A} \in \mathbf{A}} \mathrm{lift}_{m \nearrow n}(\mathcal{A})$.

As a consequence of Lemma 3.36 and Theorem 3.37, bound-lifting is sound for all $n \geq \lceil \varphi \rceil$, i.e.,

$$
\mathrm{lift}_{\lceil \varphi \rceil \nearrow n}(\alpha_s(\varphi) \cap \mathbf{AMS}_{\lceil \varphi \rceil}) = \alpha_s(\varphi) \cap \mathbf{AMS}_n.
$$

By combining this observation with the lemmas characterizing α_s (Lemmas 3.20,3.21,3.22, 3.24,3.31, 3.32,3.33, 3.34 and 3.35), we obtain the correctness of $\mathrm{abst}_s(\varphi)$:

Theorem 3.40. *Let s be a stack and φ be a formula. Then,* $\mathrm{abst}_s(\varphi) = \alpha_s(\varphi) \cap \mathbf{AMS}_{\lceil \varphi \rceil, s}$.

Proof. See appendix. □

(Theorem 3.40.) We proceed by induction on the structure of φ:

Case emp, $x = y$, $x \neq y$, $x \mapsto y$, $\mathtt{ls}(x, y)$. By Lemmas 3.20, 3.21, 3.22 and 3.24 and the observation that all models of φ have garbage-chunk count of 0.

Case $\varphi_1 * \varphi_2$. By the induction hypotheses, we have for $1 \leq i \leq 2$ that $\mathsf{abst}_s(\varphi_i) = \alpha_s(\varphi_i) \cap \mathbf{AMS}_{\lceil\varphi_i\rceil,s}$ Let $\mathbf{A}_i := \mathsf{lift}_{\lceil\varphi_1\rceil\nearrow\lceil\varphi_1*\varphi_2\rceil}(\mathsf{abst}_s(\varphi_i))$. By Theorem 3.37, it follows that $\mathbf{A}_i = \alpha_s(\varphi_i) \cap \mathbf{AMS}_{\lceil\varphi_1*\varphi_2\rceil,s}$. By Lemma 3.34, it thus follows that $\mathbf{A}_1 \bullet \mathbf{A}_2$ contains all AMS in $\alpha_s(\varphi_1 * \varphi_2)$ that can be obtained by composing AMS with a garbage-chunk count of at most $\lceil\varphi_i * \varphi_2\rceil$. Thus, in particular, (1) $\mathbf{A}_1 \bullet \mathbf{A}_2 \subseteq \alpha_s(\varphi_1 * \varphi_2)$ and (2) $\mathbf{A}_1 \bullet \mathbf{A}_2 \supseteq \alpha_s(\varphi_1 * \varphi_2) \cap \mathbf{AMS}_{\lceil\varphi_1*\varphi_2\rceil,s}$. The claim follows.

Case $\varphi_1 \mathrel{-\!\!\circledast} \varphi_2$. By the induction hypotheses, we have for $1 \leq i \leq 2$ that $\mathsf{abst}_s(\varphi_i) = \alpha_s(\varphi_i) \cap \mathbf{AMS}_{\lceil\varphi_i\rceil,s}$. Let $\mathbf{A}_2 := \mathsf{lift}_{\lceil\varphi_2\rceil\nearrow\lceil\varphi_1*\varphi_2\rceil}(\mathsf{abst}_s(\varphi_2))$. By Theorem 3.37, it follows that $\mathbf{A}_2 = \alpha_s(\varphi_2) \cap \mathbf{AMS}_{\lceil\varphi_1*\varphi_2\rceil,s}$. Thus, in particular, $\mathbf{A}_2$ contains every AMS in $\alpha_s(\varphi_2)$ that can be obtained by composing an AMS in $\mathbf{AMS}_{\lceil\varphi_1 \mathrel{-\!\!\circledast} \varphi_2\rceil} = \mathbf{AMS}_{\lceil\varphi_2\rceil}$ with an AMS from $\alpha_s(\varphi_1) \cap \mathbf{AMS}_{\lceil\varphi_1\rceil,s}$. With Lemma 3.35 we then get that $(\mathsf{abst}_s(\varphi_1) \mathrel{-\!\!\bullet} \mathbf{A}_2) \cap \mathbf{AMS}_{\lceil\varphi_1 \mathrel{-\!\!\circledast} \varphi_2\rceil}$ is precisely the set of AMS $\alpha_s(\varphi_1 \mathrel{-\!\!\circledast} \varphi_2) \cap \mathbf{AMS}_{\lceil\varphi_1 \mathrel{-\!\!\circledast} \varphi_2\rceil}$.

Case $\varphi_1 \wedge \varphi_2$. By the induction hypotheses, we have for $1 \leq i \leq 2$ that $\mathsf{abst}_s(\varphi_i) = \alpha_s(\varphi_i) \cap \mathbf{AMS}_{\lceil\varphi_i\rceil,s}$. In case of $\lceil\varphi_1\rceil = 0$ or $\lceil\varphi_2\rceil = 0$ we have that all models of $\varphi_1 \wedge \varphi_2$ have a garbage-chunk count of 0 (by Lemma 3.36); hence, $\mathsf{abst}_s(\varphi_1 \wedge \varphi_2) = \mathsf{abst}_s(\varphi_1) \cap \mathsf{abst}_s(\varphi_2)$. Otherwise, for $1 \leq i \leq 2$, let $\mathbf{A}_i := \mathsf{lift}_{\lceil\varphi_1\rceil\nearrow\lceil\varphi\rceil}(\mathsf{abst}_s(\varphi_i))$. By Theorem 3.37, we have $\mathbf{A}_i = \alpha_s(\varphi_i) \cap \mathbf{AMS}_{\lceil\varphi\rceil,s}$. The claim thus follows from Lemma 3.31.

Case $\varphi_1 \vee \varphi_2$. By the induction hypotheses, we have for $1 \leq i \leq 2$ that $\mathsf{abst}_s(\varphi_i) = \alpha_s(\varphi_i) \cap \mathbf{AMS}_{\lceil\varphi_i\rceil,s}$. For $1 \leq i \leq 2$, let $\mathbf{A}_i := \mathsf{lift}_{\lceil\varphi_1\rceil\nearrow\lceil\varphi\rceil}(\mathsf{abst}_s(\varphi_i))$. By Theorem 3.37, we have $\mathbf{A}_i = \alpha_s(\varphi_i) \cap \mathbf{AMS}_{\lceil\varphi\rceil,s}$. The claim thus follows from Lemma 3.32.

Case $\neg\varphi_1$. By the induction hypothesis, we have that $\mathsf{abst}_s(\varphi_1) = \alpha_s(\varphi_1) \cap \mathbf{AMS}_{\lceil\varphi_1\rceil,s}$. We proceed by a case distinction: Assume $\lceil\varphi_1\rceil = \lceil\neg\varphi_1\rceil$. From Lemma 3.33, it follows that $\alpha_s(\neg\varphi_1) \cap \mathbf{AMS}_{\lceil\neg\varphi_1\rceil,s} = \mathbf{AMS}_{\lceil\neg\varphi_1\rceil,s} \setminus \alpha_s(\varphi_1) = \mathbf{AMS}_{\lceil\neg\varphi_1\rceil,s} \setminus (\alpha_s(\varphi_1) \cap \mathbf{AMS}_{\lceil\varphi_1\rceil,s})$. Assume $\lceil\varphi_1\rceil = 0$. By Lemma 3.36, all models of φ_1 have a garbage-chunk count of 0. Hence, $\mathsf{abst}_s(\neg\varphi_1) = \alpha_s(\neg\varphi_1) \cap \mathbf{AMS}_{\lceil\neg\varphi_1\rceil,s} = \mathbf{AMS}_{\lceil\neg\varphi_1\rceil,s} \setminus (\alpha_s(\varphi_1) \cap \mathbf{AMS}_{\lceil\varphi_1\rceil,s})$.

Computability of $\mathsf{abst}_s(\varphi)$. We note that the operators $\bullet$, $\mathrel{-\!\!\bullet}$, $\cap$, $\cup$ and $\setminus$ are all computable as the sets that occur in the definition of $\mathsf{abst}_s(\varphi)$ are all finite. It remains to argue that we can compute the set of AMS for all atomic formulas. This is trivial for **emp**, (dis-)equalities, and points-to assertions. For the list-segment predicate, we note that the set $\mathsf{abst}_s(\mathtt{ls}(x, y)) = \mathbf{AbstLists}(x, y) \cap \mathbf{AMS}_{\lceil 0\rceil,s}$ can be easily computed as there are only finitely many abstract lists w.r.t. the set of nodes $V = \mathsf{cls}_=(s)$. We obtain the following results:

Corollary 3.41. *Let s be a (finite) stack. Then* $\mathsf{abst}_s(\varphi)$ *is computable for all formulas φ.*

Theorem 3.42. *Let $\varphi \in \mathbf{SL}$ and let $\mathbf{x} \subseteq \mathbf{Var}$ be a finite set of variables with* $\mathsf{fvs}(\varphi) \subseteq \mathbf{x}$. *It is decidable whether there exists a model (s, h) with* $\mathsf{dom}(s) = \mathbf{x}$ *and* $(s, h) \models^{\mathsf{st}} \varphi$.

Proof. We consider stacks s with $\mathsf{dom}(s) = \mathbf{x}$; we observe that $\mathbf{C} := \{\mathsf{cls}_=(s) \mid \mathsf{dom}(s) \subseteq \mathbf{x}\}$ is finite; and that all stacks s, s' with $\mathsf{cls}_=(s) = \mathsf{cls}_=(s')$ have the same abstractions by Lemma 3.18. Consequently, we can compute the set $\{\mathsf{abst}_s(\varphi) \mid \mathsf{dom}(s) \subseteq \mathbf{x}\}$ by picking for each element $V \in \mathbf{C}$ one stack s with $\mathsf{cls}_=(s) = V$, and calculating $\mathsf{abst}_s(\varphi)$ for this stack. By Cor. 3.41, $\mathsf{abst}_s(\varphi)$ is computable for every such stack. By Theorem 3.40 and Cor. 3.38, φ is satisfiable over stack s iff $\mathsf{abst}_s(\varphi)$ is nonempty. Putting all this together, we obtain φ is satisfiable in stacks of size n if and only if any of finitely many computable sets $\mathsf{abst}_s(\varphi)$ is nonempty. □

Corollary 3.43. $\varphi \models^{\mathsf{st}}_{\mathbf{x}} \psi$ *is decidable for all finite sets of variables* $\mathbf{x} \subseteq \mathbf{Var}$ *and* $\varphi, \psi \in \mathbf{SL}$ *with* $\mathsf{fvs}(\varphi) \subseteq \mathbf{x}$ *and* $\mathsf{fvs}(\psi) \subseteq \mathbf{x}$.

Proof. $\varphi \models^{\mathsf{st}}_{\mathbf{x}} \psi$ iff $\varphi \wedge \neg\psi$ is unsatisfiable w.r.t. $\mathbf{x}$, which is decidable by Theorem 3.42. □

$$\mathsf{qbf\_to\_sl}(F) := \mathbf{emp} \wedge \bigwedge_{\text{pairwise different QBF variables } x,y} x \neq y \wedge \mathsf{aux}(F)$$

$$\mathsf{aux}(x) := (x \mapsto \mathsf{nil}) * \mathsf{t} \qquad \mathsf{aux}(\neg x) := \neg \mathsf{aux}(x)$$

$$\mathsf{aux}(F \wedge G) := \mathsf{aux}(F) \wedge \mathsf{aux}(G) \qquad \mathsf{aux}(F \vee G) := \mathsf{aux}(F) \vee \mathsf{aux}(G)$$

$$\mathsf{aux}(\exists x.\, F) := (x \mapsto \mathsf{nil} \vee \mathbf{emp}) \text{-}\circledast \mathsf{aux}(F) \qquad \mathsf{aux}(\forall x.\, F) := (x \mapsto \mathsf{nil} \vee \mathbf{emp}) \text{-}\!* \mathsf{aux}(F)$$

Fig. 7. Translation $\mathsf{qbf\_to\_sl}(F)$ from closed QBF formula F (in negation normal form) to a formula that is satisfiable iff F is true.

3.7 Complexity of the SSL Satisfiability Problem

It is easy to see that the algorithm $\mathsf{abst}_s(\varphi)$ runs in exponential time. We conclude this section with a proof that SSL satisfiability and entailment are actually PSpace-complete.

PSpace-hardness. An easy reduction from quantified Boolean formulas (QBF) shows that the SSL satisfiability problem is PSpace-hard. The reduction is presented in Fig. 7. We encode positive literals x by $(x \mapsto \mathsf{nil}) * \mathsf{t}$ (the heap contains the pointer $x \mapsto \mathsf{nil}$) and negative literals by $\neg((x \mapsto \mathsf{nil}) * \mathsf{t})$ (the heap does not contain the pointer $x \mapsto \mathsf{nil}$). The magic wand is used to simulate universals (i.e., to enforce that we consider both the case $x \mapsto \mathsf{nil}$ and the case **emp**, setting x both to true and to false). Analogously, septraction is used to simulate existentials. Similar reductions can be found (for standard SL) in [Calcagno et al. 2001].

Lemma 3.44. *The SSL satisfiability problem is PSpace-hard (even without the* ls *predicate).*

Note that this reduction simultaneously proves the PSpace-hardness of SSL model checking: If F is a QBF formula over variables $x_1, \ldots, x_k$, then $\mathsf{qbf\_to\_sl}(F)$ is satisfiable iff $(\{x_i \mapsto \ell_i \mid 1 \leq i \leq n\}, \emptyset) \models^{\mathsf{st}} \mathsf{qbf\_to\_sl}(F)$ for some locations ℓ_i with $\ell_i \neq \ell_j$ for $i \neq j$.

PSpace-membership. For every stack s and every bound on the garbage-chunk count of the AMS we consider, it is possible to encode every AMS by a string of polynomial length.

Lemma 3.45. *Let $k \in \mathbb{N}$, let s be a stack and $n := k + |s|$. There exists an injective function* $\mathsf{encode} \colon \mathbf{AMS}_{k,s} \to \{0,1\}^*$ *such that*

$$|\mathsf{encode}(\mathcal{A})| \in O(n \log(n)) \quad \text{for all } \mathcal{A} \in \mathbf{AMS}_{k,s}.$$

Proof. *(Lemma 3.45.)* Let $\mathcal{A} = \langle V, E, \rho, \gamma \rangle \in \mathbf{AMS}_{k,s}$. Each of the $|s| \leq n$ variables that occur in A can be encoded by a logarithmic number of bits. Observe that $|V| \leq |s|$, so V can be encoded by at most $O(n \log(n) + n)$ symbols (using a constant-length delimiter between the nodes). Each of the at most $|V|$ edges can be encoded by $O(\log(n))$ bits, encoding the position of the source and target nodes in the encoding of V by $O(\log(n))$ bits each and expending another bit to differentiate between $= 1$ and ≥ 2 edges. ρ can be encoded like V. Since $\gamma \leq k \leq n$, γ can be encoded by at most $\log(n)$ bits. In total, we thus have an encoding of length $O(n \log(n))$. □

An enumeration-based implementation of the algorithm in Fig. 6 (that has to keep in memory at most one AMS per subformula at any point in the computation) therefore runs in PSpace:

Lemma 3.46. *Let $\varphi \in \mathbf{SL}$ and let $\mathbf{x} \subseteq \mathbf{Var}$ be a finite set of variables with* $\mathsf{fvs}(\varphi) \subseteq \mathbf{x}$. *It is decidable in PSpace (in $|\varphi|$ and $|\mathbf{x}|$) whether there exists a model (s,h) with* $\mathrm{dom}(s) = \mathbf{x}$ *and* $(s,h) \models^{\mathsf{st}} \varphi$.

Proof. A simple induction on the structure of φ shows that it is possible to enumerate the set $\mathsf{abst}_s(\varphi)$ using at most $|\varphi|$ registers (each storing an AMS). The most interesting case is $\varphi_1 \text{-}\circledast \varphi_2$. Assume we can enumerate the

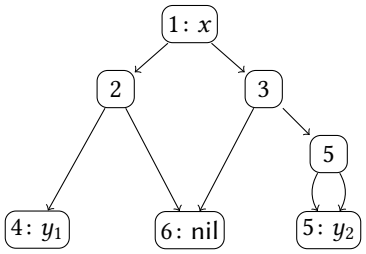


Fig. 8. Tree example: A stack-heap pair (s,h) with $(s,h) \models \mathtt{tree}(x; y_1, y_2, y_2; \mathsf{nil})$.

sets $\mathrm{abst}_s(\varphi_1) = \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ and $\mathrm{abst}_s(\varphi_2) = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ in polynomial space. We then use a new register in which we successively enumerate all $\mathcal{A} \in \mathbf{AMS}_{\lceil \varphi_1 \circledast \varphi_2 \rceil, s}$. This is done as follows: we enumerate all pairs of AMS $(\mathcal{A}_i, \mathcal{B}_j)$, $1 \le i \le m$, $1 \le j \le n$; we recognize that $\mathcal{A} \in \mathbf{AMS}_{\lceil \varphi_1 \circledast \varphi_2 \rceil, s}$ iff $\mathcal{B}_j = \mathcal{A}_i \bullet \mathcal{A}$ for any of these pairs $(\mathcal{A}_i, \mathcal{B}_j)$.

□

The PSPACE-completeness result, Theorem 3.1, follows by combining Lemmas 3.44 and 3.46.

3.8 Extension to Trees

In this section, we show that all our results continue to hold when we add a tree predicate to our separation logic. In what follows, we only state the definitions and results that need to be adapted, most of the definitions and results from the previous sections, however, do not need to be changed.

We begin by extending our memory model: We allow pointers to point to either one or two successor locations, i.e., we extend our previous definition of heaps and consider partial functions

$$h \colon \mathbf{Loc} \rightharpoonup \mathbf{Loc} \cup \mathbf{Loc} \times \mathbf{Loc}.$$

With pointers being able to point to more than one location, the heap can now form more general graph-theoretic structures, in particular trees.

We now extend the syntax and semantics of our separation logic (as stated in Figures 2 and 3) to tree predicates and points-to predicates with two target locations:

$$\tau \quad ::= \quad \cdots \mid x \mapsto \langle y, z \rangle \mid \mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$$

$$(s,h) \models x \mapsto \langle y, z \rangle \qquad \text{iff} \quad h = \{s(x) \mapsto \langle s(y), s(y) \rangle\}$$

$(s,h) \models \mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$ iff

$\mathrm{dom}(h) = \emptyset, n = 1$ and $s(x) = s(y_1)$, or
$\mathrm{dom}(h) = \emptyset, n = 0$ and $s(x) = s(z_i)$ for some $i \in \{1, \ldots, m\}$, or
there is some $\ell \in \mathbf{Loc}$ and a fresh variable $u \in \mathbf{Var}$ such that
$\quad (s[u \mapsto \ell], h) \models x \mapsto u * \mathtt{tree}(u; y_1, \ldots, y_n; z_1, \ldots, z_m)$, or
there are some $\ell_1, \ell_2 \in \mathbf{Loc}$, fresh variables $u, v \in \mathbf{Var}$, and some
partitioning of $y_1, \ldots, y_n$ into $a_1, \ldots, a_k$ and $b_1, \ldots, b_l$ such that
$\quad (s[u \mapsto \ell_1, v \mapsto \ell_2], h) \models x \mapsto \langle u, v \rangle * \mathtt{tree}(u; a_1, \ldots, a_k; z_1, \ldots, z_m)$
$\qquad\qquad * \mathtt{tree}(v; b_1, \ldots, b_l; z_1, \ldots, z_m)$

We note that in a tree predicate $\mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$ we distinguish between the *root* x, *leaves* $y_1, \ldots, y_n$ and *sinks* $z_1, \ldots, z_m$. We give an example for a tree with three leaves and one sink in Fig. 8; note that leaves can repeat (as y_2 in Fig. 8) but the tree definition ensures that for a leaf we precisely track the

number of incoming pointers, whereas sinks can always have an arbitrary number of incoming pointers. We comment on the recursive definition of the tree predicate: The base cases state that each tree either ends in a leaf or in a sink location; the composite cases stipulate that the successor locations respect the requirements for leaves and sinks; in particular, in case of two successors the leaves $y_1, \ldots, y_n$ can be partitioned into the leaves $a_1, \ldots, a_k$ and $b_1, \ldots, b_l$ of the respective sub-trees, i.e., we require that $k + l = n$ and that $a_1, \ldots, a_k, b_1, \ldots, b_l$ is a permutation of $y_1, \ldots, y_n$. We want to distinguish between leaves and sinks in order to be able to reason about tree composition. That is, we want to generalize the following reasoning to trees: For lists, we can prove that $\texttt{ls}(x, y) * \texttt{ls}(y, z) \models \texttt{ls}(x, z)$, i.e., that the composition of the list-segment predicates $\texttt{ls}(x, y)$ and $\texttt{ls}(y, z)$ implies the list-segment predicate $\texttt{ls}(x, z)$. Indeed, we have the following property about tree composition:

Proposition 3.47.

$$\texttt{tree}(x; y, y_1, \ldots, y_n; z_1, \ldots, z_m) * \texttt{tree}(y; w_1, \ldots, w_k; z_1, \ldots, z_m) \models \texttt{tree}(x; y_1, \ldots, y_n, w_1, \ldots, w_k; z_1, \ldots, z_m)$$

Proof. Direct from the semantics of the tree predicate. □

We note that the tree predicate generalizes the list segment predicate: it is easy to verify that the predicate $\texttt{tree}(x; y; \epsilon)$, where ϵ is the empty sequence of variables, is satisfied by the same set of stack-heap pairs (s, h) as the list segment predicate $\texttt{ls}(x, y)$.

Correspondence of Strong and Weak Semantics on Positive Formulas. The correspondence continues to hold for the positive fragment of the extended logic. (It is sufficient to check that the base case of Lemma 2.5 is also satisfied for the tree predicate).

Chunks. The definition of positive and negative chunks (Definitions 3.8) does not have to be changed, because positive chunks are defined with regard to the satisfaction of any atomic formula τ, which can now also be tree predicates.

The AMS abstraction. We need to generalize the AMS abstraction in order to incorporate pointers with multiple successors and trees. For this, we need to assume an upper bound k on the number of leaves that can appear in a tree predicate, i.e., we require $n \leq k$ for $\texttt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$. We are now ready to state the AMS generalization; we only need to change the second component of AMSs:

Definition 3.48 (AMS Edges). We define AMS edges to be the partial function

$$E \colon V \rightharpoonup (V \cup V \times V) \times \{=1\} \cup (V \to \{0, \ldots, k+1\} \cup \{\infty\}) \times \{\geq 2\}$$

such that there is no $\mathbf{v} \in \mathrm{dom}(E)$ with $\mathsf{nil} \in \mathbf{v}$.

Intuitively, the AMS edges store whether there is a single pointer with one or two successors, or a tree with at least two allocated locations, for which we store the number of tree edges whose target location is in the image of the stack. We store the exact number of such edges in case of less or equal to $k + 1$ edges in order to be able to precisely reason about the number of incoming edges of a tree leaf. We represent more than $k + 1$ edges by ∞, which is sufficient to reason about sinks. We now give an intuition why the bound of $k + 1$ is sufficient to reason about the existence of models for formulas that include tree predicates with at most k leaves: Consider for example the predicate $\texttt{tree}(x; \epsilon; \epsilon)$; i.e., we have a predicate with $k = 0$ leaves. Then, a model of $\texttt{tree}(x; \epsilon; \epsilon)$ might be composed of two chunks that are models of $\texttt{tree}(x; y; \epsilon)$ and $\texttt{tree}(y; \epsilon; \epsilon)$; note that these predicates use at most $k + 1 = 1$ leaves. In contrast, there can never be a model of $\texttt{tree}(x; \epsilon; \epsilon)$ that is composed of some chunk that has two or more times the same leaf, e.g., $\texttt{tree}(x; y, y; \epsilon)$, because such a chunk would need to be composed with two other chunks that allocate y (which is not possible).

We generalize the AMS induced by a model (s,h) accordingly: For every equivalence class $[x]^s_= \in \mathrm{cls}_=(s)$, we set

$$\mathrm{edges}(s,h)([x]^s_=) := \begin{cases} \langle [y]^s_=, =1\rangle & \text{there are } y \in \mathrm{dom}(s), h_c \in \mathsf{chunks}^+(s,h) \text{ with } (s,h_c) \overset{\text{st}}{\models} x \mapsto y \\ \langle \langle [y]^s_=, [z]^s_= \rangle, =1\rangle & \text{there are } y,z \in \mathrm{dom}(s), h_c \in \mathsf{chunks}^+ \text{ with } (s,h_c) \overset{\text{st}}{\models} x \mapsto \langle y,z\rangle \\ \langle \{v \mapsto d_v\}, \geq 2\rangle, & \text{there are } y_1,\dots,y_n,z_1,\dots,z_m \in \mathrm{dom}(s), h_c \in \mathsf{chunks}^+(s,h) \text{ with} \\ & (s,h_c) \overset{\text{st}}{\models} \mathtt{tree}(x; y_1,\dots,y_n; z_1,\dots,z_m) \\ & \qquad \wedge \neg x \mapsto y_1 \wedge \neg x \mapsto \langle y_1, y_2\rangle \wedge \neg x \mapsto \langle y_2, y_1\rangle, \text{ and} \\ & \text{for every } v \in V \text{ with } d_v < \infty \text{ there are exactly } d_v \text{ variables } y_i \\ & \text{with } y_i \in v, \text{ and for every } v \in V \text{ with } d_v = \infty \text{ there is some } z_i \in v \\ & \text{with } |h_c^{-1}(s(z_i))| > k+1 \\ \bot, & \text{otherwise} \end{cases}$$

We note that we do not need to include a separate case for lists in the extended definition of AMS edges, as lists are covered as a special case of $\langle \{v \mapsto d_v\}, \geq 2\rangle$, where $\sum_{v\in V} d_v = 1$, i.e., there is a single edge whose target location is in the image of the stack.

Lemma 3.49 (Realizability of AMS). *Let $\mathcal{A} = \langle V, E, \rho, \gamma\rangle$ be an AMS. There exists a model $(s,h) = \mathrm{model}(\mathcal{A})$ with $\mathrm{ams}(s,h) = \mathcal{A}$ whose size is linear in the size of $\mathcal{A}$ (we assume a unary representation of the numbers in AMS edges).*

Proof. *(Lemma 3.49.)* For every $v \in V$ we fix a location $\ell_v \in \mathbf{Loc}$, for every $\mathbf{r} \in \rho$ we fix a location $\ell_{\mathbf{r}} \in \mathbf{Loc}$ and for every $1 \leq i \leq \gamma$ we fix a location $\ell_i \in \mathbf{Loc}$; we assume all these locations to be different. We set $s := \bigcup_{x\in v, v\in V} \{x \mapsto \ell_v\}$. For every node $v \in V$ with $E(v) = \langle \{w \mapsto d_w\}, \geq 2\rangle$, we fix some set of locations $\mathbf{Loc}_v = \bigcup_{w\in V} \left\{c^w_1, \dots, c^w_{e_w}\right\} \subseteq \mathbf{Loc}$, where $e_w = d_w$, in case $d_w < \infty$, and $e_w = k+2$, otherwise; we require all those sets $\mathbf{Loc}_v$ to be pairwise disjoint and disjoint from the sets $\{\ell_v \in \mathbf{Loc} \mid v \in V\}$, $\{\ell_{\mathbf{r}} \mid \mathbf{r} \in \rho\}$, and $\{\ell_i \mid \mathbf{r} \in 1 \leq i \leq \gamma\}$.

We now define h as the (disjoint) union of the following sets:

- For every $v \in V$ with $E(v) = \langle v', =1\rangle$ the set
$$\{\ell_v \mapsto \ell_{v'}\}.$$
- For every $v \in V$ with $E(v) = \langle \langle v', v''\rangle, =1\rangle$ the set
$$\{\ell_v \mapsto \langle \ell_{v'}, \ell_{v''}\rangle\}.$$
- For every $v \in V$ with $E(v) = \langle \{w \mapsto d_w\}, \geq 2\rangle$, using the locations $\mathbf{Loc}_v = \bigcup_{w\in V} \left\{c^w_1, \dots, c^w_{e_w}\right\}$ and fixing an arbitrary order $V = \{w_1, \dots, w_n\}$, the set
$$\left\{\ell_v \mapsto c^{w_1}_1\right\} \cup \bigcup_{1\leq i<n} \left\{c^{w_i}_j \mapsto \left\langle \ell_{w_i}, c^{w_i}_{j+1}\right\rangle \mid 1 \leq j < e_{w_i}\right\} \cup \left\{c^{w_i}_{e_{w_i}} \mapsto \left\langle \ell_{w_i}, c^{w_{i+1}}_1\right\rangle\right\}$$
$$\cup \left\{c^{w_n}_j \mapsto \left\langle \ell_{w_n}, c^{w_n}_{j+1}\right\rangle \mid 1 \leq j < e_{w_n}\right\} \cup \left\{c^{w_n}_{e_{w_n}} \mapsto \ell_{w_n}\right\}$$
- For every $\mathbf{r} \in \rho$ the set
$$\{\ell_{\mathbf{r}} \mapsto \ell_{\mathbf{r}}\} \cup \bigcup_{v\in\mathbf{r}} \{\ell_v \mapsto \ell_{\mathbf{r}}\}.$$

$$\frac{}{\{x \mapsto z\}\, x.\mathsf{next} := y\, \{x \mapsto y\}} \qquad \frac{}{\{\mathbf{emp}\}\, \mathsf{malloc}(x)\, \{x \mapsto m\}}$$

$$\frac{}{\{x \mapsto z\}\, \mathsf{free}(x)\, \{\mathbf{emp}\}} \qquad \frac{}{\{\mathbf{emp}\}\, x := y\, \{x = y\}}$$

$$\frac{}{\{y \mapsto z\}\, x := y.\mathsf{next}\, \{y \mapsto z * x = z\}}\ x \text{ different from } y$$

$$\frac{}{\{\mathbf{emp}\}\, \mathsf{assume}(\varphi)\, \{\varphi\}}\ \varphi \text{ is } x = y \text{ or } x \neq y$$

Fig. 9. Local proof rules of program statements for forward symbolic execution.

- For every $1 \leq i \leq \gamma$ the set

$$\{\ell_i \mapsto \ell_i\}\,.$$

It is easy to verify that $\mathsf{ams}(s,h) = \mathcal{A}$ and that $|h| \in O(|\mathcal{A}|)$. □

We now define abstract trees; this notion allows us to characterize the AMSs arising from abstracting trees.

Definition 3.50. Given some $\mathcal{A} = \langle V, E, \rho, \gamma \rangle \in \mathbf{AMS}$ and $x, y_1, \ldots, y_n, z_1, \ldots, z_m \in \mathbf{Var}$, we say that $\mathcal{A}$ is an *abstract tree* with root x, leaves $y_1, \ldots, y_n$ and sinks $z_1, \ldots, z_m$, in signs $\mathcal{A} \in \mathbf{AbstTrees}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$, iff $\mathsf{model}(\mathcal{A}) \models \mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$, where $\mathsf{model}(\mathcal{A})$ is the canonical model of AMS $\mathcal{A}$ from Lemma 3.49.

We now show that the notion of abstract trees indeed characterizes the models that satisfy tree predicates:

Lemma 3.51. *For all stack-heap pairs (s,h), we have that $(s,h) \models \mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$ iff $\mathsf{ams}(s,h) \in \mathbf{AbstTrees}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$.*

Proof. We need to argue that $(s,h) \models \mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$ iff $\mathsf{model}(\mathsf{ams}(s,h)) \models \mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$.

Given some (s,h), we argue that $(s,h) \models \mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$ implies $\mathsf{model}(\mathsf{ams}(s,h)) \models \mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$. The proof of the other implication is similar. Let us assume that $(s,h) \models \mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$. We note that $\mathsf{ams}(\mathsf{model}(\mathsf{ams}(s,h))) = \mathsf{ams}(s,h)$. Hence, $\mathsf{model}(\mathsf{ams}(s,h))$ does not contain garbage and fully decomposes into positive chunks. We now observe that the positive chunks that consist of a single points-to assertion are the same in both models, only the chunks that belong to trees with ≥ 2 allocated locations may differ. For those tree chunks we observe that they agree on the root, leaves and sinks, only their number of internal locations may differ. We now argue that $\mathsf{model}(\mathsf{ams}(s,h)) \models \mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$. We need to ensure that we can find a recursive unfolding of the tree predicate according to the semantics of the tree predicate. We can construct such an unfolding using the unfolding of semantics of the tree predicate in $(s,h) \models \mathtt{tree}(x; y_1, \ldots, y_n; z_1, \ldots, z_m)$ and using that the chunks of both models abstract to the same AMSs, i.e., have the same roots, leaves and sinks. We note that we can precisely track the leaves $y_1, \ldots, y_n$ for an upper bound $n \leq k$ on the number of leaves. This is because each variable can be allocated at most once and hence $k+1$ is an upper bound on the number of times the same variable y_i can appear as a leaf of some tree chunk. □

This result on abstract trees is all that is needed to generalize our decision procedure and complexity results to our extended separation logic. (In particular, we note that Lemma 3.51 covers the base cases in the proofs of Theorems 3.19 and 3.37.)

$$\text{Frame rule } \frac{\{P\}\, c\, \{Q\}}{\{A * P\}\, c\, \{A[\mathbf{x}'/\mathbf{x}] * Q\}}\ \mathbf{x} = \mathsf{modifiedVars}(c),\ \mathbf{x}' \text{ fresh}$$

$$\text{Materialization } \frac{\{P\}\, c\, \{Q\}}{\{P\}\, c\, \{x \mapsto z * ((x \mapsto z) \text{-}\circledast Q)\}}\ Q \models^{\text{st}} \neg((x \mapsto \mathsf{nil}) \text{-}\circledast \mathsf{t}),\ z \text{ fresh}$$

Fig. 10. The frame and the materialization rule for forward symbolic execution.

4 PROGRAM VERIFICATION WITH STRONG-SEPARATION LOGIC

Our main practical motivation behind SSL is to obtain a decidable logic that can be used for fully automatically discharging verification conditions in a Hoare-style verification proof. Discharging VCs can be automated by calculi that symbolically execute pre-conditions forward resp. post-conditions backward, and then invoking an entailment checker. Symbolic execution calculi typically either introduce first-order quantifiers or fresh variables in order to deal with updates to the program variables. We leave the extension of SSL to support for quantifiers for future work and in this paper develop a forward symbolic execution calculus based on fresh variables.

We target the usual Hoare-style setting where a verification engineer annotates the pre- and post-condition of a function and provides loop invariants. We exemplify two annotated functions in Fig. 12; the left function reverses a list and the right function copies a list. In addition to the program variables, our annotations may contain logical variables (also known as ghost variables); for example, the annotations of list reverse only contain program variables, while the annotations of list copy also contain the logical variable u (which is assumed to be equal to x in the pre-condition)[10].

A simple heap-manipulating programming language. We consider the six program statements x.next := y, x := y.next (where x is different from y), free(x), malloc(x), x := y and assume(φ), where φ is $x = y$ or $x \neq y$. We remark that we do not include a statement x := x.next for ease of exposition; however, this is w.l.o.g. because x := x.next can be simulated by the statements y := x.next; x := y at the expense of introducing an additional program variable y. We specify the semantics of the considered program statements via a small-step operational semantics. We state the semantics in Fig. 11, where we write $(s,h) \xrightarrow{c} (s',h')$, with the meaning that executing c in state (s,h) leads to state (s',h'), and $(s,h) \xrightarrow{c}$ error, when executing c leads to an error. Our only non-standard choice is the modelling of the malloc statement: we assume a special program variable m, which is never referenced by any program statement and only used in the modelling; the malloc statement updates the value of the variable m to the target of the newly allocated memory cell; we include m in order to have a name for the target of the newly allocated memory cell. We say program statement c is *safe* for a stack-heap pair (s,h) if there is no transition $(s,h) \xrightarrow{c}$ error. Given a sequence of program statements $\mathbf{c} = c_1 \cdots c_k$, we write $(s,h) \xrightarrow{\mathbf{c}} (s',h')$, if there are stack-heap pairs (s_i,h_i), with $(s_0,h_0) = (s,h)$, $(s_k,h_k) = (s',h')$ and $(s_i,h_i) \xrightarrow{c_i} (s_{i+1},h_{i+1})$ for all $1 \leq i \leq k$.

Forward Symbolic Execution Rules. The rules for the program statements in Fig. 9 are local in the sense that they only deal with a single pointer or the empty heap. The rules in Fig. 10 are the main rules of our forward symbolic execution calculus. The frame rule is essential for lifting the local proof rules to larger heaps. Note that the frame rule requires substituting the modified program variables with fresh copies: We set $\mathsf{modifiedVars}(c) := \{x, m\}$ for $c = \mathsf{malloc}(x)$, $\mathsf{modifiedVars}(c) := \{x\}$ for $c =$ x := y.next and $c =$ x := y, and $\mathsf{modifiedVars}(c) := \emptyset$, otherwise. The materialization rule ensures that the frame rule can be applied whenever the pre-condition of a local proof rule can be met. We now give more details. For a sequence of program statements $\mathbf{c} = c_1 \cdots c_k$ and a pre-condition

[10] m is a special program variable introduced for modelling malloc.

$(s,h) \xrightarrow{x.\text{next}:=y} (s',h')$	if $s(x) \in \text{dom}(h)$, with $s' = s$ and $h' = h[s(x)/s(y)]$
$(s,h) \xrightarrow{x.\text{next}:=y} \text{error}$	if $s(x) \notin \text{dom}(h)$
$(s,h) \xrightarrow{x:=y.\text{next}} (s',h')$	if $s(y) \in \text{dom}(h)$, with $s' = s[x/h(s(y))]$ and $h' = h$
$(s,h) \xrightarrow{x:=y.\text{next}} \text{error}$	if $s(y) \notin \text{dom}(h)$
$(s,h) \xrightarrow{\text{free}(x)} (s',h')$	if $s(x) \in \text{dom}(h)$, with $s' = s$ and $h' = h[s(x)/\bot]$
$(s,h) \xrightarrow{\text{free}(x)} \text{error}$	if $s(x) \notin \text{dom}(h)$
$(s,h) \xrightarrow{\text{malloc}(x)} (s',h')$	with $s' = s[x/l][m/k]$ and $h' = h[l/k]$ for some $l \in \mathbf{Loc} \setminus \text{dom}(h)$, $k \in \mathbf{Loc}$
$(s,h) \xrightarrow{x:=y} (s',h')$	with $s' = s[x/s(y)]$ and $h' = h$
$(s,h) \xrightarrow{\text{assume}(\varphi)} (s',h')$, where φ is $x = y$ or $x \neq y$	if $s(x) = s(y)$ resp. $s(x) \neq s(y)$, with $s' = s$ and $h' = h$

Fig. 11. Semantics of program statements.

P_{start}, the symbolic execution calculus derives triples $\{P_{\text{start}}\}\, c_1 \cdots c_i\, \{Q_i\}$ for all $1 \leq i \leq k$. In order to proceed from i to $i+1$, either 1) only the frame rule is applied or 2) the materialization rule is applied first followed by an application of the frame rule. The frame rule can be applied if the formula Q_i has the shape $Q_i = A * P$, where A is suitably chosen and P is the pre-condition of the local proof rule for statement c_i. Then, Q_{i+1} is given by $Q_{i+1} = A[\mathbf{x}'/\mathbf{x}] * Q$, where $\mathbf{x} = \text{modifiedVars}(c)$, $\mathbf{x}'$ are fresh copies of the variables $\mathbf{x}$ and Q is the right hand side of the local proof rule for statement c_i, i.e., we have $\{P\}\, c_i\, \{Q\}$. The materialization rule may be applied in order to ensure that Q_i has the shape $Q_i = A * P$. This is not needed in case $P = \mathbf{emp}$ but may be necessary for points-to assertions such as $P = x \mapsto y$. We note that Q_i guarantees that a pointer x is allocated iff $Q_i \models^{\text{st}} \neg((x \mapsto \text{nil}) \circledast \text{t})$. Under this condition, the rule allows introducing a name z for the target of the pointer x. We remark that while backward-symbolic execution calculi typically employ the magic wand, our forward calculus makes use of the dual septraction operator: this operator allowed us to design a general rule that guarantees a predicate of shape $Q_i = A * P$ without the need of coming up with dedicated rules for, e.g., unfolding list predicates.

Applying the forward symbolic execution calculus for verification. We now explain how the proof rules presented in Fig. 9 and 10 can be used for program verification. Our goal is to verify that the pre-condition P of a loop-free piece of code c (in our case, a sequence of program statements) implies the post-condition Q. For this, we apply the symbolic execution calculus and derive a triple $\{P\}\, c\, \{Q'\}$. It then remains to verify that the final state of the symbolic execution Q' implies the post-condition Q. Here, we face the difficulty that the symbolic execution introduces additional variables: Let us assume that all annotations are over a set of variables $\mathbf{x}$, which includes the program variables and the logical variables. Further assume that the symbolic execution $\{P\}\, c\, \{Q'\}$ introduced the fresh variables $\mathbf{y}$. With the results of Section 3 we can then verify the entailment $Q' \models^{\text{st}}_{\mathbf{x}\cup\mathbf{y}} Q$. However, we need to guarantee that all models (s,h) of Q with $\text{dom}(s) = \mathbf{x} \cup \mathbf{y}$ are also models when we restrict $\text{dom}(s)$ to $\mathbf{x}$ (note that the variables $\mathbf{y}$ are implicitly existentially quantified; we make this statement precise in Lemma 4.6 below). In order to deal with this issue, we require annotations to be robust:

Definition 4.1 (Robust Formula). We call a formula $\varphi \in \mathbf{SL}$ *robust*, if for all models (s_1, h) and (s_2, h) with $\mathsf{fvs}(\varphi) \subseteq \mathsf{dom}(s_1)$ and $\mathsf{fvs}(\varphi) \subseteq \mathsf{dom}(s_2)$ and $s_1(x) = s_2(x)$ for all $x \in \mathsf{fvs}(\varphi)$, we have that $(s_1, h) \models^{\mathsf{st}} \varphi$ iff $(s_2, h) \models^{\mathsf{st}} \varphi$.

We identify a fragment of robust formulas in the next lemma. In particular, we obtain that the annotations in Fig. 12 are robust.

Lemma 4.2. *Let $\varphi \in \mathbf{SL}$ be a positive formula. Then, φ is robust.*

Proof. Let (s_1, h) and (s_2, h) be two models with $s_1(x) = s_2(x)$ for all $x \in \mathsf{fvs}(\varphi)$. Then, by Lemma 2.7 we have that $(s_1, h) \models^{\mathsf{st}} \varphi$ iff $(s_1, h) \models^{\mathsf{wk}} \varphi$ iff $(s_2, h) \models^{\mathsf{wk}} \varphi$ iff $(s_2, h) \models^{\mathsf{st}} \varphi$. □

The following lemma allows us to construct robust formulas from known robust formulas:

Lemma 4.3. *Let $\varphi \in \mathbf{SL}$ be formula. If φ is robust, then $\varphi * x \mapsto y$ and $x \mapsto y \text{-}\circledast \varphi$ are robust.*

Proof. Immediate from the definition of a robust formula. □

Not all formulas are robust, e.g., consider φ from Example 2.2. On the other hand, Lemma 2.7 does not cover all robust formulas, e.g., t is robust. We leave the identification of further robust formulas for future work.

Soundness of Forward Symbolic Execution. We adapt the notion of a *local action* from [Calcagno et al. 2007] to contracts:

Definition 4.4 (Local Contract). Given some program statement c and SL formulae P, Q, we say the triple $\{P\}\, c\, \{Q\}$ is a *local contract*, if for every stack-heap pair (s, h) with $(s, h) \models^{\mathsf{st}} P$, every stack t with $s \subseteq t$ and every heap h° with $h \uplus^t h^\circ \neq \bot$, we have that

1) c is safe for $(t, h \uplus^t h^\circ)$, and
2) for every stack-heap pair (t', h') with $(t, h \uplus^t h^\circ) \xrightarrow{c} (t', h')$ there is some heap $h^\#$ with $h^\# \uplus^{t'} h^\circ = h'$ and $(t', h^\#) \models^{\mathsf{st}} Q$.

We now state that our local proofs rules specify local contracts:

Lemma 4.5. *Let c be a program statement, and let $\{P\}\, c\, \{Q\}$ be the triple from its local proof rule as stated in Fig. 9. Then, $\{P\}\, c\, \{Q\}$ is a local contract.*

Proof. The requirements 1) and 2) of local contracts can be directly verified from the semantics of the program statements. □

We are now ready to state the soundness of our symbolic execution calculus (we assume robust formulas A in the frame rule, which can be ensured by the materialization rule[11]); we note that the statement makes precise the implicitly existentially quantified variables, stating that there is an extension of the stack to the variables V introduced by the symbolic execution such that Q holds:

Lemma 4.6 (Soundness of Forward Symbolic Execution). *Let $\mathbf{c}$ be a sequence of program statements, let P be a robust formula, let $\{P\}\, \mathbf{c}\, \{Q\}$ be the triple obtained from symbolic execution, and let V be the fresh variables introduced during symbolic execution. Then, Q is robust and for all $(s, h) \xrightarrow{\mathbf{c}} (s', h')$ with $(s, h) \models^{\mathsf{st}} P$, there is a stack s'' with $s' \subseteq s''$, $V \subseteq \mathsf{dom}(s'')$ and $(s'', h') \models^{\mathsf{st}} Q$.*

[11] Assume that Q is the robust formula currently derived by the symbolic execution, that c is the next program statement, that $\{P_c\}\, c\, \{Q_c\}$ is the triple from the local proof rule of program statement c, and that Q is of shape $Q = A * P_c$. In case A is not robust (note that this is only possible if $P_c = x \mapsto z$ for some variables x and z), then one can first apply the materialization rule in order to derive formula $Q' = x \mapsto z * ((x \mapsto z) \text{-}\circledast Q)$. Then, $A' = (x \mapsto z) \text{-}\circledast Q$ is robust by Lemma 4.3.

Proof. See appendix. □

(Lemma 4.6.) We prove the claim by induction on the number of applications of the frame rule and materialization rule. We consider a sequence of program statements $\mathbf{c} = c_1 \cdots c_n$ for which the triple $\{P\}\,\mathbf{c}\,\{Q\}$ was derived by symbolic execution, introducing some fresh variables V. We then assume that the claim holds for $\{P\}\,\mathbf{c}\,\{Q\}$ (for the base case we allow $\mathbf{c}$ to be the empty sequence, i.e, $\mathbf{c} = \epsilon$, and consider $\{P\}\,\epsilon\,\{P\}$) and prove the claim for one more application of the frame rule or the materialization rule.

We first consider an application of the materialization rule. Then, we have $Q \models^{\text{st}} \neg((x \mapsto \text{nil}) \text{-}\!\circledast \text{t})$ (*) and we infer the triple $\{P\}\,\mathbf{c}\,\{x \mapsto z * ((x \mapsto z) \text{-}\!\circledast Q)\}$, where z is some fresh variable. We now consider some stack-heap pairs $(s,h) \xrightarrow{\mathbf{c}} (s',h')$ with $(s,h) \models^{\text{st}} P$. Because the claim holds for $\{P\}\,\mathbf{c}\,\{Q\}$ and V, there is some stack s'' with $s' \subseteq s''$, $V \subseteq \text{dom}(s'')$ and $(s'',h') \models^{\text{st}} Q$. Because of (*) we have $(s'',h') \models^{\text{st}} \neg((x \mapsto \text{nil}) \text{-}\!\circledast \text{t})$. Hence, there is some $\ell \in \mathbf{Loc}$ such that $h'(s''(x)) = \ell$. We now consider the stack $s''' = s''[z/\ell]$. Note that $s' \subseteq s'''$. Because Q is robust by induction assumption, we then have that $(s''',h') \models^{\text{st}} x \mapsto z * ((x \mapsto z) \text{-}\!\circledast Q)$. Moreover, we get from Lemma 4.3 that $x \mapsto z * ((x \mapsto z) \text{-}\!\circledast Q)$ is robust. Thus, the claim is satisfied for the set of variables $V' = V \cup \{z\}$.

We now consider an application of the frame rule, i.e., we need to prove the claim for the sequence $\mathbf{c}' = \mathbf{c}c$, which extends $\mathbf{c}$ by some statement c. Let $\{P_c\}\,c\,\{Q_c\}$ be the triple from the local proof rule for c. Because the frame rule is applied, we have by assumption that there is some robust SL formula A with $Q = A * P_c$. From the application of the frame rule we then obtain $\{P\}\,\mathbf{c}'\,\{A[\mathbf{x}'/\mathbf{x}] * Q_c\}$, where $\mathbf{x} = \text{modifiedVars}(c)$ and $\mathbf{x}'$ fresh. We now consider some stack-heap pairs $(s,h) \xrightarrow{c'} (s',h')$ with $(s,h) \models^{\text{st}} P$. Then, there is some stack-heap pair (s'',h'') with $(s,h) \xrightarrow{\mathbf{c}} (s'',h'')$ and $(s'',h'') \xrightarrow{c} (s',h')$. Because the claim holds for $\{P\}\,\mathbf{c}\,\{Q\}$ and V, there is some stack t with $s'' \subseteq t$, $V \subseteq \text{dom}(t)$ and $(t,h'') \models^{\text{st}} Q$. Because of $Q = A * P_c$, we have that there are some heaps h_1, h_2 with $h_1 \uplus^t h_2 = h''$ such that $(t,h_1) \models^{\text{st}} P_c$ and $(t,h_2) \models^{\text{st}} A$. Because of $s'' \subseteq t$ and $(s'',h'') \xrightarrow{c} (s',h')$, we get that $(t,h'') \xrightarrow{c} (t',h')$ for some $s' \subseteq t'$. Because $\{P_c\}\,c\,\{Q_c\}$ is a local contract by Lemma 4.5, we get that there is a heap h_1' with $h_1' \uplus^{t'} h_2 = h'$ and $(t',h_1') \models^{\text{st}} Q_c$. We now consider the stack t'' defined by $t''(x) = t'(x)$ for all $x \in \text{dom}(t')$ and $t''(x') = t(x)$ for all $x \in \text{modifiedVars}(c)$, where $x' \in \mathbf{x}'$ is the fresh copy created for x. Note that $t' \subseteq t''$. We recall that A is robust by assumption. Hence, $(t'',h_2) \models^{\text{st}} A[\mathbf{x}'/\mathbf{x}]$. Moreover, Q_c is robust by Lemma 4.2. Hence, $(t'',h_1) \models^{\text{st}} Q_c$. Thus, $(t'',h') \models^{\text{st}} A[\mathbf{x}'/\mathbf{x}] * Q_c$. Moreover, we get from Lemma 4.3 that $A[\mathbf{x}'/\mathbf{x}] * Q_c$ is robust. Hence, the claim is satisfied for the set of variables $V' = V \cup \mathbf{x}'$.

Automation. We note that the presented approach can fully-automatically verify that the pre-condition of a loop-free piece of code guarantees its post-condition: For every program statement, we apply its local proof rule using the frame rule (and in addition the materialization rule in case the existence of a pointer target must be guaranteed). We then discharge the entailment query using our decision procedure from Section 3. We now illustrate this approach on the programs from Fig. 12. For both programs we verify that the loop invariant is inductive (in both cases the loop-invariant P is propagated forward through the loop body; it is then checked that the obtained formula Q again implies the loop invariant P; for verifying the implication we apply our decision procedure from Corollary 3.43):

```
% list reverse
{ls(x, nil)}
    a := nil;
    while(x ≠ nil)
    {ls(x, nil) ∗ ls(a, nil)}
    {
        b := x.next;
        x.next := a;
        a := x;
        w := b;
    }
    x := w;
{ls(x, nil)}
```

```
% list copy, where the copy has an additional head node
{ls(x, nil) ∗ u = x}
    malloc(s);
    r := s;
    while(x ≠ nil)
    {ls(u, x) ∗ ls(x, nil) ∗ ls(r, s) ∗ s ↦ m}
    {
        malloc(t);
        % t.data := x.data; not modelled
        s.next := t;
        s := t;
        y := x.next;
        x := y;
    }
    s.next := nil;
{ls(u, nil) ∗ ls(r, nil)}
```

Fig. 12. List reverse (left) and list copy (right) annotated pre- and post-condition and loop invariants.

Example 4.7. Verifying the loop invariant of list reverse:

$\{\mathtt{ls}(x, \mathsf{nil}) * \mathtt{ls}(a, \mathsf{nil})\}\ (=: P)$
  assume(x ≠ nil)
$\{\mathtt{ls}(x, \mathsf{nil}) * \mathtt{ls}(a, \mathsf{nil}) * x \neq \mathsf{nil}\}$
  # materialization
$\{x \mapsto z \circledast (\mathtt{ls}(x, \mathsf{nil}) * \mathtt{ls}(a, \mathsf{nil}) * x \neq \mathsf{nil}) * x \mapsto z\}$
  b := x.next
$\{x \mapsto z \circledast (\mathtt{ls}(x, \mathsf{nil}) * \mathtt{ls}(a, \mathsf{nil}) * x \neq \mathsf{nil}) * x \mapsto z * b = z\}$
  x.next := a
$\{x \mapsto z \circledast (\mathtt{ls}(x, \mathsf{nil}) * \mathtt{ls}(a, \mathsf{nil}) * x \neq \mathsf{nil}) * x \mapsto a * b = z\}$
  a := x
$\{x \mapsto z \circledast (\mathtt{ls}(x, \mathsf{nil}) * \mathtt{ls}(a', \mathsf{nil}) * x \neq \mathsf{nil}) * x \mapsto a' * b = z * a = x\}$
  x := b
$\{x' \mapsto z \circledast (\mathtt{ls}(x', \mathsf{nil}) * \mathtt{ls}(a', \mathsf{nil}) * x' \neq \mathsf{nil}) * x' \mapsto a' * b = z * a = x' * x = b\}(=: Q)$
$\{\mathtt{ls}(x, \mathsf{nil}) * \mathtt{ls}(a, \mathsf{nil})\}\ (=: P)$

Example 4.8. Verifying the loop invariant of list copy:

$$\{\mathtt{ls}(u,x) * \mathtt{ls}(x,\mathsf{nil}) * \mathtt{ls}(r,s) * s \mapsto m\}\ (=: P)$$
$$\quad \mathtt{assume}(\mathsf{x} \neq \mathsf{nil})$$
$$\{\mathtt{ls}(u,x) * \mathtt{ls}(x,\mathsf{nil}) * \mathtt{ls}(r,s) * s \mapsto m * x \neq \mathsf{nil}\}$$
$$\quad \mathtt{malloc}(\mathsf{t})$$
$$\{\mathtt{ls}(u,x) * \mathtt{ls}(x,\mathsf{nil}) * \mathtt{ls}(r,s) * s \mapsto m' * x \neq \mathsf{nil} * t \mapsto m\}$$
$$\quad \mathsf{s.next} := \mathsf{t}$$
$$\{\mathtt{ls}(u,x) * \mathtt{ls}(x,\mathsf{nil}) * \mathtt{ls}(r,s) * s \mapsto t * x \neq \mathsf{nil} * t \mapsto m\}$$
$$\quad \mathsf{s} := \mathsf{t}$$
$$\{\mathtt{ls}(u,x) * \mathtt{ls}(x,\mathsf{nil}) * \mathtt{ls}(r,s') * s' \mapsto t * x \neq \mathsf{nil} * t \mapsto m * s = t\}$$
$$\quad \#\ \mathtt{materialization}$$
$$\{x \mapsto z \mathrel{-\!\!\circledast} (\mathtt{ls}(u,x) * \mathtt{ls}(x,\mathsf{nil}) * \mathtt{ls}(r,s') * s' \mapsto t * x \neq \mathsf{nil} * t \mapsto m * s = t) * x \mapsto z\}$$
$$\quad \mathsf{y} := \mathsf{x.next}$$
$$\{x \mapsto z \mathrel{-\!\!\circledast} (\mathtt{ls}(u,x) * \mathtt{ls}(x,\mathsf{nil}) * \mathtt{ls}(r,s') * s' \mapsto t * x \neq \mathsf{nil} * t \mapsto m * s = t) * x \mapsto z * y = z\}$$
$$\quad \mathsf{x} := \mathsf{y}$$
$$\{x' \mapsto z \mathrel{-\!\!\circledast} (\mathtt{ls}(u,x') * \mathtt{ls}(x',\mathsf{nil}) * \mathtt{ls}(r,s') * s' \mapsto t * x' \neq \mathsf{nil} * t \mapsto m * s = t) * x' \mapsto z * y = z * x = y\}(=: Q)$$
$$\{\mathtt{ls}(u,x) * \mathtt{ls}(x,\mathsf{nil}) * \mathtt{ls}(r,s) * s \mapsto m\}\ (=: P)$$

While our decision procedure can automatically discharge the entailments in both of the above examples, we give a short direct argument for the benefit of the reader for the entailment check of Example 4.7 (a direct argument could similarly be worked out for Example 4.8): We note that Q' simplifies to $Q'' = \{a \mapsto x \mathrel{-\!\!\circledast} (\mathtt{ls}(a,\mathsf{nil}) * \mathtt{ls}(a',\mathsf{nil})) * a \mapsto a'\}$. Every model (s,h) of Q'' must consist of a pointer $a \mapsto a'$, a list segment $\mathtt{ls}(a',\mathsf{nil})$ and a heap h' to which the pointer $a \mapsto x$ can be added in order to obtain the list segment $\mathtt{ls}(a,\mathsf{nil})$; by looking at the semantics of the list segment predicate we see that h' in fact must be the list segment $\mathtt{ls}(x,\mathsf{nil})$. Further, the pointer $a \mapsto a'$ can be composed with the list segment $\mathtt{ls}(a',\mathsf{nil})$ in order to obtain $\mathtt{ls}(a,\mathsf{nil})$.

5 NORMAL FORMS AND THE ABDUCTION PROBLEM

In this section, we introduce normal forms for the separation logic considered in this paper. We obtain normal forms from the insight that we can precisely describe every AMS by a formula, i.e., we can construct a formula such that all models of the formula abstract to the AMS for which the formula was constructed. Normal forms allow us to transform a formula into an equivalent *canonic representation*: We prove that the obtained normal form is equivalent to the original formula (Theorem 5.3). Moreover, we show that the normal form transformation is a closure operator (Theorem 5.4). We then discuss that the normal form transformation has applications to the abduction problem: We recall that the weakest solution to the abduction problem can be syntactically characterized by a formula that involves the magic wand. Normal forms then allow us to compute an explicit representation of the weakest solution.

$$\mathsf{AMS2SL}^m(\mathcal{A}) := \mathsf{aliasing}(\mathcal{A}) * \mathsf{graph}(\mathcal{A}) * \mathsf{garbage}(\mathcal{A})$$

$$\mathsf{aliasing}(\mathcal{A}) := \left(\circledast_{\mathbf{v}\in V, x, y\in\mathbf{v}} x = y\right) * \left(\circledast_{\mathbf{v},\mathbf{w}\in V, \mathbf{v}\neq\mathbf{w}} \max(\mathbf{v}) \neq \max(\mathbf{w})\right)$$

$$\mathsf{graph}(\mathcal{A}) := \left(\circledast_{E(\mathbf{v})=\langle \mathbf{v}', =1\rangle} \max(\mathbf{v}) \mapsto \max(\mathbf{v}')\right) * \left(\circledast_{E(\mathbf{v})=\langle \mathbf{v}', \geq 2\rangle} \mathtt{ls}_{\geq 2}(\max(\mathbf{v}), \max(\mathbf{v}'))\right)$$

$$\mathtt{ls}_{\geq 2}(x, y) := \mathtt{ls}(x, y) \wedge \neg(x \mapsto y)$$

$$\mathsf{garbage}(\mathcal{A}) := \begin{cases} \mathbf{emp}, & \text{if } \gamma = 0 \\ (\neg\mathbf{emp} \wedge \neg(\neg\mathbf{emp} * \neg\mathbf{emp}) \vee \neg\mathbf{emp} * \neg\mathbf{emp}) \wedge \mathsf{neg}(\mathcal{A}), & \text{if } \gamma = m = 1 \\ \left(*_{\gamma \text{ times}} \neg\mathbf{emp}\right) \wedge \neg\left(*_{\gamma+1 \text{ times}} \neg\mathbf{emp}\right) \wedge \mathsf{neg}(\mathcal{A}), & \text{if } 1 \leq \gamma < m \\ \left(*_{\gamma \text{ times}} \neg\mathbf{emp}\right) \wedge \mathsf{neg}(\mathcal{A}), & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathsf{neg}(\mathcal{A}) := & \bigwedge_{\mathbf{v},\mathbf{w}\in V, \varphi\in\{\max(\mathbf{v})\mapsto\max(\mathbf{w}), \mathtt{ls}(\max(\mathbf{v}),\max(\mathbf{w}))\}} \neg(\mathrm{t} * \varphi) \\ & \wedge \bigwedge_{R\in\rho, \mathbf{v}\in R} \mathsf{alloc}(\max(\mathbf{v})) \\ & \wedge \bigwedge_{R\in\rho, \mathbf{v},\mathbf{v}'\in R, \mathbf{v}\neq\mathbf{v}'} \neg(\mathsf{alloc}(\max(\mathbf{v})) * \mathsf{alloc}(\max(\mathbf{v}'))) \\ & \wedge \bigwedge_{\mathbf{v}\in V, \mathbf{v}\notin R \text{ for all } R\in\rho} \neg\mathsf{alloc}(\max(\mathbf{v})) \end{aligned}$$

$$\mathsf{alloc}(x) := \neg((x \mapsto \mathrm{nil}) \mathrel{-\!\!\circledast} \mathrm{t})$$

Fig. 13. The induced formula $\mathsf{AMS2SL}^m(\mathcal{A})$ of AMS $\mathcal{A} = \langle V, E, \rho, \gamma\rangle$ with $\gamma \leq m$.

Normal Forms. We lift the abstraction functions from stacks to sets of variables: Let $\mathbf{x} \subseteq \mathbf{Var}$ be a finite set of variables and $\varphi \in \mathbf{SL}$ be a formula with $\mathsf{fvs}(\varphi) \subseteq \mathbf{x}$. We set $\alpha_{\mathbf{x}}(\varphi) := \{\alpha_{\mathbf{s}}(\varphi) \mid \mathrm{dom}(s) = \mathbf{x}\}$ and $\mathrm{abst}_{\mathbf{x}}(\varphi) := \alpha_{\mathbf{x}}(\varphi) \cap \mathbf{AMS}_{\lceil\varphi\rceil,\mathbf{x}}$, where $\mathbf{AMS}_{k,\mathbf{x}} := \{\langle V, E, \rho, \gamma\rangle \in \mathbf{AMS} \mid \bigcup V = \mathbf{x} \text{ and } \gamma \leq k\}$. (We note that $\mathrm{abst}_{\mathbf{x}}(\varphi)$ is computable by the same argument as in the proof of Theorem 3.42.)

Definition 5.1 (Normal Form). Let $\mathsf{NF}_{\mathbf{x}}(\varphi) := \bigvee_{\mathcal{A}\in\mathrm{abst}_{\mathbf{x}}(\varphi)} \mathsf{AMS2SL}^{\lceil\varphi\rceil}(\mathcal{A})$ the *normal form* of φ, where $\mathsf{AMS2SL}^m(\mathcal{A})$ is defined as in Fig. 13.

The formula $\mathsf{AMS2SL}^m(\mathcal{A})$ represents a direct encoding of the AMS $\mathcal{A}$: $\mathsf{aliasing}(\mathcal{A})$ encodes the aliasing between the stack variables as implied by V; $\mathsf{graph}(\mathcal{A})$ encodes the points-to assertions and lists of length at least two corresponding to the edges E (the formula can be straight-forwardly adapted to trees as introduced in Section 3.8); $\mathsf{garbage}(\mathcal{A})$ encodes that there are precisely γ negative chunks (in case $\gamma < m$) or at least γ negative chunks (in case $\gamma = m$)[12], where the formula $\mathsf{neg}(\mathcal{A})$ ensures that these chunks are indeed negative (i.e., they don't satisfy a points-to or a list predicate) and contain exactly those allocated variables within some negative

[12]For the case $\gamma = m = 1$, the formula $\mathsf{garbage}(\mathcal{A})$ contains a seemingly superfluous case distinction between exactly one negative chunk and at least two negative chunks; this case distinction is a technicality needed to cover a special case in Theorem 5.4.

chunk as is specified by the negative allocation constraint ρ. We have the following result about the formula $\mathsf{AMS2SL}^m(\mathcal{A})$:

Lemma 5.2. *Let $\mathcal{A} = \langle V, E, \rho, \gamma \rangle$ be an AMS with $\gamma \leq m$, and let (s,h) be a stack-heap pair. Then, we have that $(s,h) \models^{\mathsf{st}} \mathsf{AMS2SL}^m(\mathcal{A})$ iff $\mathsf{ams}(s,h) = \langle V, E, \rho, \gamma' \rangle$, with $\gamma' = \gamma$ for $\gamma < m$, and $\gamma' \geq \gamma$ for $\gamma = m$.*

Proof. We have that $(s,h) \models^{\mathsf{st}} \mathsf{aliasing}(\mathcal{A})$ iff the equivalence classes induced by s agree with the equivalence classes V. We have that $(s,h) \models^{\mathsf{st}} \mathsf{graph}(\mathcal{A})$ iff the positive chunks of (s,h) are precisely the ones specified by E (we note that the formula $\mathtt{ls}_{\geq 2}$ indeed ensures that the list segments corresponding to this formula have indeed length at least two). We have that $(s,h) \models^{\mathsf{st}} \mathsf{garbage}(\mathcal{A})$ iff (s,h) contains exactly γ negative chunks (in case $\gamma < m$) resp. at least γ negative chunks (in case $\gamma = m$); the formula $\mathsf{neg}(\mathcal{A})$ ensures that all chunks are indeed negative, i.e., that there is no chunk satisfying a formula $\max(\mathbf{v}) \mapsto \max(\mathbf{w})$ or $\mathtt{ls}(\max(\mathbf{v}), \max(\mathbf{w}))$ and the allocated variables correspond to the sets specified by ρ (i.e., for every $R \in \rho$ there is a chunk that allocates the variables in R, for every $R \in \rho$ the variables in R cannot be allocated in different chunks, and all variables not in some $R \in \rho$ are not allocated). □

The normal form of a formula φ is then obtained by taking the disjunction over all formulas $\mathsf{AMS2SL}^m(\mathcal{A})$ for all AMS $\mathcal{A} \in \alpha_{\mathbf{x}}(\varphi)$ that result from abstracting models of φ. Intuitively, the formulas $\mathsf{AMS2SL}^m(\mathcal{A})$ partition the models of φ (recall that the Refinement Theorem states that models that abstract to the same AMS satisfy the same formulas). We now state the normal form of a formula is equivalent to the formula for which the normal form was constructed:

Theorem 5.3 (Equivalence). *$\mathsf{NF}_{\mathbf{x}}(\varphi) \models^{\mathsf{st}}_{\mathbf{x}} \varphi$ and $\varphi \models^{\mathsf{st}}_{\mathbf{x}} \mathsf{NF}_{\mathbf{x}}(\varphi)$.*

Proof. $\varphi \models^{\mathsf{st}}_{\mathbf{x}} \mathsf{NF}_{\mathbf{x}}(\varphi)$: Assume $(s,h) \models^{\mathsf{st}} \varphi$ for some stack-heap pair (s,h). W.l.o.g. we can assume that (s,h) has at most $\lceil \varphi \rceil$ negative chunks (otherwise we can choose a stack-heap pair that has exactly $\lceil \varphi \rceil$ negative chunks; by Theorem 3.37 this model still satisfies formula φ). We consider the AMS $\mathcal{A} = \mathsf{ams}(s,h)$. With $(s,h) \models^{\mathsf{st}} \varphi$, we get $\mathcal{A} \in \mathsf{abst}_{\mathbf{x}}(\varphi)$. By Lemma 5.2, we have $(s,h) \models^{\mathsf{st}} \mathsf{AMS2SL}^{\lceil \varphi \rceil}(\mathcal{A})$. Because of $\mathcal{A} \in \mathsf{abst}_{\mathbf{x}}(\varphi)$ we get that $(s,h) \models^{\mathsf{st}} \mathsf{NF}_{\mathbf{x}}(\varphi)$.

$\mathsf{NF}_{\mathbf{x}}(\varphi) \models^{\mathsf{st}}_{\mathbf{x}} \varphi$: Assume $(s,h) \models^{\mathsf{st}} \mathsf{NF}_{\mathbf{x}}(\varphi)$ for some stack-heap pair (s,h). Hence, there is some AMS $\langle V, E, \rho, \gamma \rangle = \mathcal{A} \in \mathsf{abst}_{\mathbf{x}}(\varphi)$ with $(s,h) \models^{\mathsf{st}} \mathsf{AMS2SL}^{\lceil \varphi \rceil}(\mathcal{A})$. By Lemma 5.2 we have that $\mathsf{ams}(s,h) = \langle V, E, \rho, \gamma' \rangle$, with $\gamma' = \gamma$ for $\gamma < m$, and $\gamma' \geq \gamma$ for $\gamma = m$. Then, we can conclude that $(s,h) \models^{\mathsf{st}} \varphi$ (using Theorem 3.19 for $\gamma < m$ and Theorem 3.37 for $\gamma = m$). □

We now state that the normal form transformation is a closure operator:

Theorem 5.4 (Closure Operator). *We have $\mathsf{NF}_{\mathbf{x}}(\mathsf{NF}_{\mathbf{x}}(\varphi)) = \mathsf{NF}_{\mathbf{x}}(\varphi)$ and $\lceil \mathsf{NF}_{\mathbf{x}}(\varphi) \rceil \leq \max\{2, \lceil \varphi \rceil\}$.*

Proof. By Theorem 5.3 we have that $\mathsf{NF}_{\mathbf{x}}(\varphi)$ and φ are equivalent. Hence, we have that their models abstract to the same AMSs, i.e.,

$$\alpha_{\mathbf{x}}(\mathsf{NF}_{\mathbf{x}}(\varphi)) = \alpha_{\mathbf{x}}(\varphi) \quad (*).$$

We now analyze the chunk size of the formula $\mathsf{AMS2SL}^m(\mathcal{A})$ for some $\langle V, E, \rho, \gamma \rangle = \mathcal{A}$ with $\gamma \leq m$. We observe that $\lceil \mathsf{aliasing}(\mathcal{A}) \rceil = \lceil \mathsf{graph}(\mathcal{A}) \rceil = 0$, $\lceil \mathsf{neg}(\mathcal{A}) \rceil = 2$ and $\lceil \mathsf{garbage}(\mathcal{A}) \rceil = \max\{2, \gamma + 1\}$, for $\gamma < m$, and $\lceil \mathsf{garbage}(\mathcal{A}) \rceil = \max\{2, \gamma\}$, for $\gamma = m$.

From these observations and (*), we obtain

$$\lceil \mathsf{NF}_{\mathbf{x}}(\varphi) \rceil \leq \max\{2, \lceil \varphi \rceil\}.$$

We now observe that

$$\alpha_{\mathbf{x}}(\mathsf{NF}_{\mathbf{x}}(\varphi)) \cap \mathbf{AMS}_{\lceil \mathsf{NF}_{\mathbf{x}}(\varphi) \rceil, \mathbf{x}} = \alpha_{\mathbf{x}}(\varphi) \cap \mathbf{AMS}_{\lceil \varphi \rceil, \mathbf{x}},$$

for $\lceil\varphi\rceil = 0$ and $\lceil\varphi\rceil \geq 2$. Moreover, we have

$$\alpha_{\mathbf{x}}(\mathsf{NF}_{\mathbf{x}}(\varphi)) \cap \mathbf{AMS}_{\lceil \mathsf{NF}_{\mathbf{x}}(\varphi)\rceil,\mathbf{x}} = \alpha_{\mathbf{x}}(\varphi) \cap \mathbf{AMS}_{2,\mathbf{x}},$$

for $\lceil\varphi\rceil = 1$. We thus get that

$$\bigvee_{\mathcal{A}\in\mathsf{abst}_{\mathbf{x}}(\mathsf{NF}_{\mathbf{x}}(\varphi))} \mathsf{AMS2SL}^{\lceil \mathsf{NF}_{\mathbf{x}}(\varphi)\rceil}(\mathcal{A}) = \bigvee_{\mathcal{A}\in\mathsf{abst}_{\mathbf{x}}(\varphi)} \mathsf{AMS2SL}^{\lceil\varphi\rceil}(\mathcal{A})$$

(up to logical equivalence), where the equation holds for $\lceil\varphi\rceil = 1$ because the case distinction for $\gamma = m = 1$ in $\mathsf{AMS2SL}^m(\mathcal{A})$ amounts to an explicit chunk bound lifting from 1 to 2. □

The abduction problem. We recall the following generalization of the entailment problem: The *abduction problem* is to replace the question mark in the entailment $\varphi * [?] \models^{\mathsf{st}}_{\mathbf{x}} \psi$ by a formula such that the entailment becomes true. This problem is central for obtaining a scalable program analyzer as discussed in [Calcagno et al. 2011] [13]. The abduction problem does in general not have a unique solution. Following [Calcagno et al. 2011], we thus consider optimization versions of the abduction problem, looking for *logically weakest* and *spatially minimal* solutions:

Definition 5.5. Let $\varphi, \psi \in \mathbf{SL}$ and $\mathbf{x} \subseteq \mathbf{Var}$ be a finite set of variables. A formula ζ is the *weakest* solution to the abduction problem $\varphi * [?] \models^{\mathsf{st}}_{\mathbf{x}} \psi$ if it holds for all abduction solutions ζ' that $\zeta' \models^{\mathsf{st}}_{\mathbf{x}} \zeta$. An abduction solution is ζ *minimal*, if there is no abduction solution ζ' with $\zeta \models^{\mathsf{st}}_{\mathbf{x}} \zeta' * (\neg\mathbf{emp})$.

Lemma 5.6. *Let φ, ψ be formulas and let $\mathbf{x} \subseteq \mathbf{Var}$ be a finite set of variables. Then, 1) the weakest solution to the abduction problem $\varphi * [?] \models^{\mathsf{st}}_{\mathbf{x}} \psi$ is given by the formula $\varphi \mathrel{-\!\!*} \psi$, and the 2) weakest minimal solution is given by the formula $\varphi \mathrel{-\!\!*} \psi \wedge \neg((\varphi \mathrel{-\!\!*} \psi) * \neg\mathbf{emp})$.*

Proof. 1) follows directly from the definition of the abduction problem and the semantics of $\mathrel{-\!\!*}$.

For 2), we introduce the shorthand $\zeta := \varphi \mathrel{-\!\!*} \psi \wedge \neg((\varphi \mathrel{-\!\!*} \psi) * \neg\mathbf{emp})$. We note that $\zeta \models^{\mathsf{st}}_{\mathbf{x}} \varphi \mathrel{-\!\!*} \psi$, and hence ζ is a solution to the abduction problem by 1). Assume further that there is an abduction solution ζ' with $\zeta \models^{\mathsf{st}}_{\mathbf{x}} \zeta' * (\neg\mathbf{emp})$. By 1) we have $\zeta' \models^{\mathsf{st}}_{\mathbf{x}} \varphi \mathrel{-\!\!*} \psi$. Hence, $\zeta \models^{\mathsf{st}}_{\mathbf{x}} \varphi \mathrel{-\!\!*} \psi * (\neg\mathbf{emp})$. However, this contradicts $\zeta \models^{\mathsf{st}}_{\mathbf{x}} \neg((\varphi \mathrel{-\!\!*} \psi) * \neg\mathbf{emp})$. Thus, ζ is minimal. Now, consider another minimal solution ζ' to the abduction problem. By 1), we have $\zeta' \models^{\mathsf{st}}_{\mathbf{x}} \varphi \mathrel{-\!\!*} \psi$. Because ζ' is minimal, we have as above that $\zeta' \models^{\mathsf{st}}_{\mathbf{x}} \neg((\varphi \mathrel{-\!\!*} \psi) * \neg\mathbf{emp})$. Hence, $\zeta' \models^{\mathsf{st}}_{\mathbf{x}} \zeta$. Thus, ζ is the weakest minimal solution to the abduction problem. □

We now explain how normal forms have applications to the abduction problem. According to Lemma 5.6, the best solutions to the abduction problem are given by the formulas $\zeta := \varphi \mathrel{-\!\!*} \psi$ and $\zeta' := \varphi \mathrel{-\!\!*} \psi \wedge \neg((\varphi \mathrel{-\!\!*} \psi) * \neg\mathbf{emp})$. While it is an important result that the existence of these solutions is guaranteed, we do a-priori have no means to *compute* an explicit representation of these solutions nor to further analyze their structure. However, the normal form operator allows us to obtain the explicit representations $\mathsf{NF}_{\mathbf{x}}(\zeta)$ and $\mathsf{NF}_{\mathbf{x}}(\zeta')$. We believe that using these explicit representations in a program analyzer or studying their properties is an interesting topic for further research. Here, we establish one concrete result on solutions to the abduction problem based on normal forms:

We can compute the weakest resp. the weakest minimal solution to the abduction problem for the positive fragment: Given $\zeta = \varphi \mathrel{-\!\!*} \psi$ resp. $\zeta = \varphi \mathrel{-\!\!*} \psi \wedge \neg((\varphi \mathrel{-\!\!*} \psi) * \neg\mathbf{emp})$, we consider the formula $\bigvee_{\mathcal{A}\in\mathsf{abst}_{\mathbf{x}}(\zeta),\, \mathcal{A} \text{ is garbage-free}} \mathsf{AMS2SL}^{\lceil\zeta\rceil}(\mathcal{A})$. Indeed, this formula is the weakest resp. the weakest minimal solution to the abduction problem from the positive fragment, in case we are willing to consider a slight extension of the positive fragment (observe that among the sub-formulas of $\mathsf{aliasing}(\mathcal{A})$ and $\mathsf{graph}(\mathcal{A})$ only the formula $\mathtt{ls}_{\geq 2}$ is negative): one could either extend the

[13] While the program analyzer proposed in [Calcagno et al. 2011] employs *bi-abductive* reasoning, the suggested bi-abductive procedure in fact proceeds in two separate abduction and frame-inference steps, where the main technical challenge is the abduction step, as frame inference can be incorporated into entailment checking. We believe that the situation for SSL is similar, i.e., solving abduction is the key to implementing a bi-abductive prover for SSL; hence, our focus on the abduction problem.

positive fragment by allowing guarded negation[14] or add a new spatial atom $\mathtt{ls}_{\geq 2}(x, y)$ to SSL, with the semantics that $\mathtt{ls}_{\geq 2}(x, y)$ holds in a model iff the model is a list segment of length at least 2 from x to y; Sections 2 and 3 could be accordingly extended by this predicate; we can then simplify the formula $\mathsf{graph}(\mathcal{A})$ in $\mathsf{AMS2SL}^m(\mathcal{A})$ by directly translating edges $E(\mathbf{v}) = \langle \mathbf{v}', \geq 2 \rangle$ to the atom $\mathtt{ls}_{\geq 2}(\max(\mathbf{v}), \max(\mathbf{v}'))$.

6 CONCLUSION

We have shown that the satisfiability problem for "strong" separation logic with lists and trees is in the same complexity class as the satisfiability problem for standard "weak" separation logic without any data structures: PSPACE-complete. This is in stark contrast to the undecidability result for standard (weak) SL semantics, as shown in [Demri et al. 2018].

We have demonstrated the potential of SSL for program verification: 1) We have provided symbolic execution rules that, in conjunction with our result on the decidability of entailment, can be used for fully-automatically discharging verification conditions. 2) We have discussed how to compute explicit representations to optimal solutions of the abduction problem. This constitutes the first work that addresses the abduction problem for a separation logic closed under Boolean operators and the magic wand.

We consider the above results just the first steps in examining strong-separation logic, motivated by the desire to circumvent the undecidability result of [Demri et al. 2018]. Future work is concerned with the practical evaluation of our decision procedures, with extending the symbolic execution calculus to a full Hoare logic as well as extending the results of this paper to richer separation logics (SL) such as SL with nested data structures or SL with limited support for arithmetic reasoning.

REFERENCES

Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max I. Kanovich, and Joël Ouaknine. 2014. Foundations for Decision Problems in Separation Logic with General Inductive Predicates. In *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Springer Berlin Heidelberg, 411–425. https://doi.org/10.1007/978-3-642-54830-7_27

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

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A Decidable Fragment of Separation Logic. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*. Springer, 97–109. https://doi.org/10.1007/978-3-540-30538-5_9

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic Execution with Separation Logic. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*. 52–68. https://doi.org/10.1007/11575467_5

Josh Berdine, Byron Cook, and Samin Ishtiaq. 2011. SLAyer: Memory Safety for Systems-Level Code. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 178–183. https://doi.org/10.1007/978-3-642-22110-1_15

Stefan Blom and Marieke Huisman. 2015. Witnessing the elimination of magic wands. *International Journal on Software Tools for Technology Transfer* 17, 6 (2015), 757–781. https://doi.org/10.1007/s10009-015-0372-3

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. 259–270. https://doi.org/10.1145/1040305.1040327

Rémi Brochenin, Stéphane Demri, and Etienne Lozes. 2012. On the almighty wand. *Information and Computation* 211 (2012), 106 – 137. https://doi.org/10.1016/j.ic.2011.12.003

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, Cham, 3–11.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (Dec. 2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[14] One can add guarded negation to our separation logic by extending the grammar of Figures 2 with $\varphi ::= \cdots \mid \varphi \wedge \neg\varphi$. All results about the positive fragment continue to hold when guarded negation is added to the positive fragment.

C. Calcagno, P.W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*. 366–378. https://doi.org/10.1109/LICS.2007.30

Cristiano Calcagno, Hongseok Yang, and PeterW. O'Hearn. 2001. Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, Ramesh Hariharan, V. Vinay, and Madhavan Mukund (Eds.). Lecture Notes in Computer Science, Vol. 2245. Springer Berlin Heidelberg, 108–119. http://dx.doi.org/10.1007/3-540-45294-X_10

Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. 2011. Tractable Reasoning in a Fragment of Separation Logic. In *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*. 235–249. https://doi.org/10.1007/978-3-642-23217-6_16

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 207–231.

Stéphane Demri and Morgan Deters. 2014. Expressive Completeness of Separation Logic with Two Variables and No Separating Conjunction. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (Vienna, Austria) *(CSL-LICS '14)*. ACM, New York, NY, USA, 37:1–37:10. https://doi.org/10.1145/2603088.2603142

Stéphane Demri, Didier Galmiche, Dominique Larchey-Wendling, and Daniel Méry. 2014. Separation Logic with One Quantified Variable. In *Computer Science - Theory and Applications*, EdwardA. Hirsch, SergeiO. Kuznetsov, Jean-Éric Pin, and NikolayK. Vereshchagin (Eds.). Lecture Notes in Computer Science, Vol. 8476. Springer International Publishing, 125–138. http://dx.doi.org/10.1007/978-3-319-06686-8_10

Stéphane Demri, Étienne Lozes, and Alessio Mansutti. 2018. The Effects of Adding Reachability Predicates in Propositional Separation Logic. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 476–493.

Kamil Dudka, Petr Peringer, and Tomás Vojnar. 2011. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 372–378. https://doi.org/10.1007/978-3-642-22110-1_29

Mnacho Echenim, Radu Iosif, and Nicolas Peltier. 2019. The Bernays-Schönfinkel-Ramsey Class of Separation Logic on Arbitrary Domains. In *Foundations of Software Science and Computation Structures*, Mikołaj Bojańczyk and Alex Simpson (Eds.). Springer International Publishing, Cham, 242–259.

Frantisek Farka, Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2021. On algebraic abstractions for concurrent separation logics. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32.

Nikos Gorogiannis, Max Kanovich, and Peter W. O'Hearn. 2011. The Complexity of Abduction for Separated Heap Abstractions. In *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, Eran Yahav (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 25–42. https://doi.org/10.1007/978-3-642-23702-7_7

Xincai Gu, Taolue Chen, and Zhilin Wu. 2016. A Complete Decision Procedure for Linearly Compositional Separation Logic with Data Constraints. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*. 532–549. https://doi.org/10.1007/978-3-319-40229-1_36

Radu Iosif, Adam Rogalewicz, and Jirí Simácek. 2013. The Tree Width of Separation Logic with Recursive Definitions. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*. 21–38. https://doi.org/10.1007/978-3-642-38574-2_2

Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. 2014. Deciding Entailments in Inductive Separation Logic with Tree Automata. In *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*. 201–218. https://doi.org/10.1007/978-3-319-11936-6_15

Samin S. Ishtiaq and Peter W. O'Hearn. 2001a. BI as an Assertion Language for Mutable Data Structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*. 14–26.

Samin S. Ishtiaq and Peter W. O'Hearn. 2001b. BI as an Assertion Language for Mutable Data Structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, Chris Hankin and Dave Schmidt (Eds.). ACM, 14–26.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20.

Jens Katelaan, Dejan Jovanović, and Georg Weissenbacher. 2018. A Separation Logic with Data: Small Models and Automation. In *Automated Reasoning*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.). Springer International Publishing, Cham, 455–471.

Jens Katelaan, Christoph Matheja, and Florian Zuleger. 2019. Effective Entailment Checking for Separation Logic with Inductive Definitions. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*. 319–336. https://doi.org/10.1007/978-3-030-17465-1_18

Jens Katelaan and Florian Zuleger. 2020. Beyond Symbolic Heaps: Deciding Separation Logic With Inductive Definitions. In *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020 (EPiC Series in Computing, Vol. 73)*, Elvira Albert and Laura Kovács (Eds.). EasyChair, 390–408.

Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the flow: compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.* 2, POPL (2018), 37:1–37:31.

Siddharth Krishna, Alexander J. Summers, and Thomas Wies. 2019. Local Reasoning for Global Graph Properties. *CoRR* abs/1911.08632 (2019). arXiv:1911.08632 http://arxiv.org/abs/1911.08632

Siddharth Krishna, Alexander J. Summers, and Thomas Wies. 2020. Local Reasoning for Global Graph Properties. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 308–335.

Quang Loc Le, Makoto Tatsuta, Jun Sun, and Wei-Ngan Chin. 2017. A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. 495–517. https://doi.org/10.1007/978-3-319-63390-9_26

P Madhusudan, Gennaro Parlato, and Xiaokang Qiu. 2011. Decidable Logics Combining Heap Structures and Data. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 611–622. https://doi.org/10.1145/1926385.1926455

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

Jens Pagel, Christoph Matheja, and Florian Zuleger. 2020. Complete Entailment Checking for Separation Logic with Inductive Definitions. *CoRR* abs/2002.01202 (2020). arXiv:2002.01202 https://arxiv.org/abs/2002.01202

Jens Pagel and Florian Zuleger. 2021. Strong-Separation Logic. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 664–692.

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2013. Separation Logic Modulo Theories. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*. 90–106. https://doi.org/10.1007/978-3-319-03542-0_7

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Lecture Notes in Computer Science, Vol. 8044. Springer Berlin Heidelberg, 773–789.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. Automating Separation Logic with Trees and Data. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 711–728. https://doi.org/10.1007/978-3-319-08867-9_47

Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Parthasarathy Madhusudan. 2013. Natural proofs for structure, data, and separation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 231–242. https://doi.org/10.1145/2491956.2462169

Andrew Reynolds, Radu Iosif, and Cristina Serban. 2017. Reasoning in the Bernays-Schönfinkel-Ramsey Fragment of Separation Logic. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*. 462–482. https://doi.org/10.1007/978-3-319-52234-0_25

Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. 2016. A Decision Procedure for Separation Logic in SMT. In *Automated Technology for Verification and Analysis*, Cyrille Artho, Axel Legay, and Doron Peled (Eds.). Springer International Publishing, Cham, 244–261.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

Malte Schwerhoff and Alexander J. Summers. 2015. Lightweight Support for Magic Wands in an Automatic Verifier. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. 614–638. https://doi.org/10.4230/LIPIcs.ECOOP.2015.614

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 77–87.

Makoto Tatsuta and Daisuke Kimura. 2015. Separation Logic with Monadic Inductive Definitions and Implicit Existentials. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*. 69–89.

https://doi.org/10.1007/978-3-319-26529-2_5