Beyond Symbolic Heaps: Deciding Separation Logic with Inductive Definitions

Jens Pagel and Florian Zuleger

1 TU Wien, Vienna, Austria
   pagel@forsyte.at
2 TU Wien, Vienna, Austria
   zuleger@forsyte.at

Abstract
Symbolic-heap separation logic with inductive definitions is a popular formalism for reasoning about heap-manipulating programs. The fragment SLID\textsubscript{btw} introduced by Iosif, Rogalewicz and Simacek, is one of the most expressive fragments with a decidable entailment problem. In recent work, we improved on the original decidability proof by providing a direct model-theoretic construction, obtaining a 2-Exptime upper bound. In this paper, we investigate separation logics built on top of the inductive definitions from SLID\textsubscript{btw}, i.e., logics that feature the standard Boolean and separation-logic operators. We give an almost tight delineation between decidability and undecidability. We establish the decidability of the satisfiability problem (in 2-Exptime) of a separation logic with conjunction, disjunction, separating conjunction and guarded forms of negation, magic wand, and septraction. We show that any further generalization leads to undecidability (under mild assumptions).

1 Introduction

Separation logics (SL) are a popular formalism for reasoning about the usage of dynamic resources [30, 25], and have been widely used in static analysis [11, 10], automated verification [5, 7, 13, 19, 29, 24, 31], and interactive verification [2, 21]. SLs extend first-order logic with so-called separating connectives, most importantly the separating conjunction $\ast$, in order to be able to split resources and reason about program parts in isolation.

In this article, we study SLs for reasoning about the heap structures arising in heap-manipulating programs. Here, the SL formula $\phi_1 \ast \phi_2$ specifies that a heap can be split into two disjoint sub-heaps that satisfy $\phi_1$ and $\phi_2$. In addition to the separating connectives, SLs for the heap usually feature points-to predicates and predicates for unbounded data-structures such as lists, trees, etc. While satisfiability and related problems for separation logic are undecidable in general, many decidable fragments have been proposed. Most decidability results have been obtained for the symbolic-heap fragment [6, 4, 16, 9, 17, 32, 20, 33]. Symbolic heaps are separation-logic formulas in which atomic predicates can only be combined with the separating conjunction; no other separating connectives or Boolean connectives are allowed. The decidability results fall into two categories: Logics with built-in predicates—usually for expressing linked lists and/or trees [4, 14, 27, 28, 22]; and logics with user-defined inductive definitions (SLID).
Beyond Symbolic Heaps: Deciding Separation Logic with Inductive Definitions

Pagel and Zuleger

In SLID, users specify the shape of data structures by a set of recursive definitions. For example, the definitions on the right define segments of singly-linked lists of odd length, even length, and length at least one. Importantly, the formulae on the right-hand side of $\Leftarrow$ in the SLID definitions are restricted to be symbolic heaps, i.e., formulae built from atomic predicates (here $(\cdot \mapsto \cdot)$, odd, even, and lseg) and the separating conjunction $\star$. The satisfiability problem for SLID is decidable in exponential time [9].

The entailment problem, which is crucial for Hoare-style deductive verification, is, however, undecidable in general [1]. Consequently, restricted fragments of SLID definitions have been studied. While it is natural to consider restrictions to trees [17, 32, 33], it is possible to obtain decidability results for more expressive logics. Iosif et al. [16] proved the decidability of a particularly expressive fragment of SLID. The fragment, for example, supports the definition of binary trees whose leaves form a linked list (which might be used to implement a sorted set data structure). Following [16], we denote this logic by SLID_{btw}, where the subscript alludes to the fact that all models of SLID_{btw} formulas (when viewed as graphs) are of bounded treewidth (BTW). The original decidability result [16] was obtained by a reduction to the satisfiability problem of monadic second-order logic over graphs of BTW. Deciding the satisfiability via this reduction would involve a blowup of several exponentials and therefore seems impractical to implement [17]. In recent work [26], we improved on the original decidability proof by providing a direct model-theoretic construction, obtaining a 2-Exptime upper bound and promising experimental results. A matching lower bound has been announced in [15].

In this article, we study SLs that go beyond the symbolic-heap fragment. We consider SLs that feature the standard Boolean and separation-logic connectives in addition to atomic SLID predicates. In these logics, the predicates (such as odd and even above) must still be defined using symbolic heaps, but the SL formulae that are built on top of atomic predicates can use other classical and separating connectives. For example, our decidability result allows discharging entailment queries such as $(lseg(x, y) \land (\text{even}(y, \text{nil}) \rightarrow \text{odd}(x, \text{nil}))) \land \neg (x \mapsto y) \models \phi \odot \text{odd}(x, y)$. Such queries naturally arise in Hoare-style verification, where conditions require reasoning about the Boolean connectives ($\land$, $\lor$, and $\neg$) and weakest pre-condition computation requires reasoning about the magic-wand $\rightarrow$ [18, 30]. In this paper, we are concerned with the fundamental question what is decidable about an SL that is built on top of the inductive definitions from SLID_{btw}.

Inspired by work on guarded first-order logic [3], we propose a guarded fragment of separation logic, SL_{btw}$. Formulas in SL_{btw} may combine user-defined predicates from SLID_{btw} with the separating conjunction $\star$, classical conjunction $\land$ and classical $\lor$. Moreover, we allow negation $\neg$, magic wand $\rightarrow$ and septraction $\ominus$ [8] to appear in formulas of the form $\phi \land \neg \psi$, $\phi \land (\psi \rightarrow \zeta)$, and $\phi \land (\psi \ominus \zeta)$. Here, $\phi$ acts as a guard on the operators; hence the name.

Contributions. Our two main technical results are as follows:

- We show that further generalizing SL_{btw} by allowing any one of the three operators $\neg$, $\rightarrow$ and $\ominus$ in an unguarded form yields an undecidable logic.

- We then prove that our approach for satisfiability- and entailment checking from [26] can be lifted to SL_{btw}, obtaining a 2-Exptime decision procedure for guarded separation logic. We thus obtain the first decidability result for a separation logic with support both for unbounded data structures and for the magic wand.
We thus obtain an (almost perfectly) tight delineation between what is decidable and undecidable in SLs with inductive definitions from SLID$_{btw}$: the only cases left open are whether the undecidability results can be tightened to proper subsets of $\{\land, *, \top\}, \{\land, *, \lnot\}$ and $\{\land, *, \lnot\}$.

**Outline.** We introduce SL with inductive definitions and its guarded fragment SL$_{gbw}^g$ in Section 2. We show that all extensions of SL$_{gbw}^g$ are undecidable in Section 3. In Section 4, we introduce the abstraction that underlies our decision procedure for SL$_{gbw}^g$. We present the decidability result in Section 5 and conclude in Section 6. All proofs omitted in Sections 4 and 5 are in our technical report [26].

## 2 Separation Logic with Inductive Definitions

**Preliminaries.** We denote by $|X|$ the cardinality of the set $X$. Let $f$ be a (partial) function. Then $\text{dom}(f)$ and $\text{img}(f)$ denote the domain and image of $f$, respectively. We frequently use set notation to define and reason about partial functions. For example, $f := \{x_1 \mapsto y_1, \ldots, x_k \mapsto y_k\}$ is the partial function that maps $x_i$ to $y_i$, $1 \leq i \leq k$, and is undefined on all other values: $f \cup g$ is the union of partial functions $f$ and $g$; and $f \subseteq g$ means $f(x) = g(x)$ for all $x \in \text{dom}(f)$. Sets and ordered sequences are denoted in boldface, e.g., $x$. To list the elements of a sequence, we write $(x_1, \ldots, x_k)$. We shorten $(x)$ to $x$ to reduce clutter. The empty sequence is $\epsilon$, the concatenation of $x$ and $y$ is $x \cdot y$. We lift functions to sequences, i.e., $f((x_1, \ldots, x_k)) := (f(x_1), \ldots, f(x_k))$. The grammar in Fig. 1 defines three variants of separation logic (SL) with inductive definitions: guarded SL, formulas of the form $\phi_g$, collected in the set SL$_{gbw}^g$; quantifier-free SL, formulas of the form $\phi_{qf}$, collected in SL$_{gbw}^{qf}$; and existentially-quantified symbolic heaps, collected in SLH$^3$.

In Fig. 1, $\text{pred} \in \text{Preds}$ is a predicate identifier, $x, y \in \text{Var}$ are variables, and $x, y, z \in \text{Var}^*$ are (possibly empty) sequences of variables with $|x| = \text{ar}(\text{pred})$. The first line of Fig. 1 defines the atomic formulas, $\tau$, common to all SL variants studied in this article. $\text{emp}$ is the empty heap, $x \mapsto y$ asserts that $x$ points to $y$, $x \approx y$ asserts the equality between variables $x$ and $y$, and $x \not\approx y$ asserts the disequality of $x$ and $y$. Guarded formulas, $\phi_g$, are built from atomic formulas using the separating conjunction $\ast$, conjunction $\land$, disjunction $\lor$, guarded negation $\phi_g \land \lnot \phi_g$, guarded sepration $\phi_g \land (\phi_g \oplus \phi_g)$, and guarded magic wands $\phi_g \land (\phi_g \Rightarrow \phi_g)$. In quantifier-free formulas, $\phi_{qf}$, all operators may occur unguarded and we include an additional atom $\top$ representing true. Finally, $\phi_{sh}$ formulas are existentially-quantified symbolic heaps. We use $\ast$ instead of $\land$ in the pure constraint, $\Pi$, because in our semantics, (dis)equalities only hold in empty heaps.

\[
\begin{align*}
\tau &:= \text{emp} \mid x \mapsto y \mid \text{pred}(x) \mid x \approx y \mid x \not\approx y \\
\phi_g &:= \tau \mid \phi_g \ast \phi_g \mid \phi_g \land \phi_g \mid \phi_g \lor \phi_g \mid \phi_g \land \lnot \phi_g \mid \phi_g \land (\phi_g \oplus \phi_g) \mid \phi_g \land (\phi_g \Rightarrow \phi_g) \\
\phi_{qf} &:= \tau \mid \phi_{qf} \ast \phi_{qf} \mid \phi_{qf} \land \lnot \phi_{qf} \mid \phi_{qf} \lor \phi_{qf} \mid \phi_{qf} \land \lnot \phi_{qf} \mid \phi_{qf} \lor \phi_{qf} \mid \phi_{qf} \land \lnot \phi_{qf} \\
\phi_{sh} &:= \exists e. (x_1 \mapsto y_1) \ast \cdots \ast (x_k \mapsto y_k) \ast \text{pred}_1(z_1) \ast \cdots \ast \text{pred}_m(z_m) \ast \Pi, \\
\text{where } &\Pi := a_1 \approx b_1 \ast \cdots \ast a_m \approx b_m \ast c_1 \not\approx d_1 \ast \cdots \ast c_n \not\approx d_n
\end{align*}
\]

Figure 1: The syntax of the separation-logic fragments studied in this article: Guarded formulas $\phi_g$; quantifier-free formulas $\phi_{qf}$; and existentially-quantified symbolic heaps, $\phi_{sh}$.
\[ \text{tll}(x_1, x_2, x_3) \iff x_1 \mapsto (\text{nil}, \text{nil}, x_3) \ast x_1 = x_2 \]
\[ \text{tll}(x_1, x_2, x_3) \iff \exists l, r, m. \ x_1 \mapsto (l, r, \text{nil}) \ast \text{tll}(l, x_2, m) \ast \text{tll}(r, m, x_3) \]

(a) The definition of the \text{tll} predicate representing trees with linked leaves.

(b) A model of \text{tll}(x_1, x_2, x_3).

Figure 2: A system of inductive definition (SID) defining trees with linked leaves.

We denote by $\text{SL}(\cdot_1, \ldots, \cdot_k)$ the restriction of $\text{SL}_{\text{sbv}}$ to formulas built from $\tau$ and the additional symbols and operators $\cdot_1, \ldots, \cdot_k$. For example, $\text{SL}(\land, \ast, t)$ is the SL in which formulas are built from atomic predicates $\tau$, additional predicate $t$ and binary operators $\ast$, $\land$.

**Additional notation.** We write $\phi([x_1, \ldots, x_k] / [y_1, \ldots, y_k])$ for the formula obtained from $\phi$ by instantiating every occurrence of $x_i$ with $y_i$. We sometimes write $\ast_{1 \leq i \leq n} \phi_i$ to denote $\phi_1 \ast \cdots \ast \phi_n$. For $n = 0$, this expression evaluates to the neutral element of $\ast$, $\text{emp}$. For an atomic formula $\tau$ and a formula $\phi$, $\tau \in \phi$ denotes that $\tau$ occurs in $\phi$. The size of a formula $\phi$, $|\phi|$, is the sum of the number of atoms, the number of unary operators and binary operators, and the number of quantifiers in the formula. Finally, $\text{fvars}(\phi)$ is the set of all free variables in $\phi$.

**Inductive definitions.** Predicates are defined by a system of inductive definitions (SID). An SID is a finite set $\Phi$ of rules of the form $\text{pred}(\cdot) \iff \phi$, where $\text{pred} \in \text{Preds}$ is a predicate symbol, $\cdot$ are the parameters of $\text{pred}$, and $\phi \in \text{SH}^3$ is an existentially-quantified symbolic heap as defined in Fig. 1 [16, 1]. We assume that all rules of the same predicate $\text{pred}$ have the same parameters. We collect these free variables of $\text{pred}$ in the set $\text{fvars}(\text{pred})$. The size of an SID $\Phi$, $|\Phi|$, is the sum of the sizes of the formulas in its rules.

**Example 2.1.** The predicates \text{odd} and \text{even} in Section 1 define all lists of odd and even length, respectively. The predicate \text{tll} in Fig. 2a defines binary trees whose leaves are connected in a singly-linked list (TLL). The parameters correspond to the root, the left-most leaf ($x_2$) and the successor of the right-most leaf ($x_3$). Every node of a TLL contains three pointer fields: The left successor, the right successor (non-null at inner nodes, null at leaves), and the next leaf (null at inner nodes, non-null at the leaves). We show a graphical representation of a TLL in Fig. 2b.

**Semantics.** We use the standard stack-heap semantics of separation logic [30]. Let $\text{Loc}$ be an infinite set of locations. A stack is a finite partial function $s: \text{Var} \rightharpoonup \text{Loc}$. A heap is a finite partial function $h: \text{Loc} \rightharpoonup \text{Loc}^+$. A model is a stack–heap pair $(s, h)$ with $s(\text{nil}) \notin \text{dom}(h)$.

We define the set of allocated variables of a model, $\text{alloc}(s, h) := \{ x \mid s(x) \in \text{dom}(h) \}$. We denote by $h_1 + h_2$ the disjoint union of the heaps $h_1, h_2$. If $\text{dom}(h_1) \cap \text{dom}(h_2) \neq \emptyset$, then $h_1 + h_2$ is undefined. We let $\text{locs}(h) := \text{dom}(h) \cup \bigcup \text{img}(h)$ and $\text{dangling}(h) := \{ l \in \bigcup \text{img}(h) \mid l \notin \text{dom}(h) \}$.

Figure 3 defines the semantics of separation logic formulas $\phi$ w.r.t. a fixed SID $\Phi$. In the semantics of equalities and disequalities, we follow [27, 22] and require that the heap is empty. This ensures that $t$ is not definable in guarded formulas (e.g., as $x \approx x$). Observe that we use a precise [12] semantics of the points-to assertion: $(s, h) \models \phi. \ x \mapsto y$ holds only in single-pointer heaps. A heap is the model of a predicate call $\text{pred}(y)$ iff it is the model of a rule of the predicate once the free variables of the rule, $\cdot$, have been replaced by the actual arguments, $y$. Our semantics of predicates is equivalent to the least fixed-point semantics as used e.g. in [9].

As usual, $(s, h) \models \phi_1 \ast \phi_2$ iff $h$ can be split into disjoint heaps that are models of $\phi_1$ and $\phi_2$; $(s, h) \models \phi_1 \Rightarrow \phi_2$ iff extending $h$ with a model of $\phi_1$ always yields a model of $\phi_2$, provided the
In guarded models, we call all rules $(\text{pred}(x) \iff \text{pred}(y))$.

Definition 2.2.

Let $(s, h)$ be models. $(s, h)$ and $(s', h')$ are isomorphic, $(s, h) \cong (s', h')$, if there exists a bijection $\sigma : \text{locs}(h) \cup \text{img}(s) \rightarrow \text{locs}(h') \cup \text{img}(s')$ such that (1) for all $x$, $s'(x) = \sigma(s(x))$ and (2) $h' = \{ (\sigma(l) \Rightarrow \sigma(h(l)) \mid l \in \text{dom}(h)) \}$.

Lemma 2.3. Let $\phi \in \text{SL}^\text{bfw}_\text{sf}$. Let $(s, h)$ and $(s', h')$ be models with $(s, h) \cong (s', h')$. Then $(s, h) \models \phi$ iff $(s', h') \models \phi$.

The logic $\text{SLID}^\text{bfw}$. In this article, we assume that all SIDs satisfy progress, connectivity, and establishment and denote the resulting restricted formalism by $\text{SLID}^\text{bfw}$.

A predicate $\text{pred}$ satisfies progress iff every rule of $\text{pred}$ contains exactly one points-to assertion and there exists a variable $x \in \text{fvars}(\text{pred})$ such that for all rules $\langle \text{pred} \Leftarrow \phi \rangle \in \Phi$, $x$ is the variable on the left-hand side of this points-to assertion. In this case, we call $x$ the root of the predicate. Moreover, if the $i$-th parameter of $\text{pred}$ is the root of $\text{pred}$, then $\text{predroot}(\text{pred}(z_1, \ldots, z_k)) := z_i$.

A predicate $\text{pred}$ satisfies connectivity iff for all rules $\langle \text{pred}(x) \Leftarrow (x \Rightarrow y) \ast \psi \rangle \in \Phi$, the root parameters of the recursive calls in $\psi$ occur in $y$. A predicate $\text{pred}$ is established iff all existentially quantified variables across all rules of $\text{pred}$ are eventually allocated. Formally, for all rules $\langle \text{pred} \Leftarrow \exists y. \phi \rangle \in \Phi$ and for all models $(s, h)$, if $(s, h) \models \phi$ then $s(y) \subseteq \text{dom}(h)$.

Example 2.4 (SID Assumptions). The SIDs in Ex. 2.1 are all in $\text{SLID}^\text{bfw}$.

Throughout this article, we often restrict our attention to models of guarded formulas, which we call guarded models and collect in $\text{M}^\text{sf}_\Phi := \{ (s, h) \mid \text{ex}. \phi \in \text{SL}^\text{bfw}_\text{sf}, (s, h) \models \phi \}$.

In guarded models, only locations in $\text{img}(s)$ can be dangling.

Lemma 2.5. Let $\phi \in \text{SL}^\text{bfw}_\text{sf}$ and $(s, h) \models \phi$. Then dangling $(h) \subseteq \text{fvars}(\phi)$.
that encodes the aforementioned derivation tree. Every nonterminal is translated to a node in a finite set of production rules. Let Theorem 3.3.

Example 3.4. We illustrate the encoding in Fig. 5. Figure 5a shows the rules R of a simple CFG G = ⟨{S, A, B, C}, {a₁, a₂}, R, S⟩. Figure 5b, in Fig. 5c, we show the stack–heap model that encodes the aforementioned derivation tree. Every nonterminal is translated to a node in a
induced letters \((a)\) Production rules of binary tree (blue). The leaves of the tree are linked. They each have a successor that encodes a terminal symbol of the derivation (orange): The node contains \(a\) in the figure) to represent terminal \(a\). The list of linked leaves and orange nodes together form the induced word of the model as defined later in Definition 3.7.

To show the correctness of the encoding, we need the induced words of the models of \(\Phi\).

**Definition 3.5.** Let \(G = (N, T, R, S)\) and let \(\Phi\) be the corresponding SID encoding. Let \((s, h) \models_\Phi \text{word}(x, y)\) and let \(j_1, \ldots, j_m \in \{1, \ldots, n\}\) be such that
\[
(s, h) \models_\Phi \exists n_1, \ldots, n_{m-1}, b_1, \ldots, b_m. ((x \mapsto \langle n_1, b_1 \rangle) \ast \text{letter}_{j_1}(b_1))
\ast ((n_1 \mapsto \langle n_2, b_2 \rangle) \ast \text{letter}_{j_2}(b_2)) \ast \cdots \ast ((n_{m-1} \mapsto \langle y, b_m \rangle) \ast \text{letter}_{j_m}(b_m)).
\]
We define the induced letters of \((s, h)\) and \(x, y\) as \(\text{letters}(s, h, x, y) := a_{j_1}a_{j_2}\cdots a_{j_m}\).

Every model that satisfies \(N(x_1, x_2, x_3)\) contains a sub-heap that satisfies the word predicate.

**Lemma 3.6.** Let \(G_1 = (N_1, T, R_1, S_1)\), \(G_2 = (N_2, T, R_2, S_2)\) and let \(\Phi\) be the corresponding SID encoding. Let \(x_1, x_2, x_3 \in \text{Var}\), \(N \in N_1 \cup N_2\) and let \((s, h) \models_\Phi N(x_1, x_2, x_3) \ast t\). Then there exists a unique heap \(h_w \subseteq h\) with \((s, h_w) \models_\Phi \text{word}(x_2, x_3)\).

Lemma 3.6 ensures that the following is well defined.

**Definition 3.7.** Let \(G = (N, T, R, S)\) and let \(\Phi\) be the corresponding SID encoding. Let \(x_1, x_2, x_3 \in \text{Var}\), \(N \in N\) and let \((s, h) \models_\Phi N(x_1, x_2, x_3)\). Let \(h_w \subseteq h\) be the unique heap with \((s, h_w) \models_\Phi \text{word}(x_2, x_3)\). We define the induced word of \((s, h)\) and \(N\) as \(\text{wordof}_N(s, h, x_2, x_3) := \text{letters}(s, h_w, x_2, x_3)\).

**Lemma 3.8.** Let \(G = (N, T, R, S)\) and let \(\Phi\) be the corresponding SID encoding. Let \(1 \leq i \leq 2\), \(x_1, x_2, x_3 \in \text{Var}\), and let \(w \in \mathcal{L}(G)\). Then there exists a model \((s, h)\) with \((s, h) \models_\Phi S(x_1, x_2, x_3)\) and \(\text{wordof}_S(s, h, x_2, x_3) = w\).

**Lemma 3.9.** Let \(G = (N, T, R, S)\) and let \(\Phi\) be the corresponding SID encoding. Let \(x_1, x_2, x_3 \in \text{Var}\) and let \((s, h)\) be a model with \((s, h) \models_\Phi S(x_1, x_2, x_3)\). Then \(\text{wordof}_S(s, h, x_2, x_3) \in \mathcal{L}(G)\).

We need an auxiliary result involving the formula
\[
\text{word}_2(x, y) := (\text{word}(x, y) \oplus S_2(a, x, y)) \oplus S_2(a, x, y)
\]
before we can prove the undecidability results.
Lemma 3.10. Let $G_2 = \langle N_2, T, R_2, S_2 \rangle$, let $\Phi$ be the corresponding SID encoding, and let $(s, h)$ be a model. Then $(s, h) \models \Phi \text{ word}_2(x, y)$ iff $(s, h) \models \Phi \text{ word}(x, y)$ and letters$(s, h, x, y) \in \mathcal{L}(G_2)$.

Proof. Let $(s, h) \models \Phi \text{ word}_2(x, y)$. By the semantics of $\Phi$, there exists a heap $h_1$ with $(s, h_1) \models \Phi \text{ word}(x, y) \oplus S_2(a, x, y)$ such that $(s, h + h_1) \models \Phi S_2(a, x, y)$. Observe that $h_1$ contains precisely the inner nodes of the model $(s, h + h_1)$, i.e., everything except the part of the model that induces the word. Consequently, $h$ is the part of the model that induces the word, i.e., $\text{word}_2(s, h + h_1, x, y) = \text{letters}(s, h, x, y)$ and $(s, h) \models \Phi \text{ word}(x, y)$. Lemma 3.9 then yields letters$(s, h, x, y) \in \mathcal{L}(G_2)$.

Conversely, let $(s, h)$ be such that $w := \text{letters}(s, h, x, y) \in \mathcal{L}(G_2)$. As a consequence of Lemma 3.8, there exists a heap $h_1$ with $(s, h + h_1) \models \Phi S_2(a, x, y)$. By the semantics of $\Phi$ and because $(s, h) \models \Phi \text{ word}(x, y)$ by assumption, we then have $(s, h_1) \models \Phi \text{ word}(x, y) \oplus S_2(a, x, y)$. Because $(s, h + h_1) \models \Phi S_2(a, x, y)$, we obtain by the semantics of $\Phi$ that $(s, h) \models \Phi \text{ word}(x, y) \oplus S_2(a, x, y)$.

We are ready to prove the undecidability results.

Theorem 3.11. The satisfiability problems of the separation logics (1) $\text{SL}(\land, \ast, \lor, =)$, (2) $\text{SL}(\land, \ast, \neg)$, (3) $\text{SL}(\land, \ast, \rightarrow)$, and (4) $\text{SL}(\neg, \lor, \ast)$ are undecidable.

Proof. Throughout this proof, let $G_1 = \langle N_1, T, R_1, S_1 \rangle$, $G_2 = \langle N_2, T, R_2, S_2 \rangle \in \mathcal{CFG}$ and let $\Phi$ be the corresponding SID encoding.

Undecidability of $\text{SL}(\land, \ast, \lor, =)$: We claim that $\phi := (S_1(a, x, y) \ast t) \land (S_2(b, x, y) \ast t)$ is satisfiable iff $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) \neq \emptyset$. We prove the implications separately.

Assume $\phi$ is satisfiable. Then there exists a model $(s, h)$ with $(s, h) \models \phi$. Let $h_{w_1}, h_{w_2} \subseteq h$ be such that $\text{word}_2(s, h, x, y) = \text{letters}(s, h_{w_1}, x, y)$; such heaps exist by Lemma 3.6.

Observe that both $(s, h_{w_1}) \models \phi \text{ word}(x, y)$ and $(s, h_{w_2}) \models \phi \text{ word}(x, y)$. Consequently, $h_{w_1} = h_{w_2}$, which implies $\text{word}_2(s, h, x, y) = \text{word}_2(s, h, x, y) = w$. By Lemma 3.9, it follows that $w \in \mathcal{L}(G_1)$ and $w \in \mathcal{L}(G_2)$, i.e., $w \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$.

Conversely, assume $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) \neq \emptyset$ and let $w \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$. By Lemma 3.8, there exist models $(s, h_1), (s, h_2)$ with $(s, h_1) \models S_1(a, x, y)$ and $(s, h_2) \models S_2(b, x, y)$. Let $h_{w_1} \subseteq h_1, h_{w_2} \subseteq h_2$ be the unique heaps with $\text{word}_2(s, h_1, x, y) = \text{letters}(s, h_{w_1}, x, y) = w = \text{letters}(s, h_{w_2}, x, y) = \text{word}_2(s, h_2, x, y)$.

Observe that $(s, h_{w_1}) \simeq (s, h_{w_2})$. Consequently, we can replace $h_2$ with an isomorphic heap that contains $h_{w_1}$ (as opposed to $h_{w_2}$) as sub-heap and is otherwise disjoint from $h_1$, i.e., there exists a heap $h'_2$ such that $h_2 \cong h'_2$, $\text{locs}(h'_2) \cap \text{locs}(h_1) = \text{locs}(h_{w_1})$, and $\text{word}_2(s, h_2', x, y) = \text{letters}(s, h_{w_1}, x, y) = w$. Note in particular that $(s, h'_2) \models \phi S_2(b, x, y)$, because isomorphic models satisfy the same formulas. Now let $h := h_1 \cup h'_2$ be the (non-disjoint) union of $h_1$ and $h'_2$. Since $h_1 \subseteq h$ and $(s, h_1) \models S_1(a, x, y)$, we have $(s, h) \models S_1(a, x, y) \ast t$ and similarly, since $h'_2 \subseteq h$ and $(s, h'_2) \models \phi S_2(b, x, y)$, we have that $(s, h) \models \phi S_2(b, x, y)$. Consequently, $(s, h) \models \phi$.

Undecidability of $\text{SL}(\land, \ast, \lor, =)$ follows directly from the undecidability of $\text{SL}(\land, \ast, \lor)$, because $t$ is definable in $\text{SL}(\land, \ast, \lor, =)$; for example $t := \neg(\mathsf{emp} \land \neg \mathsf{emp})$.

Undecidability of $\text{SL}(\land, \ast, \rightarrow)$: Follows directly from the undecidability of $\text{SL}(\land, \ast, \lor, =)$, because $t$ is definable in $\text{SL}(\land, \ast, \rightarrow)$; for example $t := (x \neq x) \rightarrow \mathsf{emp}$.

Undecidability of $\text{SL}(\neg, \lor, \ast)$: We claim that the formula $\psi := \text{word}_2(x, y) \oplus S_1(a, x, y)$, for word$_2$ as defined earlier in this section, is satisfiable iff $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) \neq \emptyset$. Intuitively, this holds because $\psi$ is satisfiable iff it is possible to replace the “word part” of a model of $S_1(a, x, y)$ with the “word part” of a model of $S_2(b, x, y)$.

We now formalize this intuition. Assume $\psi$ is satisfiable and let $(s, h)$ be such that $(s, h) \models \phi \psi$. By the semantics of $\oplus$, we have that there exists a heap $h_0 \subseteq h$ with $h_0 \models \phi \text{ word}_2(x, y)$.
and \( (s, h + h_0) \models S_1(a, x, y) \). As \( \text{letters}(s, h_0, x, y) \in \mathcal{L}(G_2) \) by Lemma 3.10, we have in particular that \( (s, h_0) \models \text{word}(x, y) \). It follows that \( h_0 \) is the unique sub-heap of \( h + h_0 \) with \( \text{wordof}_s(s, h + h_0) = \text{letters}(s, h_0, x, y) \). By Lemma 3.9, \( \text{letters}(s, h_0, x, y) \in \mathcal{L}(G_1) \). Together with Lemma 3.10, we thus have that \( \text{letters}(s, h_0, x, y) \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2) \).

Conversely, assume there exists a word \( w \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2) \). By Lemma 3.8, there exist heaps \( h, h_0, h', h_0' \) with \( (s, h) \models S_1(a, x, y) \), \( \text{wordof}_{S_1}(s, h, x, y) = \text{letters}(s, h_0, x, y) \), \( (s, h') \models S \), \( \text{wordof}_{S_2}(s, h', x, y) = \text{letters}(s, h_0', x, y) \).

Because \( \text{letters}(s, h_0, x, y) = \text{letters}(s, h_0', x, y) \), it holds that \( h_0 \equiv h_0' \)—if the models are not isomorphic, simply replace \( h' \) with an appropriate isomorphic heap to establish this property. Let \( h_2 \subseteq h' \) be the sub-heap of \( h' \) with \( h_2 + h_0 = h' \).

By Lemma 3.10, \( (s, h_0) \models \text{word}(x, y) \). Consequently, \( (s, h_2) \models \psi \), i.e., \( \psi \) is satisfiable.

The fundamental difference between the logics in Theorem 3.11 and \( \text{SL}_{\text{bw}}^\Phi \) lies in the possibility to decompose the heap into parts with unboundedly many dangling pointers: The number of dangling pointers in the models of \( \text{SL}_{\text{bw}}^\Phi \) formulas is always bounded by the number of free variables of the formula (cf. Lemma 2.5).

## 4 The Types Abstraction

Having established that all natural extensions of \( \text{SL}_{\text{bw}}^\Phi \) are undecidable, we now turn to the decidability of \( \text{SL}_{\text{bw}}^\Phi \). In this section, we present the \( \Phi \)-type abstraction that is at the heart of our decision procedure. The \( \Phi \)-types abstraction and the associated results essentially follow our paper [23]; we only summarize here what is needed for proving the decidability results in the next section. We note, however, that since the publication of [23], we have tweaked the definition of the abstraction to fix an incompleteness issue and improve the presentation; we refer the reader to our technical report [26] for a full self-contained exposition.

The idea behind the \( \Phi \)-type abstraction is as follows. Given a SID \( \Phi \) and a model \( (s, h) \), we compute a set of formulas that encodes all the ways that one or more predicates of \( \Phi \) can be partially unfolded such that \( (s, h) \) is a model of the partially-unfolded predicates.

**Example 4.1.** Consider the following SID \( \Phi \).

\[
\begin{align*}
\text{tree}(r) & \equiv \exists x. (r \mapsto x) \ast \text{treerp}(x, r) \\
\text{treerp}(x, r) & \equiv (x \mapsto (\text{nil}, \text{nil}, r)) \\
\text{treerp}(x, r) & \equiv \exists c_1, c_2. (x \mapsto (c_1, c_2, r)) \ast \text{treerp}(c_1, r) \ast \text{treerp}(c_2, r)
\end{align*}
\]

Let \( s = \{ r \mapsto 1, x \mapsto 2, y \mapsto 4 \} \), \( h = \{ 1 \mapsto 2, 2 \mapsto (3, 4, 1), 3 \mapsto (0, 0, 1), 4 \mapsto (5, 6, 1), 5 \mapsto (0, 0, 1) \} \). We display \( (s, h) \) in Figure 6a. Each node is labeled with a location and the stack variable interpreted by the location (if any). The first two outgoing pointers of each node are displayed by solid edges, the third pointer by a dashed edge. Fig. 6b shows a \( \Phi \)-forest \( f = \{ t_1, t_2, t_3 \} \) that encodes one way to derive the model \( (s, h) \) by unfolding predicates of the SID. Both \( t_1 \) and \( t_2 \) only partially unfold the predicates at their roots. The stack-forest projection of \( s \) and \( f \) is the formula \( (\text{treerp}(x, r) \rightarrow \text{tree}(r)) \ast (\text{treerp}(y, r) \rightarrow \text{treerp}(x, r)) \ast (\text{emp} \rightarrow \text{treerp}(y, r)) \).

In this section, we formalize the notion of \( \Phi \)-forests; and we show how to systematically obtain the projections of all relevant forests of a model.

### 4.1 \( \Phi \)-Forests

We begin by formalizing partial unfoldings of \( \Phi \)-predicates. To this end, we introduce \( \Phi \)-forests (Definition 4.3) made up of \( \Phi \)-trees (Definition 4.2). Intuitively, a \( \Phi \)-tree encodes one fixed way
Finally, we define the projection of $t$ as the nodes of the tree; and every node is mapped to its successors in the (directed) tree of the Φ-tree. Figure 6: A model of $\text{tree}(r)$ and one of the Φ-forests corresponding to this model.

to unfold a predicate call by means of the rules of the SID $\Phi$. Our notion of Φ-trees is related to the unfolding trees used, e.g., in [16, 17, 20], but there are two key differences. First, we instantiate variables with locations; second, we explicitly allow the unfolding process to stop at any point, i.e., we allow that one or more of the predicate calls introduced in the unfolding process remain folded. We call such folded predicate calls the holes of the Φ-tree.

The nodes of a Φ-tree are labeled with rule instances, which are obtained from the rules of the SID by instantiating both the formal arguments of the predicates and the existentially quantified variables of the rule with locations:

$$\text{RuleInst}(\Phi) := \{(\text{pred}(l) \Leftarrow \phi[x \cdot y/1 \cdot m]) | (\text{pred}(x) \Leftarrow \exists y. \phi) \in \Phi,\ l \in \text{Loc}^{\text{ser}(\text{pred})}, m \in \text{Loc}^{\text{y}}\ and\ all\ (\text{dis})\equal{}\text{equalities in } \phi[x \cdot y/1 \cdot m] \text{ are valid}\}$$

Rule instances are not SL formulas themselves, as the terms in rule instances are locations rather than variables. We will come back to this point later.

We model Φ-trees as functions $\text{t} : \text{Loc} \rightarrow (\text{Loc}^* \times \text{RuleInst}(\Phi))$. The set of locations $\text{Loc}$ serves as the nodes of the tree; and every node is mapped to its successors in the (directed) tree as well as to a rule instance that serves as a node label. For $t$ to be a Φ-tree, it must satisfy a certain set of consistency criteria. To make it easier to work with Φ-trees and formalize the consistency criteria, we first introduce some additional notation.

Let $\Phi$ be an SID and let $t(l) = (m, (\text{pred}(z) \Leftarrow (a \mapsto b) \cdot \phi))$. Note that, by progress of the SID $\Phi$, $\phi$ does not contain points-to assertions. We define:

$$\text{succ}_l(l) := m \quad \text{ptr}_l(l) := a \mapsto b \quad \text{head}_l(l) := \text{pred}(z) \quad \text{calls}_l(l) := \phi$$

$$\text{holepreds}_l(l) := \{\text{pred}(z') \in \text{calls}_l(l) | \forall c \in \text{succ}_l(l). \text{head}_l(c) \neq \text{pred}(z')\}$$

Informally, the hole predicates of $l$ are those predicate calls in $\text{calls}_l(l)$ whose root does not occur in $\text{succ}_l(m)$. We collect all hole predicates of $t$ in $\text{allholepreds}(t) := \bigcup_{c \in \text{dom}(t)} \text{holepreds}_c(c)$.

Finally, we define the projection of $t$ onto the directed graph, $\text{graph}(t) \subseteq \text{Loc} \times \text{Loc}$, induced by its first component, $\text{graph}(t) := \{(x, y) | x \in \text{dom}(t), y \in \text{succ}_l(x)\}$.

**Definition 4.2 (Φ-Tree).** Let $\Phi$ be an SID. A partial function $t : \text{Loc} \rightarrow (\text{Loc}^* \times \text{RuleInst}(\Phi))$ is a Φ-tree iff

$$t_1 := (1 \mapsto 2) \cdot \text{treerp}(2, 1)$$

$$t_2 := (2 \mapsto 3, 4, 1) \cdot \text{treerp}(3, 1) \cdot \text{treerp}(4, 1)$$

$$t_3 := (3 \mapsto 0, 0, 1)$$

$$t_1 := (3 \mapsto 0, 0, 1)$$

$$t_2 := (5 \mapsto 0, 0, 1)$$

$$t_3 := (6 \mapsto 0, 0, 1)$$

(a) A model of $\text{tree}(r)$. (b) A corresponding Φ-forest $f$ consisting of trees $t_1, t_2, t_3$. 
1. graph(t) is a (directed) tree and

2. t is \( \Phi \)-consistent, i.e., for all \( l \in \text{dom}(t) \), if \( \text{succ}_1(l) = \langle y_1, \ldots, y_k \rangle \), \( \text{head}_1(l) = \text{pred}(z) \), \( \text{ptr}_1(l) = a \mapsto b \), and \( \text{calls}_1(l) = \text{pred}_1(z_1) \cdots \text{pred}_m(z_m) \ast \Pi \), \( \Pi \) pure, then (1) \( l = a \), (2) \( \text{succ}_1(l) \subseteq b \), and (3) \( \{ \text{head}_1(y_1), \ldots, \text{head}_1(y_k) \} \subseteq \{ \text{pred}_1(z_1), \ldots, \text{pred}_m(z_m) \} \).

Let \( t \) be a \( \Phi \)-tree. As \( t \) is a directed tree, it has a root, which we denote by \( \text{root}(t) \). We set \( \text{rootpred}(t) := \text{head}_1(\text{root}(t)) \).

We combine zero or more \( \Phi \)-trees into \( \Phi \)-forests.

**Definition 4.3 (\( \Phi \)-Forest).** Let \( \Phi \) be an SID. Let \( t_1, \ldots, t_k \) be \( \Phi \)-trees. The set \( \{ t_1, \ldots, t_k \} \) is a \( \Phi \)-forest iff \( \text{dom}(t_i) \cap \text{dom}(t_j) = \emptyset \) for \( i \neq j \).

**Example 4.4 (\( \Phi \)-forest).** Figure 6b shows the \( \Phi \)-forest \( \{ t_1, t_2, t_3 \} \) with \( t_1 = \{ \underbrace{1 \mapsto \langle \varepsilon, \text{treer}_p(1) \rangle} \mapsto (1 \mapsto 2) \ast \text{treer}_p(2, 1) \}, t_2 = \{ 2 \mapsto (3, \text{treer}_p(2, 1) \mapsto (2 \mapsto 3, 4, 1) \ast \text{treer}_p(3, 1) \ast \text{treer}_p(4, 1)}, 3 \mapsto \langle \varepsilon, \text{treer}_p(3, 1) \mapsto (3 \mapsto 0, 0, 1) \}, \text{and } t_3 = \{ 4 \mapsto (5, 6), \text{treer}_p(4, 1) \mapsto (4 \mapsto 5, 6, 1) \ast \text{treer}_p(5, 1) \ast \text{treer}_p(6, 1)), 5 \mapsto \langle \varepsilon, \text{treer}_p(5, 1) \mapsto (5 \mapsto 0, 0, 1), 6 \mapsto \langle \varepsilon, \text{treer}_p(6, 1) \mapsto (6 \mapsto 0, 0, 1) \} \} \).

**Definition 4.5 (Induced Heap).** Let \( \{ t \} \) be a \( \Phi \)-forest. The induced heap of \( \{ t \} \) is given by \( \text{heap}(\{ t \}) := \bigcup_{i \in \{ t \}} \bigcup_{e \in \text{dom}(t_i)} \text{ptr}_i(e) \).

**Example 4.6.** For \( h \) and \( f \) as in Example 4.1, \( h = \text{heap}(f) \).

Our notion of \( \Phi \)-trees generalizes “unfolding trees” [16] in the sense that every model of a predicate call corresponds to (at least one) \( \Phi \)-tree without holes.

**Lemma 4.7.** Let \( \Phi \) be an SID, \( \text{pred}(z) \) a predicate call, and \( \langle s, h \rangle \) a model. If \( \langle s, h \rangle \models \Phi \text{pred}(z) \) there exists a \( \Phi \)-tree \( t \) with \( \text{rootpred}(t) = \text{pred}(s(z)) \), \( \text{allholepreds}(t) = \emptyset \), and \( \text{heap}(\{ t \}) = h \).

### 4.2 Projecting \( \Phi \)-Forests onto Formulas

The main insight behind the projection of \( \Phi \)-forests onto SL formulas is that every \( \Phi \)-tree \( t \) can be viewed as encoding a model of \( \text{rootpred}(t) \) from which models of \( \text{allholepreds}(t) \) have been subtracted. This can be naturally encoded by a magic wand, \( (\star \text{allholepreds}(t)) \rightarrow \text{rootpred}(t) \).

Recall, however, that the above is not a formula, because it contains locations rather than variables. In our projection function, we thus need to replace the locations with variables. To this end, we need guarded versions of both existential and universal quantifiers, which we denote \( \exists \) and \( \forall \). They have the following semantics.

- \( \langle s, h \rangle \models \exists e_1, \ldots, e_k : \phi \) iff there exist pairwise different \( v_1, \ldots, v_k \in \text{dom}(h) \setminus \text{img}(s) \) such that \( \langle s \cup \{ e_1 \mapsto v_1, \ldots, e_k \mapsto v_k \}, h \rangle \models \phi \).
- \( \langle s, h \rangle \models \forall a_1, \ldots, a_k : \phi \) iff for all pairwise different \( v_1, \ldots, v_k \in \text{Loc} \setminus (\text{locs}(h) \cup \text{img}(s)) \), it holds that \( \langle s \cup \{ a_1 \mapsto v_1, \ldots, a_k \mapsto v_k \}, h \rangle \models \phi \).

Note that these quantifiers are not quite dual, i.e., \( \exists x : \phi \) is not equivalent to \( \forall x : \neg \phi \). We take the following approach to translate a forest \( f \) to a formula with guarded quantifiers:

1. Every location \( v \in \text{img}(s) \) is replaced by an arbitrary variable in \( s^{-1}(v) \).
2. Every location \( v \in \text{locs}(\text{heap}(f)) \) with \( v \notin \text{img}(s) \) is replaced by a guarded existential. Note that by Lemma 2.5, \( v \in \text{dom}(\text{heap}(f)) \), as required by the above semantics of \( \exists \).
3. All other locations are replaced by a guarded universal.

Let us formalize this construction. In the following, we assume for all stacks that \( \text{dom}(s) \cap \{a_1, a_2, \ldots \} \cup \{e_1, e_2, \ldots \} = \emptyset \).

**Definition 4.8** (Tree projection). Let \( \mathbf{t} \) be a \( \Phi \)-tree and let \( v \subseteq \text{Loc} \). Let \( \text{locs}(\phi) \) be the set of all locations that occur in \( \phi \). The tree projection of \( t \) w.r.t. \( v \), \( \text{project}^{\text{Loc}}(v, t) \), is then given by

\[
\text{project}^{\text{Loc}}(v, t) := \forall a. \psi[w/a] \quad \text{where} \quad \psi := (\star \text{allholepreds}(t)) \rightarrow \text{rootpred}(t)
\]

\[
w := \text{locs}(\psi) \setminus (v \cup \text{locs}(\text{heap}(t)))
\]

\[
a := \langle a_1, \ldots, a_{|w|} \rangle.
\]

**Definition 4.9.** Let \( f = \{t_1, \ldots, t_k\} \) be a \( \Phi \)-forest with \( \text{heap}(f) \in \mathcal{M}^S_\Phi \). Let \( v := \text{img}(s) \) and let \( w := \text{locs}(\text{heap}(f)) \setminus \text{img}(s) \) be the (arbitrarily ordered) sequence of locations that occur in a pointer in \( \text{heap}(f) \) but are not the value of any stack variable. Let \( x \subseteq \text{dom}(s) \) be such that \( s(x) = v \) and let \( e := \langle e_1, e_2, \ldots, e_{|w|} \rangle \). Finally, let \( \psi := \star_{1 \leq i \leq k} \text{project}^{\text{Loc}}(v \cup w, t_i) \). The stack–forest projection of \( s \) and \( f \), \( \text{project}(s, f) \), is given by \( \exists e. \psi[v \cdot w / x \cdot e] \).

**Example 4.10.** Let \( \Phi, s, t_1, t_2, t_3 \) and \( f \) be as in Example 4.1. (1) The formula \( \text{project}(s, f) \) is given in Example 4.1. (2) Let \( s' = \{x \rightarrow 2, y \rightarrow 4\} \). Then \( \text{project}(s', \{t_2, t_3\}) = \exists e_1. (\text{treerp}(y, e_1) \rightarrow \text{treerp}(x, e_1)) \star (\text{emp} \rightarrow \text{treerp}(y, e_1)) \). (3) For \( s'' = \{x \rightarrow 1, y \rightarrow 2\} \) and \( t'' = \{1 \rightarrow \langle e, \text{even}(1, 3) \rangle = (1 \rightarrow 2) \star \text{odd}(2, 3)\} \), \( \text{project}(s'', \{t''\}) = \forall a_1. \text{odd}(y, a_1) \rightarrow \text{even}(x, a_1) \).

Stack–forest projection is sound in the following sense.

**Lemma 4.11.** Let \( f \) be a \( \Phi \)-forest with \( \text{heap}(f) \in \mathcal{M}^S_\Phi \) and let \( s \) be a stack. Then \( (s, \text{heap}(f)) \models_{\Phi} \text{project}(s, f) \).

The correctness of Lemma 4.11 hinges on the use of guarded quantifiers. For example, let \( \Phi = \{\text{pred}(x_1, x_2) := (x_1 \rightarrow \text{nil}) \star (x_1 \neq x_2)\} \) and \( t = \{v_1 := \langle e, \text{pred}(v_1, v_2) := (v_1 \rightarrow 0) \star (v_1 \neq v_2)\}\), \( f = \{t\} \), and \( s = \{y \rightarrow v_1\} \). Then \( (s, \text{heap}(f)) \models_{\Phi} \text{project}(s, f) = \forall a_1. \text{emp} \rightarrow \text{pred}(y, a_1) \), but \( (s, \text{heap}(f)) \not\models_{\Phi} \text{emp} \rightarrow \text{pred}(y, y) \), so \( (s, \text{heap}(f)) \not\models_{\Phi} \forall a_1. \text{emp} \rightarrow \text{pred}(y, a_1) \).

All stack–forest projections can be obtained by “partially unfolding” symbolic heaps (and adding appropriate quantifiers), so we call them unfolded symbolic heaps (USHs) w.r.t. \( \Phi \).

**Delimited USHs.** Clearly, there are infinitely many USHs w.r.t. any (nonempty) SID \( \Phi \). This makes USHs unsuitable for defining a finite abstraction. To obtain a finite abstraction, we only consider USHs where (1) all root parameters of predicate calls are free variables and (2) every variable occurs at most once as a root parameter on the left-hand side of a magic wand.

**Definition 4.12.** An unfolded symbolic heap \( \phi \) is delimited iff

1. for all \( \text{pred}(z) \in \phi \), \( \text{predroot}(\text{pred}(z)) \in \text{fvars}(\phi) \), and

2. for all \( z \) there exists at most one predicate call \( \text{pred}(z) \in \phi \) such that \( z = \text{predroot}(\text{pred}(z)) \) and \( \text{pred}(z) \) occurs on the left-hand side of a magic wand.

**Example 4.13.** All projections in Example 4.10 are DUSHs. Let \( t' \) be as in Example 4.10. Then \( \text{project}((\{x \rightarrow 1\}, \{t''\}) = \exists e_1. \forall a_1. \text{odd}(e_1, a_1) \rightarrow \text{even}(x, a_1) \) is not a DUSH, because \( e_1 = \text{predroot}(\text{odd}(e_1, a_1)), \) but \( e_1 \notin \text{dom}(s) \).

We let \( \text{DUSH}_{\Phi} := \{\phi \mid \phi \text{ is USH w.r.t. } \Phi \text{ and delimited} \} \) be the set of delimited USHs (DUSHs) over SID \( \Phi \) and \( \text{DUSH}^S_{\Phi} \) the restriction of \( \text{DUSH}_{\Phi} \) to formulas \( \phi \) with \( \text{fvars}(\phi) \subseteq x \).
Lemma 4.14. Let $\Phi$ be an SID and let $x \in 2^{\text{Var}}$ be a finite set of variables. Let $n := |\Phi| + |x|$. Then $|\text{DUSH}_\Phi| = 2^{|\phi|\log(n)}$.

We will abstract a model by the set of all DUSHs that hold in the model. This makes sense because we can effectively compose such sets, as we will see next.

4.3 The $\Phi$-Type Abstraction

We abstract a model $(s, h)$ by its $\Phi$-type, which consists of the set of DUSHs that hold in $(s, h)$.

Definition 4.15 ($\Phi$-Type). Let $(s, h) \in M^\Phi_\phi$ be a guarded model and $\Phi$ an SID. The $\Phi$-type of $(s, h)$ is given by $\text{type}_\phi(s, h) := \{\text{project}(s, f) \mid f$ is a forest with heap($f) = h\} \cap \text{DUSH}_\Phi$.

In our decision procedure, we need to compose types, i.e., we need an operation $\bullet$ such that

$$\text{type}_\phi(s, h_1 + h_2) = \text{type}_\phi(s, h_1) \bullet \text{type}_\phi(s, h_2).$$

(†)

We now develop such an operation. We begin by defining an operation $\bullet_p$ that satisfies the identity (†), provided that we know that $h_1 + h_2 \neq \perp$. To define $\bullet_p$, we need two ingredients: (1) a variant of $\ast$ that captures all sound ways to move the existential quantifiers to the front of the formula $\text{project}(s, f_1) \ast \text{project}(s, f_2)$ and (2) a derivation operation on projections, $\triangleright$, that rewrites formulas based on the fact that

$$\left(\text{pred}_2(x_2) \ast \psi \Rightarrow \text{pred}_1(x_1)\right) \ast \left(\psi \Rightarrow \text{pred}_2(x_2)\right) \Rightarrow \left(\psi \ast \psi' \Rightarrow \text{pred}_1(x_1)\right).$$

(††)

We need the following auxiliary notation. We write $\forall a. \phi \xrightarrow{3e/\text{mu}} \forall b. \phi'$ if there exist disjoint subsets $a_1, a_2 \subseteq a$ and subsets $u \subseteq u, e \subseteq e$ such that $\phi' = \phi[u_1 \cdot a_2 / u \cdot e]$ and $b = a \cup (a_1 \cup a_2) \cup u$.

Example 4.16. Let $\phi = \forall(a_1, a_2). (\text{odd}(y, a_2) \Rightarrow \text{odd}(e_1, a_2)) \ast (\text{even}(e_1, a_1) \Rightarrow \text{odd}(x, a_1))$. Then

1. $\phi \xrightarrow{3e/\text{mu}} \forall u_1. (\text{odd}(y, e_2) \Rightarrow \text{odd}(e_1, e_2)) \ast (\text{even}(e_1, u_1) \Rightarrow \text{odd}(x, e_2))$, but also
2. $\phi \xrightarrow{3e/\text{mu}} \forall u_1. (\text{odd}(y, u_1) \Rightarrow \text{odd}(e_1, u_1)) \ast (\text{even}(e_1, e_2) \Rightarrow \text{odd}(x, e_2))$, and
3. $\phi \xrightarrow{3e/\text{mu}} \forall a_1. (\text{odd}(y, e_2) \Rightarrow \text{odd}(e_1, e_2)) \ast (\text{even}(e_1, a_1) \Rightarrow \text{odd}(x, a_1))$, etc.

The notions of re-scoping and derivability provide the two ingredients we need to define $\bullet_p$.

Definition 4.17 (Re-scoping). Let $\phi_1 = \exists e_1. \ast_{1 \leq j \leq n} \psi_{i,j}$ for $1 \leq i \leq 2$ such that $e_1 \cap e_2 = \emptyset$. We say that $\zeta$ is a re-scoping of $\phi_1$ and $\phi_2$ if there exist formulas $\psi_{1,\ast_1}, \ldots, \psi_{n,\ast_n}$ such that

1. $\psi_{i,j} \xrightarrow{3e_{i,j}/\text{mu}} \psi_{i,j}'$ for all $i, j$; and
2. $\zeta = \exists e_1 \cdot e_2. \ast_{1 \leq j \leq n} \psi_{1,j}' \ast \exists_{1 \leq j \leq n} \psi_{2,j}'$.

Definition 4.18 (Derivability). We say that $\psi$ is derivable from $\zeta = \exists e. \ast_{1 \leq i \leq n} \forall a_i. \zeta_i$, written $\zeta \triangleright \psi$, iff there exist indices $m_1$ and $m_2$, variables $u_1, u_2, b_1, b_2$, and formulas $\phi_1, \phi_2, \phi$ such that

1. $\forall a_{m_1}. \forall b_{m_2}. \phi_1 \Rightarrow \forall b_{m_2}. \phi_2$ for $1 \leq i \leq 2$, $2. \phi_1 \ast \phi_2 \Rightarrow \phi$ by (†), and
3. $\psi$ is obtained from $\zeta$ by removing the subformulas $\forall a_{m_1} \cdot \zeta_{m_1}$ and adding the subformula $\forall (b_1 \cup b_2). \phi$.

We are finally ready to define a composition operation $\bullet_p$ that satisfies (†).

Definition 4.19 (Projection composition). Let $\phi_1, \phi_2 \in \text{DUSH}_\phi$. Then

$$\phi_1 \bullet_p \phi_2 := \{\phi \mid \text{there exist a } k \geq 1 \text{ and } \zeta_1, \ldots, \zeta_k \text{ s.t. } \zeta_1 \text{ is a re-scoping of } \phi_1 \text{ and } \phi_2, \zeta_i \triangleright \zeta_{i+1} \text{ for all } 1 \leq i < k, \text{ and } \zeta_k = \phi\}.$$
Example 4.20. • For $\phi_1 = \text{ls}(x_2, x_3) \rightarrow \text{ls}(x_1, x_3)$ and $\phi_2 = \text{emp} \rightarrow \text{ls}(x_2, x_3)$, it holds that $\phi_1 \ast \phi_2 \models \text{emp} \rightarrow \text{ls}(x_1, x_3)$. Hence, $(\text{emp} \rightarrow \text{ls}(x_1, x_3)) \in \phi_1 \bullet_p \phi_2$.

• Let $\phi_1 = \forall v. \text{ls}(x_2, a) \rightarrow \text{ls}(x_1, a)$ and $\phi_2 = \forall b. \text{ls}(x_3, b) \rightarrow \text{ls}(x_2, b)$, $\phi'_1 = \forall v. \text{ls}(x_2, c) \rightarrow \text{ls}(x_1, c)$, and $\phi'_2 = \forall c. \text{ls}(x_3, c) \rightarrow \text{ls}(x_2, c)$. Observe that $\phi_i \xrightarrow{\text{prop}/\text{pred}} \phi'_i$ and that $(\text{ls}(x_2, c) \rightarrow \text{ls}(x_1, c)) \ast (\text{ls}(x_3, c) \rightarrow \text{ls}(x_2, c)) \models \text{ls}(x_3, c) \rightarrow \text{ls}(x_1, c)$. Hence, $(\forall c. \text{ls}(x_3, c) \rightarrow \text{ls}(x_1, c)) \in \phi_1 \bullet_p \phi_2$.

• Let $\phi_1 = \exists e_1. (\text{treerp}(y, e_1) \rightarrow \text{treerp}(x, e_1)) \ast (\text{emp} \rightarrow \text{treerp}(y, e_1))$ and $\phi_2 = \text{emp}$. Then $\psi = \exists e_1. (\text{treerp}(y, e_1) \rightarrow \text{treerp}(x, e_1)) \ast (\text{emp} \rightarrow \text{treerp}(y, e_1))$ is a re-scoping of $\phi_1$ and $\phi_2$ and $(\text{treerp}(y, e_1) \rightarrow \text{treerp}(x, e_1)) \ast (\text{emp} \rightarrow \text{treerp}(y, e_1)) \models \text{emp} \rightarrow \text{treerp}(x, e_1)$, so $(\exists e_1. \text{emp} \rightarrow \text{treerp}(x, e_1)) \in \phi_1 \bullet_p \phi_2$.

Lemma 4.21 (Soundness). Let $s$ be a stack, let $h_1, h_2$ be heaps, and $\phi_1 \in \text{type}_A(s, h_1), \phi_2 \in \text{type}_A(s, h_2)$. If $h_1 + h_2 \neq \bot$ and $(s, h_1), (s, h_2) \in M^s_A$, then $(s, h_1 + h_2) \models \phi$ for all $\psi \in \phi_1 \bullet_p \phi_2$.

Moreover, $\bullet_p$ satisfies $(\dagger)$ when applied to disjoint models. The only missing piece is to ensure that the composition is only defined if the underlying heaps can be composed via $\dagger$. This is easy to ensure, as it is possible to infer from $\text{type}_A(s, h)$ the set of variables $\text{allocated}(s, h)$:

Let $T$ be a $\Phi$-type. We define the set of allocated variables of $T$ as

\[ \text{allocated}(T) := \{ x \mid \text{there ex. } \phi \in T \text{ and } (\psi \rightarrow \text{pred}(z)) \in \phi \text{ s.t. } x = \text{predroot}((\psi \rightarrow \text{pred}(z))) \} \].

Lemma 4.22. For all models $(s, h) \in M^s_A$, it holds that $\text{allocated}(s, h) = \text{allocated}(\text{type}_A(s, h))$.

This allows us to check for double allocation in the definition of composition.

Definition 4.23 (Type composition). Let $T_1, T_2$ be types. The composition of $T_1$ and $T_2$ is

\[ T_1 \bullet T_2 := \begin{cases} \bot, & \text{if } \text{allocated}(T_1) \cap \text{allocated}(T_2) \neq \emptyset \\ \phi_1 \bullet_p \phi_2, & \text{otherwise}. \end{cases} \]

Furthermore, $\text{type}_A(s, \cdot)$ is the desired homomorphism $(\dagger)$ from heaps and $\dagger$ to types and $\bullet$.

Theorem 4.24. Let $s$ be a stack and Let $h_1, h_2$ be heaps with $h_1 + h_2 \neq \bot$ and $(s, h_1), (s, h_2) \in M^s_A$. Then $\text{type}_A(s, h_1 + h_2) = \text{type}_A(s, h_1) \bullet \text{type}_A(s, h_2)$.

We use $\Phi$-types to define an abstraction of formulas.

Definition 4.25 ($s$-Types of a formula). Let $\phi \in \text{SL}^s_{\text{free}}$. The $s$-types of $\phi$ are given by

\[ \text{Types}^s_\Phi(\phi) := \{ \text{type}_A(s, h) \mid h \text{ heap, } (s, h) \models \phi \} \].

There are only finitely many $\Phi$-types for every fixed set of variables.

Lemma 4.26. For SID $\Phi$ and stack $s$, let $n := |\Phi| + |\text{dom}(s)|$. Then $|\text{Types}^s_\Phi(\Phi)| \leq 2^{O(n^2 \log(n))}$.

Theorem 4.24 and Lemma 4.26 make it possible to effectively compute the set of all types of the predicates of an SID. Intuitively, this can be achieved by a fixed-point computation: In the first iteration, we compute the types of the models of non-recursive rules of the SID; in later iterations, we compute the types of models of recursive rules by combining the types discovered in previous iterations by means of the composition operation, $\bullet$. We showed in [26] that this approach yields the types of all predicates in double-exponential time.

Theorem 4.27 ([26]). Let $\Phi$ be a SID, let $\text{pred}(y)$ be a predicate call, and let $s$ be a stack with $y \subseteq \text{dom}(s)$. Let $n := |\Phi| + |\text{dom}(s)|$. The set $\text{Types}^s_\Phi(\text{pred}(y))$ can be computed in $2^{O(n^2 \log(n))}$. 
5 Deciding Guarded Separation Logic

In this final technical section, we address two questions. First, how to compute the \( \Phi \)-types of arbitrary \( \text{SL}^{k}_{\text{btw}} \) formulas. Second, how to use \( \Phi \)-types to decide satisfiability and entailment.

In this section, we assume that \( \Phi \) is a fixed SID that is pointer-closed, by which me mean that \( \Phi \) contains predicates predicate \( \text{ptr}_1, \ldots, \text{ptr}_k \) defined by the rule \( \text{ptr}_k(x) \leftrightarrow x_1 \rightarrow \langle x_2, \ldots, x_{k+1} \rangle \), where \( k \) is the largest number of variables that occur on the right-hand side of points-to assertions in our satisfiability and entailment queries.

Our crucial insight is that \( \Phi \)-types (of pointer-closed SIDs) refine the satisfaction relation of \( \text{SL}^{k}_{\text{btw}} \). Models with the same types satisfy the same formulas. To show this for formulas with separating connectives, we need a way to reverse composition: we need to show that if \( \mathcal{T} = \mathcal{T}_1 \bullet \mathcal{T}_2 \), then the models of the type \( \mathcal{T} \) are a composition of models of types \( \mathcal{T}_1 \) and \( \mathcal{T}_2 \).

**Lemma 5.1** (Decomposition lemma). Let \( s \) be a stack, let \( h', h_1, h_2 \) be heaps such that \((s, h'), (s, h_1), (s, h_2) \in M^{k}_{\Phi} \) and \( \text{type}_{\Phi}(s, h') = \text{type}_{\Phi}(s, h_1 + h_2) \). Then there exist \( h'_1, h'_2 \) such that \((s, h'_1), (s, h'_2) \in M^{k}_{\Phi} \), \( h' = h'_1 + h'_2 \) and \( \text{type}_{\Phi}(s, h'_1) = \text{type}_{\Phi}(s, h'_2) \) for \( 1 \leq i \leq 2 \).

Using the decomposition lemma for the separating connectives \( \ast, \rightarrow \) and \( \oplus \), the refinement theorem can be proved by a straightforward induction.

**Theorem 5.2** (Refinement theorem). Let \( s \) be a stack and \( h_1, h_2 \) be heaps. Let \( \phi \in \text{SL}^{k}_{\text{btw}} \). Moreover, assume \( \text{type}_{\Phi}(s, h_1) = \text{type}_{\Phi}(s, h_2) \). Then \( (s, h_1) \models \phi \) iff \( (s, h_2) \models \phi \).

We rely on Theorems 4.27 and 5.2 in our algorithm for computing the types of arbitrary \( \text{SL}^{k}_{\text{btw}} \) formulas, presented in Fig. 7. We explain why the algorithm is correct.

1. The formulas \text{emp}, \( x \approx y \) and \( x \not\approx y \) only hold in empty heaps and \text{emp} is the only DUSH that holds in the empty heap.
2. To compute \( \text{type}_{\Phi}(s, \{s(a) \mapsto s(b)\}) \), we check for all rule instances of the SID whether they contain the pointer \( a \mapsto b \); if so, we take a single-tree \( \Phi \)-forest that contains this rule instance and nothing else and project this forest onto a formula. All such formulas make up the type of \( a \mapsto b \). The details of this construction are in [26].
3. The set \( \text{Types}_{\Phi}(\text{pred}(y)) \) is computable by Theorem 4.27. We present this computation in detail in [23, 26].
4. Theorem 4.24 guarantees that the types of \( \phi_1 \ast \phi_2 \) can be computed via \( \bullet \).
5. Theorem 5.2 guarantees that the Boolean operators can be implemented via set operations.
6. Magic wand and septraction are implemented by lifting their semantics from individual models to types in a straightforward way.
7. Guardedness guarantees that throughout the computation, we only need to consider guarded models. This is necessary for applying Theorems 4.24 and 5.2.

The asymptotic complexity of each operation is bounded by the size of the type abstraction. We thus obtain a double-exponential decision procedure for \( \text{SL}^{k}_{\text{btw}} \) for every fixed stack.

**Theorem 5.3.** Let \( \phi \in \text{SL}^{k}_{\text{btw}} \) with \( \text{fvars}(\phi) = x \). Let \( s \) be a stack with \( \text{dom}(s) = x \). Then (1) \( \text{Types}_{\Phi}(\phi) = \text{types}(\phi, s) \) and (2) types(\( \phi, s \)) can be computed in \( 2^{O(n^2 \log(n))} \).
We developed a 2-Exptime algorithm for deciding SL.

Theorem 5.4. Let \( \phi \) be a SID and \( \phi \in SL_{btw}^{\Phi} \). It is decidable in time \( 2^{2^{O(n^2 \log(n))}} \) whether \( \phi \) is satisfiable.

Proof. Consider the aliasing constraint of stack \( s \), \( \{(x, y) \mid x, y \in \text{dom}(s) \text{ and } s(x) = s(y)\} \). If \( s \) and \( s' \) have the same aliasing constraint, then \( Types_{\Phi}^{s}(\phi) = Types_{\Phi}^{s'}(\phi) \). Since there are only exponentially many (in \( n \)) different aliasing constraints, the claim follows from Theorem 5.3. \( \square \)

Since the entailment query \( \phi \models_{\Phi} \psi \) is equivalent to checking the unsatisfiability of \( \phi \land \lnot \psi \), and the negation in \( \phi \land \lnot \psi \) is guarded, we obtain an entailment checker with the same complexity.

Corollary 5.5. Let \( \Phi \) be a SID, \( \phi, \psi \in SL_{btw}^{\Phi} \) and \( n := |\Phi| + |\phi| + |\psi| \). The entailment problem \( \phi \models_{\Phi} \psi \) is decidable in time \( 2^{2^{O(n^2 \log(n))}} \).

6 Conclusion

We developed a 2-Exptime algorithm for deciding \( SL_{btw}^{\Phi} \), an SL in which formulas are built from user-defined predicates from SLID_{btw} [16] using \( \ast, \wedge, \lor \), and guarded forms of \( \lnot, \rightarrow, \leftarrow \). The only previously known decidability results for any SL with user-defined inductive definitions are limited to symbolic heaps, with a matching 2-Exptime bound for SLID_{btw} [26]. Further, we showed that removing the guard of any of the guarded operators of \( SL_{btw}^{\Phi} \) leads to undecidability, obtaining an almost perfect delineation between decidability and undecidability. We leave the integration of our decision procedure into a Hoare-style verification framework for future work.

References


[16] Radu Iosif, Adam Rogalewicz, and Jiri Simácek. The tree width of separation logic with recursive


[28] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic with trees and data. In Armin Biere and Roderick Bloem, editors, Computer Aided Verification - 26th International
Beyond Symbolic Heaps: Deciding Separation Logic with Inductive Definitions


