Unified Reasoning about Robustness Properties of Symbolic-Heap Separation Logic

Christina Jansen[1], Jens Katelaan[2,⋆,⋆⋆], Christoph Matheja[1,⋆⋆⋆], Thomas Noll[1], and Florian Zuleger[2,⋆⋆]

[1] Software Modeling and Verification Group, RWTH Aachen University, Germany
[2] TU Wien, Austria

Abstract. We introduce *heap automata*, a formalism for automatic reasoning about *robustness properties* of the symbolic heap fragment of separation logic with user-defined inductive predicates. Robustness properties, such as satisfiability, reachability, and acyclicity, are important for a wide range of reasoning tasks in automated program analysis and verification based on separation logic. Previously, such properties have appeared in many places in the separation logic literature, but have not been studied in a systematic manner. In this paper, we develop an algorithmic framework based on heap automata that allows us to derive asymptotically optimal decision procedures for a wide range of robustness properties in a uniform way.
We implemented a prototype of our framework and obtained promising results for all of the aforementioned robustness properties.
Further, we demonstrate the applicability of heap automata beyond robustness properties. We apply our algorithmic framework to the *model checking* and the *entailment problem* for symbolic-heap separation logic.

1 Introduction

Separation logic (SL) [38] is a popular formalism for Hoare-style verification of imperative, heap-manipulating programs. While its symbolic heap fragment originally emerged as an idiomatic form of assertions that occur naturally in hand-written proofs [35,5,4], a variety of program analyses based on symbolic-heap separation logic have been developed [5,2,16,30,34,9,22]. Consequently, it now serves as formal basis for a multitude of automated verification tools, such as [6,15,17,20,28,37,31,8], capable of proving complex properties of a program's heap, such as memory safety, for large code bases [16,15]. These tools typically rely on *systems of inductive predicate definitions (SID)* to specify the shape of data structures employed by a program, such as trees and linked lists. Originally, separation logic tools implemented highly-specialized procedures for such

⋆ Supported by by the Austrian Science Fund (FWF) under project W1255-N23.
⋆⋆ Supported by the Austrian National Research Network S11403-N23 (RiSE).
⋆⋆⋆ Supported by Deutsche Forschungsgemeinschaft (DFG) Grant NO 401/2-1.

fixed SIDs. As this limits their applicability, there is an ongoing trend to support custom SIDs that are either defined manually [28,17] or even automatically generated. The latter may, for example, be obtained from the tool CABER [12].

Robustness properties Allowing for arbitrary SIDs, however, raises various questions about their *robustness*. A user-defined or auto-generated SID might, for example, be *inconsistent*, introduce *unallocated logical variables*, specify data structures that contain undesired *cycles*, or produce *garbage*, i.e., parts of the heap that are unreachable from any program variable. Accidentally introducing such properties into specifications can have a negative impact on performance, completeness, and even soundness of the employed verification algorithms:

– Brotherston et al. [11] point out that tools might waste time on inconsistent scenarios due to *unsatisfiability* of specifications.
– The absence of unallocated logical variables, also known as *establishment*, is required by the approach of Iosif et al. [26,27] to obtain a decidable fragment of symbolic heaps.
– Other verification approaches, such as the one by Habermehl et al. [23,24], assume that no garbage is introduced by data structure specifications.
– During program analysis and verification, questions such as *reachability*, *acyclicity* and *garbage-freedom* arise depending on the properties of interest. For example, as argued by Zanardini and Genaim [39], acyclicity of the heap is crucial in automated termination proofs.

Being able to check such *robustness properties* of custom SIDs is thus crucial (1) in debugging of separation-logic specifications prior to program analysis and (2) in the program analyses themselves. So far, however, all of the above properties have either been addressed individually or not systematically at all. For example, satisfiability is studied in detail by Brotherston et al. [11], whereas establishment is often addressed with ad-hoc solutions [26,23].

Several reasoning tasks arise in the context of robustness properties. As a motivation, consider the problem of acyclicity. If our program analysis requires acyclicity, we would like to decide whether all interpretations of a symbolic heap are acyclic; if not, to find out how cycles can be introduced into the heap (counterexample generation); and, finally, to be able to generate a new SID that does guarantee acyclicity (called *refinement* below). A systematic treatment of robustness properties should cover these reasoning tasks in general, not just for the problem of acyclicity.

Problem statement We would like to develop a framework that enables:

1. *Decision procedures for robustness properties.* In program analysis, we generally deal with symbolic heaps that reference SIDs specifying unbounded data structures and thus usually have infinitely many interpretations. We need to be able to decide whether all, or some, of these infinitely many interpretations are guaranteed to satisfy a given robustness property.
2. *Generation of counterexamples* that violate a desired property.

3. *Refinement* of SIDs to automatically generate a new SID that respects a given robustness property.
4. *Automatic combination of decision procedures* to derive decision procedures for complex robustness properties from simpler ingredients.

Motivating example: Inductive reasoning about robustness properties The key insight underlying our solution to the above problems is that many properties of symbolic heaps can be decided iteratively by inductive reasoning. To motivate our approach, we illustrate this reasoning process with a concrete example. Consider an SID for acyclic singly-linked list segments with head x and tail y:

$$\mathtt{sll}(x,y) \Leftarrow \mathrm{emp} : \{x = y\} \qquad \mathtt{sll}(x,y) \Leftarrow \exists u \,.\, x \mapsto u * \mathtt{sll}(u,y) : \{x \neq y\}.$$

The two *rules* of the SID define a case distinction: A list is either empty or the first element has a successor u (specified by the *points-to assertion* $x \mapsto u$), which in turn is at the head of a (shorter) singly-linked list segment, $\mathtt{sll}(u,y)$. The inequality in the second rule guarantees that there is no cyclic model. Now, consider the following symbolic heap with *predicate calls* to $\mathtt{sll}$: $\varphi = \exists x,y,z \,.\, \mathtt{sll}(x,z) * z \mapsto y * \mathtt{sll}(y,x)$, which might appear as an assertion during program analysis. Say our program analysis depends on the acyclicity of φ, so we need to determine whether φ is acyclic. We can do so by *inductive reasoning* as follows.

– We analyze the call $\mathtt{sll}(x,z)$, the first list segment in the symbolic heap φ. If it is interpreted by the right-hand side of the first rule of the SID from above, then there is no cycle in $\mathtt{sll}(x,z)$ and z is reachable from x.
– If we already know for a call $\mathtt{sll}(u,z)$ that all of its models are acyclic structures and that z is reachable from u, then z is also reachable from x in the symbolic heap $\exists u \,.\, x \mapsto u * \mathtt{sll}(u,z) : \{x \neq z\}$ obtained by the second rule of the SID. Since our SID does not introduce dangling pointers, we also know that there is still no cycle.
– By induction, $\mathtt{sll}(x,z)$ is thus acyclic and z is reachable from x.
– Likewise, $\mathtt{sll}(y,x)$ is acyclic and x is reachable from y.
– Now, based on the information we discovered for $\mathtt{sll}(x,z)$ and $\mathtt{sll}(y,x)$, we examine φ and conclude that it is *cyclic*, as z is reachable from x, y is reachable from z, and x is reachable from y. Crucially, we reason inductively and thus do *not* re-examine the list segments to arrive at our conclusion.

In summary, we examine a symbolic heap and corresponding SID *bottom-up*, starting from the non-recursive base case. Moreover, at each stage of this analysis, we remember a fixed amount of information—namely what we discover about reachability between parameters and acyclicity of every symbolic heap we examine. Similar inductive constructions are defined explicitly for various robustness properties throughout the separation logic literature [11,13,26]. Our aim is to generalize such manual constructions following an *automata-theoretic approach*: We introduce automata that operate on symbolic heaps and store the relevant information of each symbolic heap they examine in their state space. Whenever such an automaton comes across a predicate that it has already analyzed, it can simply replace the predicate with the information that is encoded in the corresponding state. In other words, our automata *recognize* robustness properties *in a compositional way* by exploiting the *inductive structure* inherent in the SIDs.

Systematic reasoning about robustness properties Our novel automaton model, *heap automata*, works directly on the structure of symbolic heaps as outlined in the example, and can be applied to all the problems introduced before. In particular, heap automata enable *automatic refinement* of SIDs and enjoy a variety of closure properties through which we can derive counterexample generation as well as decision procedures for various robustness properties—including satisfiability, establishment, reachability, garbage-freedom, and acyclicity.

Our approach can thus be seen as an *algorithmic framework* for deciding a wide range of robustness properties of symbolic heaps. Furthermore, we show asymptotically optimal complexity of our automata-based decision procedures in a uniform way. By enabling this systematic approach to reasoning about robustness, our framework generalizes prior work that studied single robustness properties in isolation, such as the work by Brotherston et al. [11,13].

As a natural byproduct of our automata-based approach, we also derive decision procedures for the *model-checking problem*, which was recently studied, and proven to be ExpTime–complete in general, by Brotherston et al. [13]. This makes it possible to apply our framework to *run-time verification*—a setting in which robustness properties are of particular importance [33,28,13].

Entailment checking with heap automata Finally, we also address the *entailment problem.* In Hoare-style program analysis, decision procedures for the *entailment problem* become essential to discharge implications between assertions, as required, for example, by the rule of consequence [25]. Because of this central role in verification, there is an extensive body of research on decision procedures for entailment; see, for example [3,10,14,26,27,32,36,21]. Antonopoulos et al. [1] study the complexity of the entailment problem and show that it is undecidable in general, and already ExpTime–hard for SIDs specifying sets of trees.

We use heap automata to check entailment between *determined* symbolic heaps. Intuitively, determinedness is a strong form of the establishment property guaranteeing that two variables are either equal or unequal in every model. Unlike other decision procedures [26,27,3], our approach does not impose syntactic restrictions on the symbolic heap under consideration but merely requires that suitable heap automata for the predicates on the right-hand side of the entailment are provided. In particular, we show how to obtain ExpTime decision procedures from such heap automata—which exist for highly non-trivial SIDs. If desired, additional syntactic restrictions can be integrated seamlessly into our approach to boost our algorithms' performance.

Contributions Our main contributions can be summarized as follows.

- We introduce *heap automata*, a novel automaton model operating directly on symbolic heaps. We prove that heap automata enjoy various useful *closure* *properties*. Besides union, intersection and complement, they are closed under the conjunction with pure formulas, allowing the construction of complex heap automata from simple ones.

– We develop a powerful *algorithmic framework* for automated reasoning about and debugging of symbolic heaps with inductive predicate definitions based on heap automata.
– We show that key robustness properties, such as *satisfiability*, *establishment*, *reachability*, *garbage freedom* and *acyclicity*, can naturally be expressed as heap automata. Moreover, the upper bounds of decision procedures obtained from our framework are shown to be optimal—i.e., ExpTime–complete—in each of these cases. Further, they enable automated *refinement* of SIDs to filter out (or expose) symbolic heaps with undesired properties.
– Additionally, we apply heap automata to tackle the *entailment* and the *model checking* problem for symbolic heaps. We show that if each predicate of an SID can be represented by a heap automaton, then the entailment problem for the corresponding fragment of symbolic heaps is decidable in 2-ExpTime in general and ExpTime-complete if the maximal arity of predicates and points-to assertions is bounded. For example, our framework yields an ExpTime decision procedure for a symbolic heap fragment capable of representing trees with linked leaves—a fragment that is out of scope of most ExpTime decision procedures known so far (cf. [3,21,27]).
– We implemented a prototype of our framework that yields promising results for all robustness properties considered in the paper.

Organization of the paper The fragment of symbolic heaps with inductive predicate definitions is briefly introduced in Section 2. Heap automata and derived decision procedures are studied in Section 3. Section 4 demonstrates that a variety of robustness properties can be checked by heap automata. We report on a prototypical implementation of our framework in Section 5. Special attention to the entailment problem is paid in Section 6. Finally, Section 7 concludes. Due to lack of space, most proofs as well as detailed construction are provided in a full version of this paper that is available online [29].

2 Symbolic Heaps

This section briefly introduces the symbolic heap fragment of separation logic equipped with inductive predicate definitions.

Basic Notation $\mathbb{N}$ is the set of natural numbers and 2^S is the powerset of a set S. $(co)\mathrm{dom}(f)$ is the (co)domain of a (partial) function f. We abbreviate tuples $(u_1, \ldots, u_n)$, $n \geq 0$, by $\mathbf{u}$ and write $\mathbf{u}[i]$, $1 \leq i \leq \|\mathbf{u}\| = n$, to denote u_i, the i-th element of $\mathbf{u}$. By slight abuse of notation, the same symbol $\mathbf{u}$ is used for the set of all elements occurring in tuple $\mathbf{u}$. The empty tuple is ε and the set of all (non-empty) tuples [of length $n \geq 0$] over a finite set S is S^* (S^+ [S^n]). The concatenation of tuples $\mathbf{u}$ and $\mathbf{v}$ is $\mathbf{u}\,\mathbf{v}$.

Syntax We usually denote *variables* taken from Var (including a dedicated variable **null**) by a, b, c, x, y, z, etc. Moreover, let Pred be a set of *predicate symbols* and $\mathrm{ar} : \mathrm{Pred} \to \mathbb{N}$ be a function assigning each symbol its *arity*. *Spatial formulas* Σ and *pure formulas* π are given by the following grammar:

$$\Sigma \;::=\; \mathrm{emp} \mid x \mapsto \mathbf{y} \mid \Sigma * \Sigma \qquad \pi \;::=\; x = y \mid x \neq y,$$

where $\mathbf{y}$ is a non-empty tuple of variables. Here, emp stands for the *empty heap*, $x \mapsto \mathbf{y}$ is a *points-to assertion* and $*$ is the *separating conjunction*. Furthermore, for $P \in \mathrm{Pred}$ and a tuple of variables $\mathbf{y}$ of length $\mathrm{ar}(P)$, $P\mathbf{y}$ is a *predicate call*. A *symbolic heap* $\varphi(\mathbf{x}_0)$ with variables $Var(\varphi)$ and free variables $\mathbf{x}_0 \subseteq Var(\varphi)$ is a formula of the form $\varphi(\mathbf{x}_0) = \exists \mathbf{z} \,.\, \Sigma * \Gamma : \Pi$, $\Gamma = P_1\mathbf{x}_1 * \ldots * P_m\mathbf{x}_m$, where Σ is a spatial formula, Γ is a sequence of predicate calls and Π is a finite set of pure formulas, each with variables from $\mathbf{x}_0$ and $\mathbf{z}$. This normal form, in which predicate calls and points-to assertions are never mixed, is chosen to simplify formal constructions. If an element of a symbolic heap is empty, we usually omit it to improve readability. For the same reason, we fix the notation from above and write $\mathbf{z}^{\varphi}$, $\mathbf{x}_i^{\varphi}$, Σ^{φ} etc. to denote the respective component of symbolic heap φ in formal constructions. Hence, $\|\mathbf{x}_0^{\varphi}\|$ and $\|\Gamma^{\varphi}\|$ refer to the number of free variables and the number of predicate calls of φ, respectively. We omit the superscript whenever the symbolic heap under consideration is clear from the context. If a symbolic heap τ contains no predicate calls, i.e., $\|\Gamma^{\tau}\| = 0$, then τ is called *reduced*. Moreover, to simplify the technical development, we tacitly assume that **null** is a free variable that is passed to every predicate call. Thus, for each $i \in \mathbb{N}$, we write $\mathbf{x}_i[0]$ as a shortcut for **null** and treat $\mathbf{x}_i[0]$ as if $\mathbf{x}_i[0] \in \mathbf{x}_i$.[3]

Systems of Inductive Definitions Every predicate symbol is associated with one or more symbolic heaps by a *system of inductive definitions* (SID). Formally, an SID is a finite set of rules of the form $P\mathbf{x}_0 \Leftarrow \varphi$, where φ is a symbolic heap with $\mathrm{ar}(P) = \|\mathbf{x}_0^{\varphi}\|$. The set of all predicate symbols occurring in SID Φ and their maximal arity are denoted by $\mathrm{Pred}(\Phi)$ and $\mathrm{ar}(\Phi)$, respectively.

Example 1. An SID specifying doubly-linked list segments is defined by:

$$\begin{aligned}
\mathtt{dll}(x_1, x_2, x_3, x_4) &\Leftarrow \mathrm{emp} : \{x_1 = x_3, x_2 = x_4\} \\
\mathtt{dll}(x_1, x_2, x_3, x_4) &\Leftarrow \exists u \,.\, x_1 \mapsto (u, x_2) * \mathtt{dll}(u, x_1, x_3, x_4),
\end{aligned}$$

where x_1 corresponds to the *head* of the list, x_2 and x_3 represent the *previous* and the *next* list element and x_4 represents the *tail* of the list. Further, the following rules specify binary trees with *root* x_1, *leftmost leaf* x_2 and *successor of the rightmost leaf* x_3 in which all leaves are connected by a singly-linked list from left to right.

$$\begin{aligned}
\mathtt{tll}(x_1, x_2, x_3) &\Leftarrow x_1 \mapsto (\mathbf{null}, \mathbf{null}, x_3) : \{x_1 = x_2\} \\
\mathtt{tll}(x_1, x_2, x_3) &\Leftarrow \exists \ell\, r\, z \,.\, x_1 \mapsto (\ell, r, \mathbf{null}) * \mathtt{tll}(\ell, x_2, z) * \mathtt{tll}(r, z, x_3).
\end{aligned}$$

[3] Since $\mathbf{x}_i[0]$ is just a shortcut and not a proper variable, $\|\mathbf{x}_i\|$ refers to the number of variables in $\mathbf{x}_i$ *apart from* $\mathbf{x}_i[0]$.

$$
\begin{array}{lll}
s, h \models_{\Phi} x \sim y & \Leftrightarrow & s(x) \sim s(y), \text{ where } \sim \in \{=, \neq\} \\
s, h \models_{\Phi} \mathrm{emp} & \Leftrightarrow & \mathrm{dom}(h) = \emptyset \\
s, h \models_{\Phi} x \mapsto \mathbf{y} & \Leftrightarrow & \mathrm{dom}(h) = \{s(x)\} \text{ and } h(s(x)) = s(\mathbf{y}) \\
s, h \models_{\Phi} P\mathbf{y} & \Leftrightarrow & \exists \tau \in \mathbb{U}_{\Phi}(P\mathbf{y}) \,.\, s, h \models_{\emptyset} \tau \\
s, h \models_{\Phi} \varphi * \psi & \Leftrightarrow & \exists h_1, h_2 \,.\, h = h_1 \uplus h_2 \\
 & & \quad \text{and } s, h_1 \models_{\Phi} \varphi \text{ and } s, h_2 \models_{\Phi} \psi \\
s, h \models_{\Phi} \exists \mathbf{z}. \Sigma * \Gamma : \Pi & \Leftrightarrow & \exists \mathbf{v} \in \mathit{Val}^{\|\mathbf{z}\|} \,.\, s\left[\mathbf{z} \mapsto \mathbf{v}\right], h \models_{\Phi} \Sigma * \Gamma \\
 & & \quad \text{and } \forall \pi \in \Pi \,.\, s\left[\mathbf{z} \mapsto \mathbf{v}\right], h \models_{\Phi} \pi
\end{array}
$$

Fig. 1. Semantics of the symbolic heap fragment of separation logic with respect to an SID Φ and a state (s, h).

Definition 1. *We write* SH *for the set of all symbolic heaps and* SH^{Φ} *for the set of symbolic heaps restricted to predicate symbols taken from SID Φ. Moreover, given a computable function $\mathcal{C} : \mathrm{SH} \to \{0, 1\}$, the set of symbolic heaps $\mathrm{SH}_{\mathcal{C}}$ is given by $\mathrm{SH}_{\mathcal{C}} \triangleq \{\varphi \in \mathrm{SH} \mid \mathcal{C}(\varphi) = 1\}$. We collect all SIDs in which every right-hand side belongs to $\mathrm{SH}_{\mathcal{C}}$ in $\mathrm{SID}_{\mathcal{C}}$. To refer to the set of all reduced symbolic heaps (belonging to a set defined by $\mathcal{C}$), we write* RSH *($\mathrm{RSH}_{\mathcal{C}}$).*

Example 2. Let $\alpha \in \mathbb{N}$ and $\mathrm{FV}^{\leq \alpha}(\varphi) \triangleq \begin{cases} 1, & \|\mathbf{x}_0^{\varphi}\| \leq \alpha \\ 0, & \text{otherwise} \end{cases}$.

Clearly, $\mathrm{FV}^{\leq \alpha}$ is computable. Moreover, $\mathrm{SH}_{\mathrm{FV}^{\leq \alpha}}$ is the set of all symbolic heaps having at most α free variables.

Semantics As in a typical RAM model, we assume heaps to consist of records with a finite number of fields. Let *Val* denote an infinite set of *values* and $\mathit{Loc} \subseteq \mathit{Val}$ an infinite set of addressable *locations*. Moreover, we assume the existence of a special non-addressable value $\mathbf{null} \in \mathit{Val} \setminus \mathit{Loc}$.

A *heap* is a finite partial function $h : \mathit{Loc} \rightharpoonup \mathit{Val}^{+}$ mapping locations to non-empty tuples of values. We write $h_1 \uplus h_2$ to denote the union of heaps h_1 and h_2 provided that $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$. Otherwise, $h_1 \uplus h_2$ is undefined. Variables are interpreted by a *stack*, i.e., a partial function $s : \mathit{Var} \rightharpoonup \mathit{Val}$ with $s(\mathbf{null}) = \mathbf{null}$. Furthermore, stacks are canonically extended to tuples of variables by componentwise application. We call a stack–heap pair (s, h) a *state*. The set of all states is *States*. The semantics of a symbolic heap with respect to an SID and a state is shown in Figure 1. Note that the semantics of predicate calls is explained in detail next.

Unfoldings of Predicate Calls The semantics of predicate calls is defined in terms of *unfolding trees*. Intuitively, an unfolding tree specifies how predicate calls are *replaced* by symbolic heaps according to a given SID. The resulting reduced symbolic heap obtained from an unfolding tree is consequently called an *unfolding*.

Formally, let $\varphi = \exists \mathbf{z} . \Sigma * P_1 \mathbf{x}_1 * \ldots * P_m \mathbf{x}_m : \Pi$. Then a predicate call $P_i \mathbf{x}_i$ may be *replaced* by a reduced symbolic heap τ if $\|\mathbf{x}_i\| = \|\mathbf{x}_0^\tau\|$ and $\mathit{Var}(\varphi) \cap \mathit{Var}(\tau) \subseteq \mathbf{x}_0^\tau$. The result of such a replacement is

$$\begin{aligned}\varphi\,[P_i/\tau] \;\triangleq\; & \exists \mathbf{z}\, \mathbf{z}^\tau . \Sigma * \Sigma^{\tau[\mathbf{x}_0^\tau/\mathbf{x}_i]} * \\ & P_1\mathbf{x}_1 * \ldots * P_{i-1}\mathbf{x}_{i-1} * P_{i+1}\mathbf{x}_{i+1} * \ldots * P_m\mathbf{x}_m \;:\; \left(\Pi \cup \Pi^{\tau[\mathbf{x}_0^\tau/\mathbf{x}_i]}\right),\end{aligned}$$

where $\tau\,[\mathbf{x}_0^\tau/\mathbf{x}_i]$ denotes the substitution of each free variable of τ by the corresponding parameter of P_i.

A *tree* over symbolic heaps SH^Φ is a finite partial function $t : \mathbb{N}^* \rightharpoonup \mathrm{SH}^\Phi$ such that $\emptyset \neq \mathrm{dom}(t) \subseteq \mathbb{N}^*$ is prefix-closed and for all $\mathbf{u} \in \mathrm{dom}(t)$ with $t(\mathbf{u}) = \varphi$, we have $\{1, \ldots, \|\Gamma^\varphi\|\} = \{i \in \mathbb{N} \mid \mathbf{u}\, i \in \mathrm{dom}(t)\}$. The element $\varepsilon \in \mathrm{dom}(t)$ is called the *root* of tree t. Furthermore, the *subtree* $t|_\mathbf{u}$ of t with root $\mathbf{u}$ is $t|_\mathbf{u} : \{\mathbf{v} \mid \mathbf{u}\,\mathbf{v} \in \mathrm{dom}(t)\} \to \mathrm{SH}^\Phi$ with $t|_\mathbf{u}(\mathbf{v}) \triangleq t(\mathbf{u} \cdot \mathbf{v})$.

Definition 2. *Let $\Phi \in \mathrm{SID}$ and $\varphi \in \mathrm{SH}^\Phi$. Then the set of* unfolding trees *of φ w.r.t. Φ, written $\mathbb{T}_\Phi(\varphi)$, is the least set that contains all trees t that satisfy (1) $t(\varepsilon) = \varphi$ and (2) $t|_i \in \mathbb{T}_\Phi(\psi_i)$ for each $1 \leq i \leq \|\Gamma^\varphi\|$, where $P_i^\varphi \Leftarrow \psi_i \in \Phi$.*

Note that for every reduced symbolic heap τ, we have $\|\Gamma^\tau\| = 0$. Thus, $\mathbb{T}_\Phi(\tau) = \{t\}$, where $t : \{\varepsilon\} \to \{\tau\} : \varepsilon \mapsto \tau$, forms the base case in Definition 2. Every unfolding tree t specifies a reduced symbolic heap $[\![t]\!]$, which is obtained by recursively replacing predicate calls by reduced symbolic heaps:

Definition 3. *The* unfolding *of an unfolding tree $t \in \mathbb{T}_\Phi(\varphi)$ is*

$$[\![t]\!] \;\triangleq\; \begin{cases} t(\varepsilon) & , \; \|\Gamma^{t(\varepsilon)}\| = 0 \\ t(\varepsilon)\,[P_1/[\![t|_1]\!], \ldots, P_m/[\![t|_m]\!]] & , \; \|\Gamma^{t(\varepsilon)}\| = m > 0 \;, \end{cases}$$

where we tacitly assume that the variables $\mathbf{z}^{t(\varepsilon)}$, i.e., the existentially quantified variables in $t(\varepsilon)$, are substituted by fresh variables.

Example 3. Recall from Example 1 the two symbolic heaps τ (upper) and φ (lower) occurring on the right-hand side of the `dll` predicate. Then $t : \{\varepsilon, 1\} \to \{\varphi, \tau\} : \varepsilon \mapsto \varphi, 1 \mapsto \tau$ is an unfolding tree of φ. The corresponding unfolding is

$$[\![t]\!] = \varphi\,[P_1^\varphi/\tau] \;=\; \exists z \;.\; x_1 \mapsto (z, x_2) * \mathrm{emp} \;:\; \{z = x_3, x_1 = x_4\}.$$

Definition 4. *The set of all* unfoldings *of a predicate call $P_i \mathbf{x}_i$ w.r.t. an SID Φ is denoted by $\mathbb{U}_\Phi(P_i \mathbf{x}_i)$. Analogously, the* unfoldings *of a symbolic heap φ are $\mathbb{U}_\Phi(\varphi) \;\triangleq\; \{[\![t]\!] \mid t \in \mathbb{T}_\Phi(\varphi)\}$.*

Then, as already depicted in Figure 1, the semantics of predicate calls requires the existence of an unfolding satisfying a given state. This semantics corresponds to a particular iteration of the frequently used semantics of predicate calls based on least fixed points (cf. [11]). Further note that applying the SL semantics to a given symbolic heap coincides with applying them to a suitable unfolding.

Lemma 1. *Let $\varphi \in \mathrm{SH}^\Phi$. Then, for every $(s, h) \in \mathit{States}$, we have*

$$s, h \models_\Phi \varphi \;\textit{iff}\; \exists \tau \in \mathbb{U}_\Phi(\varphi) \;.\; s, h \models_\emptyset \tau.$$

3 Heap Automata

In this section we develop a procedure to reason about robustness properties of symbolic heaps. This procedure relies on the notion of *heap automata*; a device that assigns one of finitely many states to any given symbolic heap.

Definition 5. *A* heap automaton *over* $\mathrm{SH}_{\mathcal{C}}$ *is a tuple* $\mathfrak{A} = (Q, \mathrm{SH}_{\mathcal{C}}, \Delta, F)$*, where* Q *is a finite set of* states *and* $F \subseteq Q$ *is a set of* final states*, respectively. Moreover,* $\Delta \subseteq Q^* \times \mathrm{SH}_{\mathcal{C}} \times Q$ *is a decidable* transition relation *such that* $(\mathbf{q}, \varphi, p) \in \Delta$ *implies that* $\|\mathbf{q}\| = \|\Gamma^{\varphi}\|$*. We often write* $\mathbf{q} \xrightarrow{\varphi}_{\mathfrak{A}} p$ *instead of* $(\mathbf{q}, \varphi, p) \in \Delta$*.*

A transition $\mathbf{q} \xrightarrow{\varphi}_{\mathfrak{A}} p$ takes a symbolic heap φ and an input state q_i for every predicate call P_i of φ—collected in the tuple $\mathbf{q}$—and assigns an output state p to φ. Thus, the intuition behind a transition is that φ has a property encoded by state p if every predicate call P_i of φ is replaced by a reduced symbolic heap τ_i that has a property encoded by state $\mathbf{q}[i]$.

Note that every heap automaton $\mathfrak{A}$ assigns a state p to a reduced symbolic heap τ within a single transition of the form $\varepsilon \xrightarrow{\tau}_{\mathfrak{A}} p$. Alternatively, $\mathfrak{A}$ may process a corresponding unfolding tree t with $[\![t]\!] = \tau$. In this case, $\mathfrak{A}$ proceeds similarly to the compositional construction of unfoldings (see Definition 3). However, instead of replacing every predicate call P_i of the symbolic heap $t(\varepsilon)$ at the root of t by an unfolding $[\![t|_i]\!]$ of a subtree of t, $\mathfrak{A}$ uses states to keep track of the properties of these unfolded subtrees. Consequently, $\mathfrak{A}$ assigns a state p to the symbolic heap $t(\varepsilon)$ if $(q_1, \ldots, q_m) \xrightarrow{t(\varepsilon)}_{\mathfrak{A}} p$ holds, where for each $1 \leq i \leq m$, q_i is the state assigned to the unfolding of subtree $t|_i$, i.e., there is a transition $\varepsilon \xrightarrow{[\![t|_i]\!]}_{\mathfrak{A}} q_i$. It is then natural to require that p should coincide with the state assigned directly to the unfolding $[\![t]\!]$, i.e., $\varepsilon \xrightarrow{[\![t]\!]}_{\mathfrak{A}} p$. Hence, we require all heap automata considered in this paper to satisfy a *compositionality property.*

Definition 6. *A heap automaton* $\mathfrak{A} = (Q, \mathrm{SH}_{\mathcal{C}}, \Delta, F)$ *is* compositional *if for every* $p \in Q$*, every* $\varphi \in \mathrm{SH}_{\mathcal{C}}$ *with* $m \geq 0$ *predicate calls* $\Gamma^{\varphi} = P_1\mathbf{x}_1 * \ldots * P_m\mathbf{x}_m$*, and all reduced symbolic heaps* $\tau_1, \ldots, \tau_m \in \mathrm{RSH}_{\mathcal{C}}$*, we have:*

$$\begin{gathered}\exists \mathbf{q} \in Q^m \,.\, (\mathbf{q}, \varphi, p) \in \Delta \text{ and } \textstyle\bigwedge_{1 \leq i \leq m} (\varepsilon, \tau_i, \mathbf{q}[i]) \in \Delta \\ \text{if and only if} \\ (\varepsilon,\ \varphi\,[P_1/\tau_1, \ldots, P_m/\tau_m]\,,\ p)\ \in\ \Delta.\end{gathered}$$

Due to the compositionality property, we can safely define the *language* $L(\mathfrak{A})$ *accepted* by a heap automaton $\mathfrak{A}$ as the set of all reduced symbolic heaps that are assigned a final state, i.e., $L(\mathfrak{A}) \triangleq \{\tau \in \mathrm{RSH}_{\mathcal{C}} \mid \exists q \in F \,.\, \varepsilon \xrightarrow{\tau}_{\mathfrak{A}} q\}$.

Example 4. Given a symbolic heap φ, let $|\Sigma^{\varphi}|$ denote the number of points-to assertions in φ. As a running example, we consider a heap automaton $\mathfrak{A} = (\{0,1\}, \mathrm{SH}, \Delta, \{1\})$, where Δ is given by

$$\mathbf{q} \xrightarrow{\varphi}_{\mathfrak{A}} p \text{ iff } p = \begin{cases} 1, & \text{if } |\Sigma^{\varphi}| + \sum_{i=1}^{\|\mathbf{q}\|} \mathbf{q}[i] > 0 \\ 0, & \text{otherwise.} \end{cases}$$

While $\mathfrak{A}$ is a toy example, it illustrates the compositionality property: Consider the reduced symbolic heap $\tau(x,y) = \exists z.\mathrm{emp} * \mathrm{emp} : \{x = z, z = y\}$. Since τ contains no points-to assertions, $\mathfrak{A}$ rejects τ in a single step, i.e., $\varepsilon \xrightarrow{\tau}_{\mathfrak{A}} 0 \notin \{1\}$. The compositionality property of $\mathfrak{A}$ ensures that $\mathfrak{A}$ yields the same result for every unfolding tree t whose unfolding $[\![t]\!]$ is equal to τ. For instance, τ is a possible unfolding of the symbolic heap $\varphi(x,y) = \exists z.\mathtt{sll}(x,z) * \mathtt{sll}(z,y)$, where $\mathtt{sll}$ is a predicate specifying singly-linked list segments as in Section 1. More precisely, if both predicate calls are replaced according to the rule $\mathtt{sll}(x,y) \Leftarrow \mathrm{emp} : \{x = y\}$, we obtain τ again (up to renaming of parameters as per Definition 3). In this case, $\mathfrak{A}$ rejects as before: We have $\varepsilon \xrightarrow{\mathrm{emp}:\{x=y\}}_{\mathfrak{A}} 0$ for both base cases and $(0,0) \xrightarrow{\varphi}_{\mathfrak{A}} 0$ for the symbolic heap φ. By the compositionality property, this is equivalent to $\varepsilon \xrightarrow{\tau}_{\mathfrak{A}} 0$. Analogously, if a predicate call, say the first, is replaced according to the rule $\mathtt{sll}(x,y) \Leftarrow \psi$, where $\psi = \exists z.x \mapsto z * \mathtt{sll}(z,y)$, then $0 \xrightarrow{\psi}_{\mathfrak{A}} 1$, $1 \xrightarrow{\psi}_{\mathfrak{A}} 1$ and $(1,0) \xrightarrow{\varphi}_{\mathfrak{A}} 1$ holds, i.e., $\mathfrak{A}$ accepts. In general, $L(\mathfrak{A})$ is the set of all reduced symbolic heaps that contain at least one points-to assertion.

While heap automata can be applied to check whether a single reduced symbolic heap has a property of interest, i.e., belongs to the language of a heap automaton, our main application is directed towards reasoning about *infinite* sets of symbolic heaps, such as all unfoldings of a symbolic heap φ. Thus, given a heap automaton $\mathfrak{A}$, we would like to answer the following questions:

1. Does there *exist* an unfolding of φ that is accepted by $\mathfrak{A}$?
2. Are *all* unfoldings of φ accepted by $\mathfrak{A}$?

We start with a special case of the first question in which φ is a single predicate call. The key idea behind our corresponding decision procedure is to transform the SID Φ to *filter out* all unfoldings that are *not* accepted by $\mathfrak{A}$. One of our main results is that such a refinement is always possible.

Theorem 1 (Refinement Theorem). *Let $\mathfrak{A}$ be a heap automaton over* $\mathrm{SH}_{\mathcal{C}}$ *and $\Phi \in \mathrm{SID}_{\mathcal{C}}$. Then one can effectively construct a refined $\Psi \in \mathrm{SID}_{\mathcal{C}}$ such that for each $P \in \mathrm{Pred}(\Phi)$, we have $\mathbb{U}_{\Psi}(P\mathbf{x}_0) = \mathbb{U}_{\Phi}(P\mathbf{x}_0) \cap L(\mathfrak{A})$.*

Proof. We construct $\Psi \in \mathrm{SID}_{\mathcal{C}}$ over the predicate symbols $\mathrm{Pred}(\Psi) = (\mathrm{Pred}(\Phi) \times Q_{\mathfrak{A}}) \cup \mathrm{Pred}(\Phi)$ as follows: If $P\mathbf{x}_0 \Leftarrow \varphi \in \Phi$ with $\Gamma^{\varphi} = P_1\mathbf{x}_1 * \ldots * P_m\mathbf{x}_m$, $m \geq 0$, and $(q_1, \ldots, q_m) \xrightarrow{\varphi}_{\mathfrak{A}} q_0$, we add a rule to Ψ in which P is substituted by $\langle P, q_0 \rangle$ and each predicate call $P_i\mathbf{x}_i$ is substituted by a call $\langle P_i, q_i \rangle \mathbf{x}_i$. Furthermore, for each $q \in F_{\mathfrak{A}}$, we add a rule $P\mathbf{x}_0 \Leftarrow \langle P, q \rangle \mathbf{x}_0$ to Ψ. See [29] for details. $\square$

Example 5. Applying the refinement theorem to the heap automaton from Example 4 and the SID from Example 1 yields a refined SID given by the rules:

$$\begin{aligned}
\mathtt{dll}\,\mathbf{x}_0 &\Leftarrow \langle \mathtt{dll}, 1 \rangle\,\mathbf{x}_0 \qquad\qquad \langle \mathtt{dll}, 0 \rangle \mathbf{x}_0 \Leftarrow \mathrm{emp} : \{x_1 = x_3, x_2 = x_4\} \\
\langle \mathtt{dll}, 1 \rangle \mathbf{x}_0 &\Leftarrow \exists z \;.\; x_1 \mapsto (z, x_2) * \langle \mathtt{dll}, 0 \rangle (z, x_1, x_3, x_4) \\
\langle \mathtt{dll}, 1 \rangle \mathbf{x}_0 &\Leftarrow \exists z \;.\; x_1 \mapsto (z, x_2) * \langle \mathtt{dll}, 1 \rangle (z, x_1, x_3, x_4)
\end{aligned}$$

Hence, the refined predicate $\mathtt{dll}\,\mathbf{x}_0$ specifies all non-empty doubly-linked lists.

```
Input  : SID $\Phi$, $I \in \mathrm{Pred}(\Phi)$, $\mathfrak{A} = (Q, \mathrm{SH}_{\mathcal{C}}, \Delta, F)$
Output: yes iff $\mathbb{U}_{\Phi}(I\mathbf{x}) \cap L(\mathfrak{A}) = \emptyset$

$\mathsf{R} \leftarrow \emptyset$;
repeat
    if $\mathsf{R} \cap (\{I\} \times F) \neq \emptyset$ then return $no$;
    pick a state $q$ in $Q$; pick a rule $P \Leftarrow \varphi$ in $\Phi$;
    $\mathsf{s} \leftarrow \varepsilon$; // list of states of $\mathfrak{A}$
    for $i$ in 1 to $\|\Gamma^{\varphi}\|$ do
        pick $(P_i^{\varphi}, p) \in \mathsf{R}$; append($\mathsf{s}$,$p$)  // base case if $\|\Gamma^{\varphi}\| = 0$
    end
    if $(\mathsf{s}, \varphi, q) \in \Delta$ then $\mathsf{R} \leftarrow \mathsf{R} \cup \{(P, q)\}$ ;
until $\mathsf{R}$ reaches a fixed point (w.r.t. all choices of rules);
return $yes$
```

Algorithm 1: On-the-fly construction of a refined SID with emptiness check.

To answer question (1) we then check whether the set of unfoldings of a refined SID is non-empty. This boils down to a simple reachability analysis.

Lemma 2. *Given an SID Φ and a predicate symbol $P \in \mathrm{Pred}(\Phi)$, it is decidable in linear time whether the set of unfoldings of P is empty, i.e., $\mathbb{U}_{\Phi}(P\mathbf{x}) = \emptyset$.*

Proof (sketch). It suffices to check whether the predicate P lies in the least set R such that (1) $I \in R$ if $I\mathbf{x}_0 \Leftarrow \tau \in \Phi$ for some $\tau \in \mathrm{RSH}$, and (2) $I \in R$ if $I\mathbf{x}_0 \Leftarrow \varphi \in \Phi$ and for each $P_i^{\varphi}\mathbf{x}_i^{\varphi}$, $1 \leq i \leq \|\Gamma^{\varphi}\|$, $P_i^{\varphi} \in R$. The set R is computable in linear time by a simple backward reachability analysis. $\square$

As outlined before, putting the Refinement Theorem and Lemma 2 together immediately yields a decision procedure for checking whether some unfolding of a predicate symbol P is accepted by a heap automaton: Construct the refined SID and subsequently check whether the set of unfoldings of P is non-empty.

To extend this result from unfoldings of single predicates to unfoldings of arbitrary symbolic heaps φ, we just add a rule $P \Leftarrow \varphi$, where P is a fresh predicate symbol, and proceed as before.

Corollary 1. *Let $\mathfrak{A}$ be a heap automaton over $\mathrm{SH}_{\mathcal{C}}$ and $\Phi \in \mathrm{SID}_{\mathcal{C}}$. Then, for each $\varphi \in \mathrm{SH}_{\mathcal{C}}^{\Phi}$, it is decidable whether there exists $\tau \in \mathbb{U}_{\Phi}(\varphi)$ such that $\tau \in L(\mathfrak{A})$.*

The refinement and emptiness check can also be integrated: Algorithm 1 displays a simple procedure that constructs the refined SID Ψ from Theorem 1 on-the-fly while checking whether its set of unfoldings is empty for a given predicate symbol. Regarding complexity, the size of a refined SID[4] obtained from an SID Φ and a heap automaton $\mathfrak{A}$ is bounded by $\|\Phi\| \cdot \|Q_{\mathfrak{A}}\|^{M+1}$, where M is the maximal number of predicate calls occurring in any rule of Φ. Thus, the

[4] We assume a reasonable function $\|.\|$ assigning a size to SIDs, symbolic heaps, unfolding trees, etc. For instance, the size $\|\Phi\|$ of an SID Φ is given by the product of its number of rules and the size of the largest symbolic heap contained in any rule.

aforementioned algorithm runs in time $\mathcal{O}\left(\|\Phi\| \cdot \|Q_{\mathfrak{A}}\|^{M+1} \cdot \|\Delta_{\mathfrak{A}}\|\right)$, where $\|\Delta_{\mathfrak{A}}\|$ denotes the complexity of deciding whether the transition relation $\Delta_{\mathfrak{A}}$ holds for a given tuple of states and a symbolic heap occurring in a rule of Φ.

Example 6. Resuming our toy example, we check whether some unfolding of the doubly-linked list predicate $\texttt{dll}\,\mathbf{x}_0$ (see Example 1) contains points-to assertions. Formally, we decide whether $\mathbb{U}_{\Phi}(\texttt{dll}\,\mathbf{x}_0) \cap L(\mathfrak{A}) \neq \emptyset$, where $\mathfrak{A}$ is the heap automaton introduced in Example 4. Algorithm 1 first picks the rule that maps $\texttt{dll}$ to the empty list segment and consequently adds $\langle \texttt{dll}, 0\rangle$ to the set R of reachable predicate–state pairs. In the next iteration, it picks the rule that maps to the non-empty list. Since $\langle \texttt{dll}, 0\rangle \in R$, s is set to 0 in the **do**-loop. Abbreviating the body of the rule to φ, we have $(0, \varphi, 1) \in \Delta$, so the algorithm adds $\langle \texttt{dll}, 1\rangle$ to R. After that, *no* is returned, because 1 is a final state of $\mathfrak{A}$. Hence, some unfolding of $\texttt{dll}$ is accepted by $\mathfrak{A}$ and thus contains points-to assertions.

We now revisit question (2) from above–are all unfoldings accepted by a heap automaton?–and observe that heap automata enjoy several closure properties.

Theorem 2 ([29]). *Let $\mathfrak{A}$ and $\mathfrak{B}$ be heap automata over $\mathrm{SH}_{\mathcal{C}}$. Then there exist heap automata $\mathfrak{C}_1, \mathfrak{C}_2, \mathfrak{C}_3$ over $\mathrm{SH}_{\mathcal{C}}$ with $L(\mathfrak{C}_1) = L(\mathfrak{A}) \cup L(\mathfrak{B})$, $L(\mathfrak{C}_2) = L(\mathfrak{A}) \cap L(\mathfrak{B})$, and $L(\mathfrak{C}_3) = \mathrm{RSH}_{\mathcal{C}} \setminus L(\mathfrak{A})$, respectively.*

Then, by the equivalence $X \subseteq Y \Leftrightarrow X \cap \overline{Y} = \emptyset$ and Theorem 2, it is also decidable whether every unfolding of a symbolic heap is accepted by a heap automaton.

Corollary 2. *Let $\mathfrak{A}$ be a heap automaton over $\mathrm{SH}_{\mathcal{C}}$ and $\Phi \in \mathrm{SID}_{\mathcal{C}}$. Then, for each $\varphi \in \mathrm{SH}_{\mathcal{C}}$, it is decidable whether $\mathbb{U}_{\Phi}(\varphi) \subseteq L(\mathfrak{A})$ holds.*

Note that complementation of heap automata in general leads to an exponentially larger state space and exponentially higher complexity of evaluating Δ. Thus, $\mathbb{U}_{\Phi}(\varphi) \subseteq L(\mathfrak{A})$ is decidable in $\mathcal{O}\left((\|\varphi\| + \|\Phi\|) \cdot \|2^{Q_{\mathfrak{A}}}\|^{2(M+1)} \cdot \|\Delta_{\mathfrak{A}}\|\right)$. In many cases it is, however, possibly to construct smaller automata for the complement directly to obtain more efficient decision procedures. For example, this is the case for most heap automata considered in Section 4.

Apart from decision procedures, Theorem 1 enables systematic refinement of SIDs according to heap automata in order to establish desired properties. For instance, as shown in Section 4, an SID in which every unfolding is satisfiable can be constructed from any given SID. Another application of Theorem 1 is counterexample generation for systematic debugging of SIDs that are manually written as data structure specifications or even automatically generated. Such counterexamples are obtained by constructing the refined SID w.r.t. the complement of a given heap automaton. Then an unfolding of the SID that is rejected by the original heap automaton, i.e., a counterexample, can be reconstructed from a (failed) emptiness check. Further applications are examined in the following.

Remark 1. While we focus on the well-established symbolic heap fragment of separation logic, we remark that the general reasoning principle underlying heap automata is also applicable to check robustness properties of richer fragments. For example, permissions [7] are easily integrated within our framework.

4 A Zoo of Robustness Properties

This section demonstrates the wide applicability of heap automata to decide and establish robustness properties of SIDs. In particular, the sets of symbolic heaps informally presented in the introduction can be accepted by heap automata over the set $\mathrm{SH}_{\mathrm{FV}\leq\alpha}$ of symbolic heaps with at most $\alpha \geq 0$ free variables (cf. Example 2). Furthermore, we analyze the complexity of related decision problems. Towards a formal presentation, some terminology is needed.

Definition 7. *The set of* tight models *of a symbolic heap* $\varphi \in \mathrm{SH}^{\Phi}$ *is defined as* $\mathit{Models}(\varphi) \triangleq \{(s,h) \in \mathit{States} \,|\, \mathrm{dom}(s) = \mathbf{x}_0^{\varphi},\ s,h \models_{\Phi} \varphi\}$.

We often consider relationships between variables that hold in every tight model of a reduced symbolic heap. Formally, let $\tau \triangleq \exists \mathbf{z}.\Sigma : \Pi \in \mathrm{RSH}$. Moreover, let $\mathit{strip}(\tau)$ be defined as τ except that each of its variables is free, i.e., $\mathit{strip}(\tau) \triangleq \Sigma : \Pi$. Then two variables $x, y \in \mathit{Var}(\tau)$ are *definitely (un)equal* in τ, written $x =_{\tau} y$ ($x \neq_{\tau} y$), if $s(x) = s(y)$ ($s(x) \neq s(y)$) holds for every $(s,h) \in \mathit{Models}(\mathit{strip}(\tau))$. Analogously, a variable is *definitely allocated* if it is definitely equal to a variable occurring on the left-hand side of a points-to assertion. Thus the set of definitely allocated variables in τ is given by

$$\mathit{alloc}(\tau) \;=\; \{x \in \mathit{Var}(\tau) \mid \forall (s,h) \in \mathit{Models}(\mathit{strip}(\tau))\ .\ s(x) \in \mathrm{dom}(h)\}.$$

Finally, a variable x *definitely points-to* variable y in τ, written $x \mapsto_{\tau} y$, if for every $(s,h) \in \mathit{Models}(\mathit{strip}(\tau))$, we have $s(y) \in h(s(x))$.

Example 7. Recall from Example 1 the symbolic heap τ in the first rule of $\texttt{tll}\,\mathbf{x}_0$. Then $\mathit{alloc}(\tau) = \{x_1, x_2\}$ and neither $x_1 =_{\tau} x_3$ nor $x_1 \neq_{\tau} x_3$ holds. Further,

$x_1 =_{\tau} x_2$ is true,	$x_1 =_{\tau} x_3$ is false,	$x_1 \neq_{\tau} \mathbf{null}$ is true,
$x_1 \neq_{\tau} x_3$ is false,	$x_1 \mapsto_{\tau} x_3$ is true,	$x_3 \mapsto_{\tau} x_1$ is false.

Remark 2. All definite relationships are decidable in polynomial time. In fact, each of these relationships boils down to first adding inequalities $x \neq \mathbf{null}$ and $x \neq y$ for every pair x, y of distinct variables occurring on the left-hand side of points-to assertions to the set of pure formulas and then computing its (reflexive), symmetric (and transitive) closure with respect to $\neq$ (and $=$). Furthermore, if the closure contains a contradiction, e.g., $\mathbf{null} \neq \mathbf{null}$, it is set to all pure formulas over the variables of a given reduced symbolic heap. After that, it is straightforward to decide in polynomial time whether variables are definitely allocated, (un)equal or pointing to each other.

4.1 Tracking Equalities and Allocation

Consider the symbolic heap $\varphi \triangleq \exists x,y,z.P_1(x,y) * P_2(y,z) : \{x = z\}$. Clearly, φ is unsatisfiable if $x = y$ holds for every unfolding of $P_1(x,y)$ and $y \neq z$ holds for every unfolding of $P_2(y,z)$. Analogously, φ is unsatisfiable if x is allocated in every unfolding of $P_1(x,y)$ and z is allocated in every unfolding of $P_2(y,z)$, because $x \mapsto \_ * z \mapsto \_$ implies $x \neq z$. This illustrates that robustness properties, such as satisfiability, require detailed knowledge about the relationships between parameters of predicate calls. Consequently, we construct a heap automaton $\mathfrak{A}_{\texttt{TRACK}}$ that keeps track of this knowledge. More precisely, $\mathfrak{A}_{\texttt{TRACK}}$ should accept those unfoldings in which it is guaranteed that

- given a set $A \subseteq \mathbf{x}_0$, exactly the variables in A are definitely allocated, and
- exactly the (in)equalities in a given set of pure formulas Π hold.

Towards a formal construction, we formalize the desired set of symbolic heaps.

Definition 8. *Let $\alpha \in \mathbb{N}_{>0}$ and $\mathbf{x}_0$ be a tuple of variables with $\|\mathbf{x}_0\| = \alpha$. Moreover, let $A \subseteq \mathbf{x}_0$ and Π be a finite set of pure formulas over $\mathbf{x}_0$. The* tracking property $\texttt{TRACK}(\alpha, A, \Pi)$ *is the set*

$$\{\tau(\mathbf{x}_0) \in \mathrm{RSH}_{FV^{\leq \alpha}} \mid \forall i, j \;.\; \mathbf{x}_0[i] \in A \textit{ iff } \mathbf{x}_0[i] \in \mathit{alloc}(\tau) \\ \textit{and } \mathbf{x}_0[i] \sim \mathbf{x}_0[j] \in \Pi \;\; \textit{iff} \;\; \mathbf{x}_0^{\tau}[i] \sim_{\tau} \mathbf{x}_0^{\tau}[j]\}.$$

Intuitively, our heap automaton $\mathfrak{A}_{\texttt{TRACK}}$ stores in its state space which free variables are definitely equal, unequal and allocated. Its transition relation then enforces that these stored information are correct, i.e., a transition $\mathbf{q} \xrightarrow{\varphi}_{\mathfrak{A}_{\texttt{TRACK}}} p$ is only possible if the information stored in p is consistent with φ and with the information stored in the states $\mathbf{q}$ for the predicate calls of φ.

Formally, let $\mathbf{x}_0$ be a tuple of variables with $\|\mathbf{x}_0\| = \alpha$ and $\mathrm{Pure}(\mathbf{x}_0) \triangleq 2^{\{\mathbf{x}_0[i] \sim \mathbf{x}_0[j] \;\mid\; 0 \leq i,j \leq \alpha, \sim \in \{=, \neq\}\}}$ be the powerset of all pure formulas over $\mathbf{x}_0$. The information stored by our automaton consists of a set of free variables $B \subseteq \mathbf{x}_0$ and a set of pure formulas $\Lambda \in \mathrm{Pure}(\mathbf{x}_0)$. Now, for some unfolding τ of a symbolic heap φ, assume that B is chosen as the set of all definitely allocated free variables of τ. Moreover, assume Λ is the set of all definite (in)equalities between free variables in τ. We can then construct a reduced symbolic heap $\mathit{kernel}(\varphi, (B, \Lambda))$ from B and Λ that precisely captures these relationships between free variables.

Definition 9. *Let $\varphi\mathbf{x}_0$ be a symbolic heap, $B \subseteq \mathbf{x}_0$ and $\Lambda \in \mathrm{Pure}(\mathbf{x}_0)$. Furthermore, let $\mathit{min}(B, \Lambda) = \{\mathbf{x}_0^i \in B \mid \neg\exists \mathbf{x}_0^j \in B . j < i \text{ and } \mathbf{x}_0^i =_{\Lambda} \mathbf{x}_0^j\}$ be the set of minimal (w.r.t. to occurrence in $\mathbf{x}_0$) allocated free variables. Then*

$$\mathit{kernel}(\varphi, (B, \Lambda)) \;\triangleq\; \bigstar_{\mathbf{x}_0[i] \in \mathit{min}(B,\Lambda)} \; \mathbf{x}_0^{\varphi}[i] \mapsto \mathbf{null} \;:\; \Lambda,$$

where we write $\bigstar_{s \in S}\, s \mapsto \mathbf{null}$ for $s_1 \mapsto \mathbf{null} * \ldots * s_k \mapsto \mathbf{null}$, $S = \{s_1, \ldots, s_k\}$.

Consequently, the relationships between free variables remain unaffected if a predicate call of φ is replaced by $\mathit{kernel}(\varphi, (B, \Lambda))$ instead of τ. Thus, $\mathfrak{A}_{\texttt{TRACK}}$ has one state per pair (B, Λ). In the transition relation of $\mathfrak{A}_{\texttt{TRACK}}$ it suffices to replace each predicate call $P\mathbf{x}_0$ by the corresponding symbolic heap $\mathit{kernel}(P\mathbf{x}_0, (B, \Lambda))$. and check whether the current state is consistent with the resulting symbolic heap. Intuitively, a potentially large unfolding of a symbolic heap φ with m predicate calls is "compressed" into a small one that contains all necessary information about parameters of predicate calls. Here, $\mathbf{q}$ is a sequence of pairs (B, Λ) as explained above. Formally,

Definition 10. $\mathfrak{A}_{\mathtt{TRACK}} = (Q, \mathrm{SH}_{FV^{\leq \alpha}}, \Delta, F)$ *is given by:*

$$\begin{aligned}
Q &\triangleq 2^{\mathbf{x}_0} \times \mathrm{Pure}(\mathbf{x}_0), \qquad F \triangleq \{(A, \Pi)\}, \\
\Delta &: \mathbf{q} \xrightarrow{\varphi}_{\mathfrak{A}_{\mathtt{TRACK}}} (A_0, \Pi_0) \text{ iff } \forall x, y \in \mathbf{x}_0 \,. \\
&\qquad y \in A_0 \leftrightarrow y^{\varphi} \in alloc(compress(\varphi, \mathbf{q})) \\
&\qquad \text{and } x \sim y \in \Pi_0 \;\leftrightarrow\; x^{\varphi} \sim_{compress(\varphi,\mathbf{q})} y^{\varphi} \;, \\
compress(\varphi, \mathbf{q}) &\triangleq \varphi\left[P_1/kernel(P_1\mathbf{x}_1, \mathbf{q}[1]), \ldots, P_m/kernel(P_m\mathbf{x}_m, \mathbf{q}[m])\right] \;,
\end{aligned}$$

where $m = \|\Gamma^{\varphi}\| = \|\mathbf{q}\|$ is the number of predicate calls in φ and y^{φ} denotes the free variable of φ corresponding to $y \in \mathbf{x}_0$, i.e., if $y = \mathbf{x}_0[i]$ then $y^{\varphi} = \mathbf{x}_0^{\varphi}[i]$.

Since $compress(\tau, \varepsilon) = \tau$ holds for every reduced symbolic heap τ, it is straightforward to show that $L(\mathfrak{A}_{\mathtt{TRACK}}) = \mathtt{TRACK}(\alpha, A, \Pi)$. Furthermore, $\mathfrak{A}_{\mathtt{TRACK}}$ satisfies the compositionality property [29]. Hence,

Lemma 3. *For all $\alpha \in \mathbb{N}_{>0}$ and all sets $A \subseteq \mathbf{x}_0$, $\Pi \in \mathrm{Pure}(\mathbf{x}_0)$, there is a heap automaton over $\mathrm{SH}_{FV^{\leq \alpha}}$ accepting $\mathtt{TRACK}(\alpha, A, \Pi)$.*

4.2 Satisfiability

Tracking relationships between free variables of symbolic heaps is a useful auxiliary construction that serves as a building block in automata for more natural properties. For instance, the heap automaton $\mathfrak{A}_{\mathtt{TRACK}}$ constructed in Definition 10 can be reused to deal with the
Satisfiability problem (SL-SAT): Given $\Phi \in \mathrm{SID}$, $\varphi \in \mathrm{SH}^{\Phi}$, decide whether φ is satisfiable, i.e., there exists $(s, h) \in \mathit{States}$ such that $s, h \models_{\Phi} \varphi$.

Theorem 3. *For each $\alpha \in \mathbb{N}_{>0}$, there is a heap automaton over $\mathrm{SH}_{FV^{\leq \alpha}}$ accepting the set $\mathtt{SAT}(\alpha) \triangleq \{\tau \in \mathrm{RSH}_{FV^{\leq \alpha}} \mid \tau \text{ is satisfiable}\}$ of all satisfiable reduced symbolic heaps with at most α free variables.*

Proof. A heap automaton accepting $\mathtt{SAT}(\alpha)$ is constructed as in Definition 10 except for the set of final states $F \triangleq \{(A, \Pi) \mid \mathbf{null} \neq \mathbf{null} \notin \Pi\}$ (cf. [29]). $\square$

A heap automaton accepting the complement of $\mathtt{SAT}(\alpha)$ is constructed analogously by choosing $F \triangleq \{(A, \Pi) \mid \mathbf{null} \neq \mathbf{null} \in \Pi\}$. Thus, together with Corollary 1, we obtain a decision procedure for the satisfiability problem similar to the one proposed in [11]. Regarding complexity, the heap automaton $\mathfrak{A}_{\mathtt{SAT}}$ from Definition 10 has $2^{2\alpha^2+\alpha}$ states. By Remark 2, membership in $\Delta_{\mathfrak{A}_{\mathtt{SAT}}}$ is decidable in polynomial time. Thus, by Corollary 1, our construction yields an exponential-time decision procedure for SL-SAT. If the number of free variables α is bounded, an algorithm in NP is easily obtained by guessing a suitable unfolding tree of height at most $\|Q_{\mathfrak{A}_{\mathtt{SAT}}}\|$ and running $\mathfrak{A}_{\mathtt{SAT}}$ on it to check whether its unfolding is decidable (cf. [29]). This is in line with the results of Brotherston et al. [11], where the satisfiability problem is shown to be ExpTime–complete in general and NP–complete if the number of free variables is bounded. These complexity bounds even hold for the following special case [13]:

Restricted satisfiability problem (SL-RSAT) Given an SID Φ that contains no points-to assertions, and a predicate symbol P, decide whether $P\mathbf{x}$ is satisfiable w.r.t. Φ. The complement of this problem is denoted by $\overline{\text{SL-RSAT}}$.

4.3 Establishment

A symbolic heap φ is *established* if every existentially quantified variable of every unfolding of φ is definitely equal to a free variable or definitely allocated.[5] This property is natural for symbolic heaps that specify the shape of data structures; for example, the SIDs in Example 1 define sets of established symbolic heaps. Further, establishment is often required to ensure decidability of the entailment problem [26,27]. Establishment can also be checked by heap automata.

Theorem 4. *For all $\alpha \in \mathbb{N}_{>0}$, there is a heap automaton over $\text{SH}_{FV \leq \alpha}$ accepting the set of all established reduced symbolic heaps with at most α free variables:*

$$\texttt{EST}(\alpha) \triangleq \{\tau \in \text{RSH}_{FV \leq \alpha} \mid \forall y \in \mathit{Var}(\tau) \;.\; y \in \mathit{alloc}(\tau) \text{ or } \exists x \in \mathbf{x}_0^{\tau} \;.\; x =_{\tau} y\}$$

Proof. The main idea in the construction of a heap automaton $\mathfrak{A}_{\texttt{EST}}$ for $\texttt{EST}(\alpha)$ is to verify that every variable is definitely allocated or equal to a free variable while running $\mathfrak{A}_{\texttt{TRACK}}$ (see Definition 10) in parallel to keep track of the relationships between free variables. An additional flag $q \in \{0,1\}$ is attached to each state of $\mathfrak{A}_{\texttt{TRACK}}$ to store whether the establishment condition is already violated ($q = 0$) or holds so far ($q = 1$). Formally, $\mathfrak{A}_{\texttt{EST}} = (Q, \text{SH}_{\text{FV} \leq \alpha}, \Delta, F)$, where

$$\begin{aligned}
Q &\triangleq Q_{\mathfrak{A}_{\texttt{TRACK}}} \times \{0,1\}, \qquad F \triangleq Q_{\mathfrak{A}_{\texttt{TRACK}}} \times \{1\}, \\
\Delta &: (p_1, q_1) \dots (p_m, q_m) \xrightarrow{\varphi}_{\mathfrak{A}_{\texttt{EST}}} (p_0, q_0) \\
&\text{iff } p_1 \dots p_m \xrightarrow{\varphi}_{\mathfrak{A}_{\texttt{TRACK}}} p_0 \text{ and } q_0 = \min\{q_1, \dots, q_m, \mathit{check}(\varphi, p_1 \dots p_m)\}.
\end{aligned}$$

Here, $\mathit{check} : \text{SH}_{\text{FV} \leq \alpha} \times Q^{*}_{\mathfrak{A}_{\texttt{TRACK}}} \to \{0,1\}$ is a predicate given by

$$\mathit{check}(\varphi, \mathbf{p}) \triangleq \begin{cases} 1 & , \text{ if } \forall y \in \mathit{Var}(\varphi) \;.\; y \in \mathit{alloc}(\mathit{compress}(\varphi, \mathbf{p})) \\ & \quad \text{or } \exists x \in \mathbf{x}_0^{\varphi} \;.\; x =_{\mathit{compress}(\varphi,\mathbf{p})} y \\ 0 & , \text{ otherwise }, \end{cases}$$

where $\mathit{compress}(\varphi, \mathbf{p})$ is the reduced symbolic heap obtained from the tracking property as in Definition 10. Moreover, unlike in the construction of $\mathfrak{A}_{\texttt{TRACK}}$, we are not interested in a specific set of relationships between the pure formulas, so any state of $\mathfrak{A}_{\texttt{TRACK}}$ is chosen as a final state provided that predicate *check* could be evaluated to 1. See [29] for a correctness proof. □

Again, it suffices to swap the final- and non-final states of $\mathfrak{A}_{\texttt{EST}}$ to obtain a heap automaton $\mathfrak{A}_{\overline{\texttt{EST}}}$ accepting the complement of $\texttt{EST}(\alpha)$. Thus, by Corollary 1 and Remark 2, we obtain an ExpTime decision procedure for the
Establishment problem (SL-EST): Given an SID Φ and $\varphi \in \text{SH}^{\Phi}$, decide whether every $\tau \in \mathbb{U}_{\Phi}(\varphi)$ is established.

[5] Sometimes this property is also defined by requiring that each existentially quantified variable is "eventually allocated" [26].

Lemma 4. $\overline{\text{SL-RSAT}}$ *is polynomial-time reducible to* SL-EST. *Hence, the establishment problem* SL-EST *is* ExpTime*–hard in general and* coNP*–hard if the maximal number of free variables is bounded.*

Proof. Let (Φ, P) be an instance of $\overline{\text{SL-RSAT}}$. Moreover, let $\varphi\mathbf{x}_0 \triangleq \exists \mathbf{z} y \,.\, P\mathbf{z} : \{\mathbf{x}_0[1] = \mathbf{null}, y \neq \mathbf{null}\}$. As y is neither allocated nor occurs in $P\mathbf{z}$, φ is established iff $\mathbf{x}_0[1] = y$ iff $\mathbf{null} \neq \mathbf{null}$ iff $P\mathbf{x}$ is unsatisfiable. Hence, $(\Phi, \varphi) \in$ SL-EST iff $(\Phi, P) \in \overline{\text{SL-RSAT}}$. A full proof is found in [29]. □

Lemma 5. SL-EST *is in* coNP *for a bounded number of free variables* α.

Proof. Let (Φ, φ) be an instance of SL-EST, $N = \|\Phi\| + \|\varphi\|$, and $M \leq N$ be the maximal number of predicate calls occurring in φ and any rule of Φ. Moreover, let $\mathfrak{A}_{\overline{\mathtt{EST}}}$ be a heap automaton accepting $\overline{\mathtt{EST}(\alpha)}$—the complement of $\mathtt{EST}(\alpha)$ (cf. Theorem 4). Since α is bounded by a constant, so is the number of states of $\mathfrak{A}_{\overline{\mathtt{EST}}}$, namely $\|Q_{\mathfrak{A}_{\overline{\mathtt{EST}}}}\| \leq k = 2^{2\alpha^2+\alpha+1}$. Now, let $\mathbb{T}_\Phi(\varphi)^{\leq k}$ denote the set of all unfolding trees $t \in \mathbb{T}_\Phi(\varphi)$ of height at most k. Clearly, each of these trees is of size $\|t\| \leq M^k \leq N^k$, i.e., polynomial in N. Moreover, let $\omega : \text{dom}(t) \to Q_{\mathfrak{A}_{\overline{\mathtt{EST}}}}$ be a function mapping each node of t to a state of $\mathfrak{A}_{\overline{\mathtt{EST}}}$. Again, ω is of size polynomial in N; as such $\|\omega\| \leq k \cdot N^k$. Let Ω_t denote the set of all of these functions ω for a given unfolding tree t with $\omega(\varepsilon) \in F_{\mathfrak{A}_{\overline{\mathtt{EST}}}}$. Given an unfolding tree $t \in \mathbb{T}_\Phi(\varphi)^{\leq k}$ and $\omega \in \Omega_t$, we can easily decide whether $\varepsilon \xrightarrow{[\![t]\!]}_{\mathfrak{A}_{\overline{\mathtt{EST}}}} \omega(\varepsilon)$ holds: For each $u, u1, \ldots, un \in \text{dom}(t)$, $u(n+1) \notin \text{dom}(t)$, $n \geq 0$, it suffices to check whether $\omega(u1) \ldots \omega(un) \xrightarrow{t(u)}_{\mathfrak{A}_{\overline{\mathtt{EST}}}} \omega(u)$. Since, by Remark 2, each of these checks can be performed in time polynomial in N the whole procedure is feasible in polynomial time. We now show that $(\Phi, \varphi) \in$ SL-EST if and only if

$$\forall t \in \mathbb{T}_\Phi(\varphi)^{\leq k} \,.\, \forall \omega \in \Omega_t \,.\, \text{not } \varepsilon \xrightarrow{[\![t]\!]}_{\mathfrak{A}_{\overline{\mathtt{EST}}}} \omega(\varepsilon).$$

Since each $t \in \mathbb{T}_\Phi(\varphi)$ and each $\omega \in \Omega_t$ is of size polynomial in N, this is equivalent to SL-EST being in coNP. To complete the proof, note that $\mathbb{U}_\Phi(\varphi) \subseteq \mathtt{EST}(\alpha)$ holds iff $[\![t]\!] \notin \overline{\mathtt{EST}(\alpha)}$ for each $t \in \mathbb{T}_\Phi(\varphi)$. Furthermore, by a standard pumping argument, it suffices to consider trees in $\mathbb{T}_\Phi(\varphi)^{\leq k}$: If there exists a taller tree t with $[\![t]\!] \in \overline{\mathtt{EST}(\alpha)}$ then there is some path of length greater k in t on which two nodes are assigned the same state by a function $\omega \in \Omega_t$ proving membership of t in $\overline{\mathtt{EST}(\alpha)}$. This path can be shortened to obtain a tree of smaller height. □

Putting upper and lower bounds together, we conclude:

Theorem 5. SL-EST *is* ExpTime*–complete in general and* coNP*–complete if the number of free variables* α *is bounded.*

4.4 Reachability

Another family of robustness properties is based on reachability questions, e.g., "is every location of every model of a symbolic heap reachable from the location of a program variable?" or "is every model of a symbolic heap acyclic?". For established SIDs, heap automata accepting these properties are an extension of the tracking automaton introduced in Definition 10.

More precisely, a variable y is *definitely reachable* from x in $\tau \in \text{RSH}$, written $x \rightsquigarrow_\tau y$, if and only if $x \mapsto_\tau y$ or there exists a $z \in \mathit{Var}(\tau)$ such that $x \mapsto_\tau z$ and $z \rightsquigarrow_\tau y$.[6] Note that we define reachability to be transitive, but not reflexive. As for the other definite relationships between variables, definite reachability is computable in polynomial time for reduced symbolic heaps, e.g., by performing a depth-first search on the definite points-to relation $\mapsto_\tau$. Note that our notion of reachability does not take variables into account that are only reachable from one another in *some* models of a reduced symbolic heap. For example, consider the symbolic heap $\tau = x \mapsto y * z \mapsto \mathbf{null}$. Then $x \rightsquigarrow_\tau z$ does *not* hold, but there exists a model (s, h) with $s(z) = s(y) \in h(s(x))$. Thus, reachability introduced by unallocated variables is not detected. However, the existence (or absence) of such variables can be checked first due to Theorem 4.

Theorem 6. *Let $\alpha \in \mathbb{N}_{>0}$ and $R \subseteq \mathbf{x}_0 \times \mathbf{x}_0$ be a binary relation over the variables $\mathbf{x}_0$ with $\|\mathbf{x}_0\| = \alpha$. Then the* reachability property $\mathtt{REACH}(\alpha, R)$*, given by the set $\{\tau \in \text{RSH}_{FV \leq \alpha} \mid \forall i, j \ . \ (\mathbf{x}_0[i], \mathbf{x}_0[j]) \in R \text{ iff } \mathbf{x}_0^\tau[i] \rightsquigarrow_\tau \mathbf{x}_0^\tau[j]\}$, can be accepted by a heap automaton over $\text{SH}_{FV \leq \alpha}$.*

Proof (sketch). A heap automaton $\mathfrak{A}_{\mathtt{REACH}}$ accepting $\mathtt{REACH}(\alpha, R)$ is constructed similarly to the heap automaton $\mathfrak{A}_{\mathtt{TRACK}}$ introduced in Definition 10. The main difference is that $\mathfrak{A}_{\mathtt{REACH}}$ additionally stores a binary relation $S \subseteq \mathbf{x}_0 \times \mathbf{x}_0$ in its state space to remember which free variables are reachable from one another. Correspondingly, we adapt Definition 9 as follows:

$$\mathit{kernel}(\varphi, (B, \Lambda, S)) \triangleq \exists z \ . \bigstar_{\min(B,\Lambda)} \mathbf{x}_0^\varphi[i] \mapsto (\mathbf{v}_i) \ : \ \Lambda \ ,$$

where z is a fresh variable and $\mathbf{v}_i[j] \triangleq \mathbf{x}_0^\varphi[j]$ if $(i, j) \in S$ and $\mathbf{v}_i[j] \triangleq z$, otherwise. The other parameters φ, B, Λ are the same as in Definition 10. Note that the additional variable z is needed to deal with allocated free variables that cannot reach any other free variable, including **null**. Moreover, the set of final states is $F_{\mathfrak{A}_{\mathtt{REACH}}} = Q_{\mathfrak{A}_{\mathtt{TRACK}}} \times \{R\}$. Correctness of this encoding is verified in the transition relation. Hence, the transition relation of $\mathfrak{A}_{\mathtt{REACH}}$ extends the transition relation of $\mathfrak{A}_{\mathtt{TRACK}}$ by the requirement $(x, y) \in S$ iff $x^\varphi \rightsquigarrow_{\mathit{compress}(\varphi, \mathbf{p})} y^\varphi$ for every pair of free variables $x, y \in \mathbf{x}_0$. Here, $\mathit{compress}(\varphi, \mathbf{p})$ is defined as in Definition 10 except that the new encoding $\mathit{kernel}(P_i\mathbf{x}_i, \mathbf{q}[i])$ from above is used. Since $\mathit{compress}(\tau, \varepsilon) = \tau$ holds for every reduced symbolic heap τ, it is straightforward to verify that $L(\mathfrak{A}_{\mathtt{REACH}}) = \mathtt{REACH}(\alpha)$. Further details are found in [29]. □

Furthermore, we consider the related
Reachability problem (SL-REACH): Given an SID Φ, $\varphi \in \text{SH}^\Phi$ with $\alpha = \|\mathbf{x}_0^\varphi\|$ and variables $x, y \in \mathbf{x}_0^\varphi$, decide whether $x \rightsquigarrow_\tau y$ holds for all $\tau \in \mathbb{U}_\Phi(\varphi)$.

Theorem 7. *The decision problem* SL-REACH *is* ExpTime*–complete in general and* coNP*–complete if the number of free variables is bounded.*

[6] The definite points-to relation $\mapsto_\tau$ was defined at the beginning of Section 4.

Proof. Membership in ExpTime follows from our upper bound derived for Algorithm 1, the size of the state space of $\mathfrak{A}_{\texttt{REACH}}$, which is exponential in α, and Remark 2. If α is bounded, membership in coNP is shown analogously to Lemma 5. Lower bounds are shown by reducing $\overline{\text{SL-RSAT}}$ to SL-REACH. Formally, let (Φ, P) be an instance of $\overline{\text{SL-RSAT}}$. Moreover, let $\varphi\mathbf{x}_0 \triangleq \exists\mathbf{z} \;.\; \mathbf{x}_0[1] \mapsto \mathbf{null} * P\mathbf{z} \;:\; \{\mathbf{x}_0[2] \neq \mathbf{null}\}$. As $\mathbf{x}_0[2]$ is neither allocated nor **null**, $\mathbf{x}_0[2]$ is *not* definitely reachable from $\mathbf{x}_0[1]$ in *any* model of φ. Hence $(\Phi, \varphi, \mathbf{x}_0[1], \mathbf{x}_0[2]) \in$ SL-REACH iff P is unsatisfiable. A detailed proof is found in [29]. □

4.5 Garbage-Freedom

Like the tracking automaton $\mathfrak{A}_{\texttt{TRACK}}$, the automaton $\mathfrak{A}_{\texttt{REACH}}$ is a useful ingredient in the construction of more complex heap automata.

For instance, such an automaton can easily be modified to check whether a symbolic heap is *garbage-free*, i.e., whether every existentially quantified variable in every unfolding is reachable from some program variable.[7]

Garbage-freedom is a natural requirement if SIDs represent data structure specifications. For example, the SIDs in Example 1 are garbage-free. Furthermore, this property is needed by the approach of Habermehl et al. [24].

Lemma 6. *For each $\alpha \in \mathbb{N}_{>0}$, the set* $\texttt{GFREE}(\alpha)$*, given by*

$$\{\tau \in \text{RSH}_{FV\leq\alpha} \mid \forall y \in \mathit{Var}(\tau) \;.\; \exists x \in \mathbf{x}_0^{\tau} \;.\; x =_{\tau} y \text{ or } x \rightsquigarrow_{\tau} y\},$$

of garbage-free symbolic heaps can be accepted by a heap automaton over $\text{SH}_{FV\leq\alpha}$.

Proof (sketch). A heap automaton $\mathfrak{A}_{\texttt{GFREE}}$ accepting $\texttt{GFREE}(\alpha)$ is constructed similarly to the heap automaton $\mathfrak{A}_{\texttt{EST}}$ introduced in the proof of Theorem 4. The main difference is that heap automaton $\mathfrak{A}_{\texttt{REACH}}$ is used instead of $\mathfrak{A}_{\texttt{TRACK}}$. Furthermore, the predicate $\mathit{check} : \text{SH}_{\text{FV}\leq\alpha} \times Q^{*}_{\mathfrak{A}_{\texttt{REACH}}} \to \{0,1\}$ is redefined to verify that every variable of a symbolic heap φ is established in $\mathit{compress}(\varphi, \mathbf{p})$, where $\mathit{compress}(\varphi, \mathbf{p})$ is the same as in the construction of $\mathfrak{A}_{\texttt{REACH}}$ (see Theorem 6):

$$\mathit{check}(\varphi, \mathbf{p}) \;\triangleq\; \begin{cases} 1 & , \text{ if } \forall y \in \mathit{Var}(\varphi) \,.\, \exists x \in \mathbf{x}_0^{\varphi} \;. \\ & \qquad x =_{\mathit{compress}(\varphi,\mathbf{p})} y \text{ or } x \rightsquigarrow_{\mathit{compress}(\varphi,\mathbf{p})} y \\ 0 & , \text{ otherwise }, \end{cases}$$

Since $\mathit{compress}(\tau, \varepsilon) = \tau$ holds for every reduced symbolic heap τ, it is straightforward that $L(\mathfrak{A}_{\texttt{GFREE}}) = \texttt{GFREE}(\alpha)$. A proof is found in [29]. □

To guarantee that symbolic heaps are garbage-free, we solve the
Garbage-freedom problem (SL-GF): Given an SID Φ and $\varphi \in \text{SH}^{\Phi}$, decide whether every $\tau \in \mathbb{U}_{\Phi}(\varphi)$ is garbage-free, i.e., $\tau \in \texttt{GFREE}(\alpha)$ for some $\alpha \in \mathbb{N}$.

Theorem 8. SL-GF *is* ExpTime*–complete in general and* coNP*–complete if the number of free variables α is bounded.*

[7] Note that a variable may be reachable from different program variables in different unfoldings as garbage-freedom is formally defined as a set of *reduced* symbolic heaps in which no form of disjunction exists (cf. Lemma 6).

4.6 Acyclicity

Automatic termination proofs of programs frequently rely on the acyclicity of employed data structures, i.e., they assume that no variable is reachable from itself (cf. [39]). Hence, we are interested in verifying that an SID is acyclic.

Lemma 7. *For each $\alpha \in \mathbb{N}_{>0}$, the set of all weakly acyclic symbolic heaps*

$$\texttt{ACYCLIC}(\alpha) \triangleq \{\tau \in \mathrm{RSH}_{FV^{\leq \alpha}} \mid \mathbf{null} \neq_\tau \mathbf{null} \text{ or } \forall x \in \mathit{Var}(\tau) \;.\; \text{not } x \rightsquigarrow_\tau x\}$$

can be accepted by a heap automaton over $\mathrm{SH}_{FV^{\leq \alpha}}$.

Here, the condition $\mathbf{null} \neq_\tau \mathbf{null}$ ensures that an unsatisfiable reduced symbolic heap is considered weakly acyclic. Further, note that our notion of acyclicity is weak in the sense that dangling pointers may introduce cyclic models that are not considered. For example, $\exists z.x \mapsto z$ is weakly acyclic, but contains cyclic models if x and z are aliases. However, weak acyclicity coincides with the absence of cyclic models for established SIDs—a property considered in Section 4.3.

Proof (sketch). A heap automaton $\mathfrak{A}_{\texttt{ACYCLIC}}$ for the set of all weakly acyclic reduced symbolic heaps is constructed analogously to the heap automaton $\mathfrak{A}_{\texttt{GFREE}}$ in the proof of Lemma 6. The main difference is the predicate $\mathit{check} : \mathrm{SH}_{\mathrm{FV}^{\leq \alpha}} \times Q^*_{\mathfrak{A}_{\texttt{REACH}}} \rightarrow \{0,1\}$, which now checks whether a symbolic heap is weakly acyclic:

$$\mathit{check}(\varphi, \mathbf{p}) \triangleq \begin{cases} 1 & \text{, if } \forall y \in \mathit{Var}(\varphi) \text{ . not } x \rightsquigarrow_{\mathit{compress}(\varphi,\mathbf{p})} x \\ 0 & \text{, otherwise.} \end{cases}$$

Moreover, the set of final states $F_{\mathfrak{A}_{\texttt{ACYCLIC}}}$ is chosen such that accepted symbolic heaps are unsatisfiable or $\mathit{check}(\varphi, \mathbf{p}) = 1$. See [29] for details. □

For example, the symbolic heap $\texttt{sll}\,\mathbf{x}_0$ is weakly acyclic, but $\texttt{dll}\,\mathbf{x}_0$ (cf. Example 1) is not. In general, we are interested in the
Acyclicity problem (SL-AC): Given an SID Φ and $\varphi \in \mathrm{SH}^{\Phi}$, decide whether every $\tau \in \mathbb{U}_{\Phi}(\varphi)$ is weakly acyclic, i.e., $\tau \in \texttt{ACYCLIC}(\alpha)$ for some $\alpha \in \mathbb{N}$.

Theorem 9. SL-AC *is* ExpTime*–complete in general and* coNP*–complete if the number of free variables α is bounded.*

Proof. Similar to the proof of Theorem 5. For lower bounds, we show that $\overline{\text{SL-RSAT}}$ is reducible to SL-AC. Let (Φ, P) be an instance of $\overline{\text{SL-RSAT}}$. Moreover, let $\varphi\mathbf{x}_0 = \exists \mathbf{z}.\mathbf{x}_0[1] \mapsto \mathbf{x}_0[1] * P\mathbf{z}$. Since $\mathbf{x}_0[1]$ is definitely reachable from itself, $\varphi\mathbf{x}_0$ is weakly acyclic iff $P\mathbf{x}_0$ is unsatisfiable. Thus, $(\Phi, \varphi) \in$ SL-AC iff $(\Phi, P) \in \overline{\text{SL-RSAT}}$. See [29] for details. □

5 Implementation

We developed a prototype of our framework—called Harrsh[8]—that implements Algorithm 1 as well as all heap automata constructed in the previous sections.

[8] Heap Automata for Reasoning about Robustness of Symbolic Heaps

In addition, our tool supports automatic refinement of SIDs. Algorithms for counterexample generation and automatic combination of decision procedures can be extracted from the (constructive) proof of Theorem 2, but have not yet been implemented. The code, the tool and our experiments are available online.[9]

For our experimental results, we first considered common SIDs from the literature, such as singly- and doubly-linked lists, trees, trees with linked-leaves etc. For each of these SIDs, we checked all robustness properties presented throughout this paper, i.e., the existence of points-to assertions (Example 4), the tracking property $\texttt{TRACK}(B, \Lambda)$ (Section 4.1), satisfiability (Section 4.2), establishment (Section 4.3), the reachability property $\texttt{REACH}(\alpha, R)$ (Section 4.4), garbage-freedom (Section 4.5), and weak acyclicity (Section 4.6). All in all, our implementation of Algorithm 1 takes 300ms to successfully check these properties on all 45 problem instances. Since the SIDs under consideration are typically carefully handcrafted to be robust, the low runtime is to be expected. Moreover, we ran heap automata on benchmarks of the tool Cyclist [11]. In particular, our results for the satisfiability problem—the only robustness property checked by both tools—were within the same order of magnitude.

Further details are found in [29].

6 Entailment Checking with Heap Automata

So far, we have constructed heap automata for reasoning about robustness properties, such as satisfiability, establishment and acyclicity. This section demonstrates that our approach can also be applied to discharge *entailments* for certain fragments of separation logic. Formally, we are concerned with the
Entailment problem (SL-ENTAIL$_{\mathcal{C}}^{\Phi}$): Given symbolic heaps $\varphi, \psi \in \text{SH}_{\mathcal{C}}^{\Phi}$, decide whether $\varphi \models_{\Phi} \psi$ holds, i.e., $\forall (s,h) \in \mathit{States} \,.\, s,h \models_{\Phi} \varphi$ implies $s,h \models_{\Phi} \psi$.

Note that the symbolic heap fragment of separation logic is *not* closed under conjunction and negation. Thus, a decision procedure for satisfiability (cf. Theorem 3) does *not* yield a decision procedure for the entailment problem. It is, however, essential to have a decision procedure for entailment, because this problem underlies the important rule of consequence in Hoare logic [25]. In the words of Brotherston et al. [10], "effective procedures for establishing entailments are at the foundation of automatic verification based on separation logic".

We show how our approach to decide robustness properties, is applicable to discharge entailments for certain fragments of symbolic heaps. This results in an algorithm deciding entailments between so-called *determined symbolic heaps* for SIDs whose predicates can be characterized by heap automata.

Definition 11. *A reduced symbolic heap τ is* determined *if all tight models of τ are isomorphic.*[10] *If τ is also satisfiable then we call τ* well-determined.

[9] https://bitbucket.org/jkatelaan/harrsh/

[10] Two states $(s_1, h_1), (s_2, h_2)$ are *isomorphic* iff $\text{dom}(s_1) = \text{dom}(s_2)$ and there exists a bijective function $g : \text{dom}(h_1) \to \text{dom}(h_2)$ such that for all $x \in \text{dom}(s_1)$ and all $\ell \in \text{dom}(h_1)$, we have $g(s_1(x)) = s_2(x)$ and $g(h_1(\ell)) = h_2(g(\ell))$, where g is lifted to tuples by componentwise application.

Moreover, for some SID Φ, a symbolic heap $\varphi \in \mathrm{SH}^{\Phi}$ is (well-)determined if all of its unfoldings $\tau \in \mathbb{U}_{\Phi}(\varphi)$ are (well-)determined. Consequently, an SID Φ is (well-)determined if $P\mathbf{x}$ is (well-)determined for each predicate symbol P in Φ.

We present two sufficient conditions for determinedness of symbolic heaps. First, a reduced symbolic heap τ is determined if all equalities and inequalities between variables are explicit, i.e., $\forall x, y \in \mathit{Var}(\tau) \,.\, x = y \in \Pi^{\tau}$ or $x \neq y \in \Pi^{\tau}$ [29]. Furthermore, a reduced symbolic heap τ is determined if every variable is definitely allocated or definitely equal to **null**, i.e., $\forall x \in \mathit{Var}(\tau) \,.\, x \in \mathit{alloc}(\tau)$ or $x =_{\tau} \mathbf{null}$. These two notions can also be combined: A symbolic heap is determined if every variable x is definitely allocated or definitely equal to **null** or there is an explicit pure formula $x \sim y$ between x and each other variable y.

Example 8. By the previous remark, the SID generating acyclic singly-linked lists from Section 1 is well-determined. Furthermore, although the predicate $\mathtt{dll}\,\mathbf{x}_0$ from Example 1 is not determined, the following symbolic heap is well-determined: $\mathbf{x}_0[4] \mapsto \mathbf{null} * \mathtt{dll}\,\mathbf{x}_0 : \{\mathbf{x}_0[1] \neq \mathbf{x}_0[3]\}$.

6.1 Entailment between predicate calls

We start by considering entailments between predicate calls of well-determined SIDs. By definition, an entailment $\varphi \models_{\Phi} \psi$ holds if for every stack–heap pair (s, h) that satisfies an unfolding of φ, there exists an unfolding of ψ that is satisfied by (s, h) as well. Our first observation is that, for well-determined unfoldings, two quantifiers can be switched: It suffices for each unfolding σ of φ to find *one* unfolding τ of ψ such that every model of σ is also a model of τ.

Lemma 8. *Let $\Phi \in \mathrm{SID}$ and P_1, P_2 be predicate symbols with $\mathrm{ar}(P_1) = \mathrm{ar}(P_2)$. Moreover, let $\mathbb{U}_{\Phi}(P_1\mathbf{x})$ be well-determined. Then*

$$P_1\mathbf{x} \models_{\Phi} P_2\mathbf{x} \quad \textit{iff} \quad \forall \sigma \in \mathbb{U}_{\Phi}(P_1\mathbf{x}) \,.\, \exists \tau \in \mathbb{U}_{\Phi}(P_2\mathbf{x}) \,.\, \sigma \models_{\emptyset} \tau.$$

Note that, even if only well-determined predicate calls are taken into account, it is undecidable in general whether an entailment $P_1\mathbf{x}_0 \models_{\Phi} P_2\mathbf{x}_0$ holds [1, Theorem 3]. To obtain decidability, we additionally require the set of reduced symbolic heaps entailing a given predicate call to be accepted by a heap automaton.

Definition 12. *Let $\Phi \in \mathrm{SID}_{\mathcal{C}}$ and $\varphi \in \mathrm{SH}^{\Phi}_{\mathcal{C}}$. Then*

$$H^{\mathcal{C}}_{\varphi,\Phi} \triangleq \{\sigma \in \mathrm{RSH}_{\mathcal{C}} \mid \|\mathbf{x}_0^{\sigma}\| = \|\mathbf{x}_0^{\varphi}\| \textit{ and } \exists \tau \in \mathbb{U}_{\Phi}(\varphi) \,.\, \sigma \models_{\emptyset} \tau\}$$

is the set of all reduced symbolic heaps in $\mathrm{SH}_{\mathcal{C}}$ over the same free variables as φ that entail an unfolding of φ.

Example 9. Let $\varphi(\mathbf{x}_0) = \mathtt{tll}\,\mathbf{x}_0 : \{\mathbf{x}_0[1] \neq \mathbf{x}_0[2]\}$, where $\mathtt{tll}$ is a predicate of SID Φ introduced in Example 1. Then $H^{\mathrm{FV}^{\leq 3}}_{\varphi,\Phi}$ consists of all reduced symbolic heaps with three free variables representing non-empty trees with linked leaves. In particular, note that these symbolic heaps do not have to be derived using the SID Φ. For instance, they might contain additional pure formulas.

In particular, $H^{\mathcal{C}}_{P\mathbf{x},\Phi}$ can be accepted by a heap automaton for common predicates specifying data structures such as lists, trees, and trees with linked leaves. We are now in a position to decide entailments between predicate calls.

Lemma 9. *Let $\Phi \in \mathrm{SID}_{\mathcal{C}}$ and $P_1, P_2 \in \mathrm{Pred}(\Phi)$ be predicate symbols having the same arity. Moreover, let $\mathbb{U}_\Phi(P_1\mathbf{x})$ be well-determined and $H^{\mathcal{C}}_{P_2\mathbf{x},\Phi}$ be accepted by a heap automaton over $\mathrm{SH}_{\mathcal{C}}$. Then the entailment $P_1\mathbf{x} \models_\Phi P_2\mathbf{x}$ is decidable.*

Proof. Let $\mathfrak{A}_{P_2\mathbf{x}}$ be a heap automaton over $\mathrm{SH}_{\mathcal{C}}$ accepting $H^{\mathcal{C}}_{P_2\mathbf{x},\Phi}$. Then

$$\begin{aligned}
& P_1\mathbf{x} \models_\Phi P_2\mathbf{x} \\
\Leftrightarrow\ & \forall \sigma \in \mathbb{U}_\Phi(P_1\mathbf{x})\,.\,\exists \tau \in \mathbb{U}_\Phi(P_2\mathbf{x})\,.\,\sigma \models_\emptyset \tau && \text{(Lemma 8)} \\
\Leftrightarrow\ & \forall \sigma \in \mathbb{U}_\Phi(P_1\mathbf{x})\,.\,\sigma \in H^{\mathcal{C}}_{P_2\mathbf{x},\Phi} && \text{(Definition 12)} \\
\Leftrightarrow\ & \mathbb{U}_\Phi(P_1\mathbf{x}) \subseteq L(\mathfrak{A}_{P_2\mathbf{x}}). && (L(\mathfrak{A}_{P_2\mathbf{x}}) = H^{\mathcal{C}}_{P_2\mathbf{x},\Phi})
\end{aligned}$$

where the last inclusion is decidable by Corollary 2. □

6.2 Entailment between symbolic heaps

Our next step is to generalize Lemma 9 to arbitrary determined symbolic heaps φ instead of single predicate calls. This requires the construction of heap automata $\mathfrak{A}_\varphi$ accepting $H^{\mathcal{C}}_{\varphi,\Phi}$. W.l.o.g. we assume SIDs and symbolic heaps to be *well*-determined instead of determined only. Otherwise, we apply Theorem 1 with the heap automaton $\mathfrak{A}_{\mathtt{SAT}}$ (cf. Theorem 3) to obtain a *well*-determined SID. Thus, we restrict our attention to the following set.

Definition 13. *The set $\mathrm{SH}_{\langle\alpha\rangle}$ is given by $\langle\alpha\rangle : \mathrm{SH} \to \{0,1\}$, where $\langle\alpha\rangle(\varphi) = 1$ iff φ is well-determined and every predicate call of φ has $\leq \alpha \in \mathbb{N}$ parameters.*

Clearly, $\langle\alpha\rangle$ is decidable, because satisfiability is decidable (cf. Theorem 3) and verifying that a symbolic heap has at most α parameters amounts to a simple syntactic check. Note that, although the number of parameters in predicate calls is bounded by α, the number of free variables of a symbolic heap $\varphi \in \mathrm{SH}_{\langle\alpha\rangle}$ is not. We then construct heap automata for well-determined symbolic heaps.

Theorem 10 ([29]). *Let $\alpha \in \mathbb{N}$ and $\Phi \in \mathrm{SID}_{FV\leq\alpha}$ be established. Moreover, for each predicate symbol $P \in \mathrm{Pred}(\Phi)$, let there be a heap automaton over $\mathrm{SH}_{\langle\alpha\rangle}$ accepting $H^{\langle\alpha\rangle}_{P\mathbf{x},\Phi}$. Then, for every well-determined symbolic heap $\varphi \in \mathrm{SH}^{\Phi}$, there is a heap automaton over $\mathrm{SH}_{\langle\alpha\rangle}$ accepting $H^{\langle\alpha\rangle}_{\varphi,\Phi}$.*

Remark 3. Brotherston et al. [13] studied the *model-checking problem* for symbolic heaps, i.e., the question whether $s, h \models_\Phi \varphi$ holds for a given stack–heap pair (s, h), an SID Φ, and a symbolic heap $\varphi \in \mathrm{SH}_\Phi$. They showed that this problem is ExpTime–complete in general and NP–complete if the number of free variables is bounded. We obtain these results for *determined* symbolic heaps in a natural way: Observe that every stack–heap pair (s, h) is characterized by an established, well-determined, reduced symbolic heap, say τ, that has exactly (s, h)

Input : established SID Φ, $\varphi, \psi \in \mathrm{SH}^{\Phi}$ determined, heap automaton $\mathfrak{A}_{P_i}$ for each $P_i \in \mathrm{Pred}(\Phi)$
Output: yes iff $\varphi \models_{\Phi} \psi$ holds

```
$\Omega$ ← $\{P\mathbf{x}_0^{\varphi} \Leftarrow \varphi\} \cup \Phi$ ;              // $P$ fresh predicate symbol
$\Psi$ ← removeUnsat($\Omega$) ;                                        // Theorem 3
$\mathfrak{A}_{\psi}$ ← automaton($\psi, \mathfrak{A}_{P_1}, \mathfrak{A}_{P_2}, \ldots$) ;            // Theorem 10
$\overline{\mathfrak{A}_{\psi}}$ ← complement($\mathfrak{A}_{\psi}$) ;                                   // Lemma 2
return  yes iff $\mathbb{U}_{\Psi}(P\mathbf{x}) \cap L(\overline{\mathfrak{A}_{\psi}}) = \emptyset$ ;         // Algorithm 1
```

Algorithm 2: Decision procedure for $\varphi \models_{\Phi} \psi$.

as a tight model up to isomorphism. Then Theorem 10 yields a heap automaton $\mathfrak{A}_{\tau}$ accepting $H_{\tau,\Phi}^{\langle\alpha\rangle}$, where α is the maximal arity of any predicate in Φ. Thus, $s, h \models_{\Phi} \varphi$ iff $L(\mathfrak{A}_{\tau}) \cap \mathbb{U}_{\Phi}(\varphi) \neq \emptyset$, which is decidable by Corollary 1. Further, note that the general model-checking problem is within the scope of heap automata. A suitable state space is the set of all subformulas of the symbolic heap τ.

Coming back to the entailment problem, it remains to put our results together. Algorithm 2 depicts a decision procedure for the entailment problem that, given an entailment $\varphi \models_{\Phi} \psi$, first removes all unsatisfiable unfoldings of φ, i.e. φ becomes well-determined. After that, our previous reasoning techniques for heap automata and SIDs from Section 3 are applied to decide whether $\varphi \models_{\Phi} \psi$ holds.

Theorem 11. *Let $\alpha \in \mathbb{N}$ and $\Phi \in \mathrm{SID}_{FV \leq \alpha}$ be established. Moreover, for every $P \in \mathrm{Pred}(\Phi)$, let $H_{P\mathbf{x},\Phi}^{\langle\alpha\rangle}$ be accepted by a heap automaton over $\mathrm{SH}_{\langle\alpha\rangle}$. Then $\varphi \models_{\Phi} \psi$ is decidable for determined $\varphi, \psi \in \mathrm{SH}^{\Phi}$ with $\mathbf{x}_0^{\varphi} = \mathbf{x}_0^{\psi}$.*

Proof. We define a new SID $\Omega \triangleq \Phi \cup \{P\mathbf{x} \Leftarrow \varphi\}$, where P is a fresh predicate symbol of arity $\|\mathbf{x}_0^{\varphi}\|$. Clearly, $\varphi \models_{\Phi} \psi$ iff $P\mathbf{x} \models_{\Omega} \psi$. Since φ and Φ are established, so is Ω. Then applying Theorem 1 to Ω and $\mathfrak{A}_{\mathtt{SAT}}$ (cf. Theorem 3), we obtain a well-determined SID $\Psi \in \mathrm{SID}_{\langle\alpha\rangle}$ where none of the remaining unfoldings of Ω is changed, i.e., for each $P \in \mathrm{Pred}(\Omega)$, we have $\mathbb{U}_{\Psi}(P\mathbf{x}) \subseteq \mathbb{U}_{\Omega}(P\mathbf{x})$. By Theorem 10, the set $H_{\psi,\Phi}^{\langle\alpha\rangle} = H_{\psi,\Psi}^{\langle\alpha\rangle}$ can be accepted by a heap automaton over $\mathrm{SH}_{\langle\alpha\rangle}$. Then, analogously to the proof of Lemma 9, $\varphi \models_{\Phi} \psi$ iff $P\mathbf{x} \models_{\Psi} \psi$ iff $\mathbb{U}_{\Psi}(P\mathbf{x}) \subseteq H_{\psi,\Psi}^{\langle\alpha\rangle}$, where the last inclusion is decidable by Corollary 2. □

6.3 Complexity

Algorithm 2 may be fed with arbitrarily large heap automata. For a meaningful complexity analysis, we thus consider heap automata of bounded size only.

Definition 14. *An SID Φ is α–*bounded *if for each $P \in \mathrm{Pred}(\Phi)$ there exists a heap automaton $\mathfrak{A}_P$ over $\mathrm{SH}_{\langle\alpha\rangle}$ accepting $H_{P\mathbf{x},\Phi}^{\langle\alpha\rangle}$ such that $\Delta_{\mathfrak{A}_P}$ is decidable in $\mathcal{O}\left(2^{poly(\|\Phi\|)}\right)$ and $\|Q_{\mathfrak{A}_P}\| \leq 2^{poly(\alpha)}$.*

The bounds from above are natural for a large class of heap automata. In particular, all heap automata constructed in Section 4 stay within these bounds. Then a close analysis of Algorithm 2 for α–bounded SIDs yields the following complexity results. A detailed analysis is provided in [29].

Theorem 12. SL-ENTAIL$^{\Phi}_{\langle\alpha\rangle}$ *is decidable in* 2-ExpTime *for every* α*–bounded SID* Φ*. If* $\alpha \geq 1$ *is a constant then* SL-ENTAIL$^{\Phi}_{\langle\alpha\rangle}$ *is* ExpTime*-complete.*

Note that lower complexity bounds depend on the SIDs under consideration. Antonopoulos et al. [1, Theorem 6] showed that the entailment problem is already Π_2^P–complete (the second level of the polynomial hierarchy) for the base fragment, i.e., $\Phi = \emptyset$. Thus, under common complexity assumptions, the exponential time upper bound derived in Theorem 12 is asymptotically optimal for a deterministic algorithm. Since the entailment problem is already ExpTime–hard for points-to assertions of arity 3 and SIDs specifying regular sets of trees (cf. [1, Theorem 5] and [29]), exponential time is actually needed for certain SIDs.

6.4 Expressiveness

Most common data structures specified by SIDs, such as lists, trees, trees with linked leaves and combinations thereof can be encoded by heap automata [29]. However, SIDs are more expressive than heap automata. For example, consider two concatenated lists of the same length that use different fields. While such lists are outside the scope of heap automata, a suitable SID is given by:

$$P(x,y) \Leftarrow \exists z.x \mapsto (z, \mathbf{null}) * z \mapsto (\mathbf{null}, y)$$
$$P(x,y) \Leftarrow \exists u,v.x \mapsto (u, \mathbf{null}) * P(u,v) * v \mapsto (\mathbf{null}, y)$$

In general, the close relationship between established SIDs and context-free graph languages studied by Dodds [19, Theorem 1] and Courcelle's work on recognizable graph languages [18, Theorems 4.34 and 5.68], suggest that heap automata exist for every set of reduced symbolic heaps that can be specified in monadic second-order logic over graphs [18].

7 Conclusion

We developed an algorithmic framework for automatic reasoning about and debugging of the symbolic heap fragment of separation logic. Our approach is centered around a new automaton model, *heap automata*, that is specifically tailored to symbolic heaps. We show that many common robustness properties as well as certain types of entailments are naturally covered by our framework—often with optimal asymptotic complexity. There are several directions for future work including automated learning of heap automata accepting common data structures and applying heap automata to the abduction problem [16].

Acknowledgements We thank Tomer Kotek, Georg Weissenbacher and the anonymous reviewers for their helpful comments.

References

1. Antonopoulos, T., Gorogiannis, N., Haase, C., Kanovich, M.I., Ouaknine, J.: Foundations for decision problems in separation logic with general inductive predicates. In: FOSSACS 2014. LNCS, vol. 8412, pp. 411–425. Springer (2014)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer (2007)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: FSTTCS 2004. LNCS, vol. 3328, pp. 97–109 (2004)
4. Berdine, J., Calcagno, C., Ohearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: International Symposium on Formal Methods for Components and Objects. pp. 115–137. Springer (2005)
5. Berdine, J., Calcagno, C., OHearn, P.W.: Symbolic execution with separation logic. In: APLAS 2005, LNCS, vol. 3780, pp. 52–68. Springer (2005)
6. Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory safety for systems-level code. In: CAV 2011. LNCS, vol. 6806, pp. 178–183. Springer (2011)
7. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: ACM SIGPLAN Notices. vol. 40, pp. 259–270. ACM (2005)
8. Botincan, M., Distefano, D., Dodds, M., Grigore, R., Naudziuniene, D., Parkinson, M.J.: coreStar: The core of jStar. BOOGIE 2011, 65–77 (2011)
9. Brookes, S.: A semantics for concurrent separation logic. Theoretical Computer Science 375(1), 227–270 (2007)
10. Brotherston, J., Distefano, D., Petersen, R.L.: Automated cyclic entailment proofs in separation logic. In: CADE-23. LNAI, vol. 6803, pp. 131–146. Springer (2011)
11. Brotherston, J., Fuhs, C., Pérez, J.A.N., Gorogiannis, N.: A decision procedure for satisfiability in separation logic with inductive predicates. In: CSL-LICS 2014. pp. 25:1–25:10. ACM (2014)
12. Brotherston, J., Gorogiannis, N.: Cyclic abduction of inductively defined safety and termination preconditions. In: SAS 2014. LNCS, vol. 8723, pp. 68–84. Springer (2014)
13. Brotherston, J., Gorogiannis, N., Kanovich, M.I., Rowe, R.: Model checking for symbolic-heap separation logic with inductive predicates. In: POPL 2016. pp. 84–96. ACM (2016)
14. Brotherston, J., Gorogiannis, N., Petersen, R.L.: A generic cyclic theorem prover. In: APLAS 2012. pp. 350–367. Springer (2012)
15. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: NFM 2011. LNCS, vol. 6617, pp. 459–465. Springer (2011)
16. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL 2009. pp. 289–300. ACM (2009)
17. Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. 77(9), 1006–1036 (2012)
18. Courcelle, B., Engelfriet, J.: Graph structure and monadic second-order logic: a language-theoretic approach, vol. 138. Cambridge University Press (2012)
19. Dodds, M.: From separation logic to hyperedge replacement and back. In: ICGT 2008. pp. 484–486. Springer (2008)
20. Dudka, K., Peringer, P., Vojnar, T.: Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In: CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer (2011)

21. Enea, C., Lengál, O., Sighireanu, M., Vojnar, T.: Compositional entailment checking for a fragment of separation logic. In: APLAS 2014. LNCS, vol. 8837, pp. 314–333. Springer (2014)
22. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI 2007. pp. 266–277. ACM (2007)
23. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. In: CAV 2011. LNCS, vol. 6806, pp. 424–440. Springer (2011)
24. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. Formal Methods in System Design 41(1), 83–106 (2012)
25. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
26. Iosif, R., Rogalewicz, A., Simacek, J.: The tree width of separation logic with recursive definitions. In: CADE-24. LNCS, vol. 7898, pp. 21–38. Springer (2013)
27. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding entailments in inductive separation logic with tree automata. In: ATVA 2014. LNCS, vol. 8837, pp. 201–218. Springer (2014)
28. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer (2011)
29. Jansen, C., Katelaan, J., Matheja, C., Noll, T., Zuleger, F.: Unified reasoning about robustness properties of symbolic-heap separation logic. CoRR abs/1610.07041 (2016), http://arxiv.org/abs/1610.07041
30. Le, Q.L., Gherghina, C., Qin, S., Chin, W.N.: Shape analysis via second-order bi-abduction. In: CAV 2014. LNCS, vol. 8559, pp. 52–68. Springer (2014)
31. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Thor: A tool for reasoning about shape and arithmetic. In: CAV 2008. LNCS, vol. 5123, pp. 428–432. Springer (2008)
32. Navarro Pérez, J., Rybalchenko, A.: Separation logic modulo theories. In: APLAS 2013, LNCS, vol. 8301, pp. 90–106. Springer (2013)
33. Nguyen, H.H., Kuncak, V., Chin, W.N.: Runtime checking for separation logic. In: VMCAI 2008. LNCS, vol. 4905, pp. 203–217. Springer (2008)
34. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (2007)
35. OHearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: International Workshop on Computer Science Logic. pp. 1–19. Springer (2001)
36. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: CAV 2014. LNCS, vol. 8559, pp. 711–728. Springer (2014)
37. Qiu, X., Garg, P., Ştefănescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: PLDI 2013. pp. 231–242. ACM (2013)
38. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002. pp. 55–74. IEEE (2002)
39. Zanardini, D., Genaim, S.: Inference of field-sensitive reachability and cyclicity. ACM Trans. Comput. Log. 15(4), 33:1–33:41 (2014)