Effective Entailment Checking for Separation Logic with Inductive Definitions

Jens Katelaan[1(✉)], Christoph Matheja[2], and Florian Zuleger[1]

[1] TU Wien, Vienna, Austria
jkatelaan@forsyte.at
[2] RWTH Aachen University, Aachen, Germany

Abstract. Symbolic-Heap Separation logic is a popular formalism for automated reasoning about heap-manipulating programs, which allows the user to give customized data structure definitions.

In this paper, we give a new decidability proof for the separation logic fragment of Iosif, Rogalewicz and Simacek. We circumvent the reduction to MSO from their proof and provide a direct model-theoretic construction with elementary complexity. We implemented our approach in the Harrsh analyzer and evaluate its effectiveness. In particular, we show that Harrsh can decide the entailment problem for data structure definitions for which no previous decision procedures have been implemented.

1 Introduction

Separation logic (SL) [12,18] is a popular formalism for Hoare-style verification of imperative, heap-manipulating programs. In particular, the *symbolic heap* separation logic fragment has received a lot of attention: Symbolic heaps serve as the basis of various automated verification tools, such as Infer [6], Sleek [7], Songbird [19], GRASShopper [17], VCDryad [16], VeriFast [13], SLS [20], and Spen [9]. Many of the aforementioned tools rely on *systems of inductive predicate definitions* (SID) that serve as specifications of dynamic data structures, e.g., linked lists and trees.

At the heart of every Hoare-style verification procedure based on separation logic lies the *entailment problem*: Given two SL formulas, say φ and ψ, is every model of φ also a model of ψ? While the entailment problem is undecidable in general [2], there are various approaches to decide entailments between symbolic heaps ranging from complete methods for fixed SIDs [3], over decision procedures for restricted classes of SIDs [10,11], to incomplete approaches, such as fold/unfold reasoning [7] or cyclic proofs [5].

Among the largest decidable fragments of symbolic heaps with inductive definitions is the fragment of *symbolic heaps with bounded tree-width* ($\mathrm{SL}_{\mathrm{btw}}$) [10]. This fragment supports a rich class of data structures in SID specifications, such as doubly-linked lists and binary trees with linked leaves. $\mathrm{SL}_{\mathrm{btw}}$ introduces

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part II, LNCS 11428, pp. 319–336, 2019.
https://doi.org/10.1007/978-3-030-17465-1_18

$$
\begin{array}{rcl}
\texttt{tree}(x_1, x_2) & \Leftarrow & x_1 \mapsto (\mathbf{null}, \mathbf{null}, x_2) \\
\texttt{tree}(x_1, x_2) & \Leftarrow & \exists \ell, r \colon x_1 \mapsto (\ell, r, x_2) * \texttt{tree}(\ell, x_1) * \texttt{tree}(r, x_1) \\
\hline
\texttt{rtree}(x_1, x_2, x_3) & \Leftarrow & \exists r \colon x_1 \mapsto (x_3, r, x_2) * \texttt{parent}(x_3, x_1) * \texttt{tree}(r, x_1) \\
\texttt{parent}(x_1, x_2) & \Leftarrow & x_1 \mapsto (\mathbf{null}, \mathbf{null}, x_2) \\
\texttt{rtree}(x_1, x_2, x_3) & \Leftarrow & \exists \ell, r \colon x_1 \mapsto (\ell, r, x_2) * \texttt{rtree}(\ell, x_1, x_3) * \texttt{tree}(r, x_1) \\
\hline
\texttt{ltree}(x_1, x_2, x_3) & \Leftarrow & \exists p \colon x_1 \mapsto (\mathbf{null}, \mathbf{null}, p) * \texttt{lltree}(p, x_1, x_2, x_3) \\
\texttt{lltree}(x_1, x_2, x_3, x_4) & \Leftarrow & \exists r \colon x_1 \mapsto (x_2, r, x_3) * \texttt{tree}(r, x_1) * \texttt{lroot}(x_3, x_1, x_4) \\
\texttt{lltree}(x_1, x_2, x_3, x_4) & \Leftarrow & \exists r, p \colon x_1 \mapsto (x_2, r, p) * \texttt{lltree}(p, x_1, x_3, x_4) * \texttt{tree}(r, x_1) \\
\texttt{lroot}(x_1, x_2, x_3) & \Leftarrow & \exists r \colon x_1 \mapsto (x_2, r, x_3) * \texttt{tree}(r, x_1)
\end{array}
$$

Fig. 1. An SID Φ with three predicates for binary trees with parent pointers.

three syntactic conditions on SIDs—progress, connectivity, and establishment—that enable a reduction from the entailment problem for $\mathrm{SL}_{\mathrm{btw}}$ to the (decidable) satisfiability problem for monadic second-order logic (MSO) over graphs of bounded tree width. This gives rise to a decision procedure of non-elementary complexity—at least without an in-depth analysis of the quantifier alternations involved in the reduction. The reduction to MSO is also technically involved and has—to the best of our knowledge—never been implemented. The authors remark in the follow-up paper [11] that "the method from [10] causes a blowup of several exponentials in the size of the input problem and is unlikely to produce an effective decision procedure."

Contributions. We give a new proof for the decidability of the entailment problem for the $\mathrm{SL}_{\mathrm{btw}}$ fragment. In contrast to [10], we circumvent the reduction to MSO and give a direct model-theoretic construction with elementary complexity. This yields an easy-to-implement decision procedure for entailments in the full $\mathrm{SL}_{\mathrm{btw}}$ fragment. We implemented our approach in the Harrsh analyzer and report on promising results for challenging examples (Sect. 6). In particular, we show that Harrsh can decide the entailment problem for data structure definitions for which no previous decision procedures have been implemented.

A challenging example. To highlight the challenges faced when developing and implementing decision procedures for entailments in $\mathrm{SL}_{\mathrm{btw}}$, consider the SID Φ consisting of the rules in Fig. 1.[1] There are three predicates, namely `tree`, `rtree`, and `ltree`, that specify binary trees with parent pointers (treep for short). The predicate `tree` takes two parameters representing the root of the tree and its parent. Predicates `rtree` and `ltree` both have the leftmost leaf of the tree as an additional parameter. Such a parameter may, for example, be required to specify tree segments for an automated program analysis. Although both `rtree` and `ltree` describe treeps, they take radically different approaches: Predicate `rtree` defines a treep starting at the root, i.e., it specifies the root of the treep

[1] The syntax and semantics of SIDs are defined formally in Sect. 3.

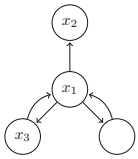

Fig. 2. treep

and then states that both subtrees are treeps (the parameter representing the leftmost leaf is additionally passed to the left subtree). In contrast, predicate `ltree` specifies treeps starting at the leftmost leaf and moving up to the root. Consequently, the models of these predicates are derived in completely different ways, which is a challenge for commonly applied approaches, such as fold/unfold (cf. [7]) or inductive reasoning (cf. [5,19,20]). In fact, the entailment $\texttt{ltree}(x_1, x_2, x_3) \models \texttt{rtree}(x_2, x_3, x_1)$ holds, whereas the entailment $\texttt{rtree}(x_2, x_3, x_1) \models \texttt{ltree}(x_1, x_2, x_3)$ is violated: Intuitively, `rtree` admits models in which all shortest paths from the root to the leftmost leaf have length one. In contrast, for `ltree`, the minimal length of all shortest paths is two. Thus, the heap illustrated in Fig. 2 is a model of `rtree`, but not of `ltree`. In fact, if we rule out this model, `rtree` and `ltree` entail each other. That is, the entailment below and its converse are both valid:

$$\texttt{ltree}(x_1, x_2, x_3) \models \exists \ell, r \colon x_2 \mapsto (l, r, x_3) * \texttt{rtree}(l, x_2, x_1) * \texttt{tree}(r, x_2) \quad (\clubsuit)$$

Harrsh solved the entailment ($\clubsuit$) from above in less than a second. The only other tool capable of successfully solving ($\clubsuit$) is Slide [11], which is based on tree automata. However, the approach in [11] is not complete for $\mathrm{SL}_{\mathrm{btw}}$.

Overview of our approach. We first present an algebra à la Courcelle [8] to systematically construct models of separation logic formulas (Sect. 2). This algebra enables us to conveniently formalize the semantics of separation logic (Sect. 3). To decide entailments, we then develop an abstraction mechanism for models with the following properties (Sect. 4):

1. The abstraction is *compositional*, i.e., we can perform our algebraic operations on abstractions instead of models (Theorem 2).
2. The abstraction is *finite*, i.e., each model of a predicate is abstracted to one of finitely many abstractions (Lemma 3).
3. The abstraction *refines* the predicate satisfaction relation, i.e., models with the same abstraction entail the same predicates among those relevant for the entailment (Lemma 2).
4. The abstraction is *effective*, i.e., for a given abstraction, one can determine which predicates are entailed (Theorem 3).

How do we obtain a decision procedure from these properties for an entailment, say $\mathsf{pred}_1(\mathbf{x_1}) \models_\Phi \mathsf{pred}_2(\mathbf{x_2})$? We iteratively compute all abstractions corresponding to models of $\mathsf{pred}_1(\mathbf{x_1})$. Due to compositionality (1), this can be achieved by applying the same operations used to construct models on previously computed abstractions until a fixed point is reached. Finiteness of the abstraction (2) ensures termination. We then exploit that the abstraction is well-defined (3) and effective (4) to decide the entailment: $\mathsf{pred}_1(\mathbf{x_1}) \models_\Phi \mathsf{pred}_2(\mathbf{x_2})$ holds iff all computed abstractions of models of $\mathsf{pred}_1(\mathbf{x_1})$ yield that they are also models of $\mathsf{pred}_2(\mathbf{x_2})$ (Sect. 5).

Due to space constraints, all proofs are in the supplementary material [1].

Notation. The set of all (non-empty) finite sequences over a set S is S^* (S^+). Bold letters denote sequences, e.g., $\mathbf{x} = \langle x_1, \ldots, x_k \rangle$. $\mathbf{x}[i]$ refers to the i-th element of $\mathbf{x}$. We often treat sequences as sets, i.e. we write $y \in \mathbf{x}$ if y occurs in $\mathbf{x}$, $\mathbf{x} \cup \mathbf{z}$ for the set of all elements in $\mathbf{x}$ or $\mathbf{z}$, etc. $f = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ is the function given by $f(x_i) = y_i$ for $i \in [1, n]$, $n \geq 0$. Moreover, functions $f \colon X \to Y$ are lifted to functions on sequences $f \colon X^* \to Y^*$ by pointwise application.

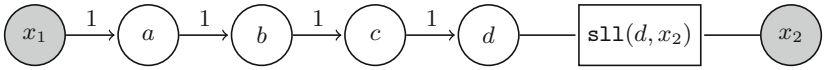

Fig. 3. A heap graph modeling a list segment of length at least 5 from x_1 to x_2.

2 Heap Graphs

Separation logic is typically interpreted in terms of stack-heap pairs consisting of a stack, i.e., an evaluation of variables, and a heap, i.e., a finite mapping from memory locations to values. In our setting, however, it is more convenient to abstract from locations and consider labeled graphs.

Formally, let **Var** be a set of variables containing a special variable **null** $\in$ **Var**. Moreover, let **Preds** be a set of *predicate identifiers*; each predicate $\mathsf{pred} \in$ **Preds** is equipped with an arity $\mathrm{ar}(\mathsf{pred}) \in \mathbb{N}$. $\mathsf{pred}(\mathbf{x})$ is a *predicate call* if the length of sequence $\mathbf{x} \in \mathbf{Var}^*$ is $\mathrm{ar}(\mathsf{pred})$.

Definition 1 (Heap Graph). *A* heap graph $\mathcal{M} = \langle \mathsf{Ptr}, \mathsf{FV}, \mathsf{calls} \rangle$ *is a graph whose nodes are a finite set of variables in* **Var**. *The edges of* $\mathcal{M}$ *are given by a partial points-to function* $\mathsf{Ptr} \colon \mathbf{Var} \setminus \{\mathbf{null}\} \rightharpoonup_{\mathit{finite}} \mathbf{Var}^+$ *mapping variables to finite tuples of variables. Moreover,* $\mathsf{FV} \subseteq \mathbf{Var}$ *is a finite set of* free variables *and* calls *is a finite set of* predicate calls. *A heap graph is* concrete *if* $\mathsf{calls} = \emptyset$. *We collect all variables in* Ptr, FV, *and* calls *in* $\mathsf{vars}(\mathcal{M})$. *Finally, we write* $\mathsf{Ptr}_{\mathcal{M}}$, $\mathsf{FV}_{\mathcal{M}}$, *and* $\mathsf{calls}_{\mathcal{M}}$ *to refer to the individual components of heap graph* $\mathcal{M}$. △

Example 1. Figure 3 depicts a heap graph modeling a singly-linked list of length at least five with head x_1 and tail x_2 (assuming the predicate call $\mathtt{sll}(d, x_2)$ stands for non-empty lists segments from d to x_2; see the left part of Fig. 5). In our graphical notation, every node corresponds to the variable it is labeled with. Gray nodes correspond to the free variables in FV. For each variable, say x, the pointers $\mathsf{Ptr}(x) = \langle y_1, \ldots, y_k \rangle$ are represented by directed edges—labeled with the position $1, 2, \ldots, k$—from the node labeled with x to nodes labeled with $y_1, \ldots, y_k$, respectively. We usually omit the edge labels if each node has at most one outgoing edge. Finally, a predicate call is drawn as a box labeled with the predicate call and connected to the nodes representing the variables occurring in the call's parameters. Formally, the heap graph in Fig. 3 is given by $\mathcal{M} = \langle \mathsf{Ptr}, \mathsf{FV}, \mathsf{calls} \rangle$ with points-to mapping $\mathsf{Ptr} = \{x_1 \mapsto a, a \mapsto b, b \mapsto c, c \mapsto d\}$, free variables $\mathsf{FV} = \{x_1, x_2\}$ and predicate calls $\mathsf{calls} = \{\mathtt{sll}(d, x_2)\}$. △

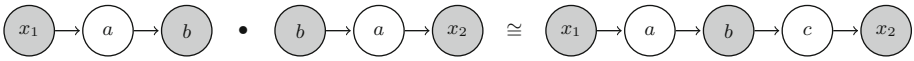

Fig. 4. Illustration of composition of two heap graphs.

Heap graphs are an abstraction of the classical stack-heap model. To reason about separation logic with heap graphs (and their abstractions), we need a few operations for their systematic construction: Let $f\colon \mathbf{Var} \to \mathbf{Var}$ be a partial function and $f(\mathcal{M})$ its application to every variable in every component of $\mathcal{M}$.

Isomorphic heap graphs. We call a variable $x \in \mathbf{Var}$ an *auxiliary variable* of heap graph $\mathcal{M}$ if x is not a free variable of $\mathcal{M}$. Throughout this article, we do not distinguish between isomorphic heap graphs, i.e., heap graphs that are identical up to renaming of auxiliary variables. Formally, two heap graphs $\mathcal{M}_1$ and $\mathcal{M}_2$ are *isomorphic*, written $\mathcal{M}_1 \cong \mathcal{M}_2$, if there exists a bijective function $f\colon \mathsf{vars}(\mathcal{M}_1) \to \mathsf{vars}(\mathcal{M}_2)$ such that (1) $\mathsf{FV}_{\mathcal{M}_1} = \mathsf{FV}_{\mathcal{M}_2}$, (2) $f(x) = x$ for all $x \in \mathsf{FV}_{\mathcal{M}_1}$, and (3) $f(\mathcal{M}_1) = \mathcal{M}_2$.

Renaming heap graphs. Our first operation enables renaming of free variables. Formally, let $\mathcal{M}$ be a heap graph and $\mathbf{x} \in \mathsf{FV}^*_{\mathcal{M}}$, $\mathbf{y} \in \mathbf{Var}^*$ be repetition free sequences of variables of the same length. Then the *renaming* of $\mathbf{x}$ to $\mathbf{y}$ in $\mathcal{M}$ is given by $\mathsf{rename}_{\mathbf{x},\mathbf{y}}(\mathcal{M}) = f(\mathcal{M})$, where

$$f\colon \mathbf{Var} \to \mathbf{Var}, \quad z \mapsto \begin{cases} \mathbf{y}[i] & \text{if } \mathbf{x}[i] = z \\ z & \text{otherwise.} \end{cases}$$

Composition. Our next operation allows composing heap graphs by "gluing" them together at their common free variables. Formally, let $\mathcal{M}_1, \mathcal{M}_2$ be heap graphs such that (1) $\mathsf{vars}(\mathcal{M}_1) \cap \mathsf{vars}(\mathcal{M}_2) \subseteq \mathsf{FV}_{\mathcal{M}_1} \cap \mathsf{FV}_{\mathcal{M}_2}$ and (2) $\mathsf{Ptr}_{\mathcal{M}_1}$ and $\mathsf{Ptr}_{\mathcal{M}_1}$ are domain disjoint, i.e., $\mathrm{dom}(\mathsf{Ptr}_{\mathcal{M}_1}) \cap \mathrm{dom}(\mathsf{Ptr}_{\mathcal{M}_2}) = \emptyset$. Then the componentwise union $\mathcal{M}_1 \cup \mathcal{M}_2$ of $\mathcal{M}_1$ and $\mathcal{M}_2$ is $\langle \mathsf{Ptr}_{\mathcal{M}_1} \cup \mathsf{Ptr}_{\mathcal{M}_2}, \mathsf{FV}_{\mathcal{M}_1} \cup \mathsf{FV}_{\mathcal{M}_2}, \mathsf{calls}_{\mathcal{M}_1} \cup \mathsf{calls}_{\mathcal{M}_2} \rangle$. Otherwise, $\mathcal{M}_1 \cup \mathcal{M}_2$ is undefined. We then define the composition $\mathcal{M}_1 \bullet \mathcal{M}_2$ of heap graphs $\mathcal{M}_1, \mathcal{M}_2$ as

$$\mathcal{M}_1 \bullet \mathcal{M}_2 = \begin{cases} \mathcal{M}_1 \cup \mathcal{M} & \text{where } \mathcal{M} \cong \mathcal{M}_2 \text{ and } \mathcal{M}_1 \cup \mathcal{M} \text{ is defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Example 2. Figure 4 depicts the composition of two heap graphs representing lists of length two. Since both heap graphs share a variable $a \notin \mathsf{FV}$, we first compute an isomorphic heap graph in which variable a is substituted by c in the second graph. Both heap graphs are then merged at their common free variable b. This results in a heap graph modeling a list of length four. $\triangle$

Forgetting free variables. To construct larger heap graphs from smaller ones, we often need additional free variables to glue the right nodes together, e.g., the variable b in Example 2. Consequently, we need a mechanism for subsequent removal of these variables from the set of free variables. To this end, for every heap graph $\mathcal{M}$ and sequence of free variables $\mathbf{x} \in \mathsf{FV}^*_{\mathcal{M}}$, we define the operation $\mathsf{forget}_{\mathbf{x}}(\mathcal{M}) = \langle \mathsf{Ptr}_{\mathcal{M}}, \mathsf{FV}_{\mathcal{M}} \setminus \mathbf{x}, \mathsf{calls}_{\mathcal{M}} \rangle$.

Single allocations. The simplest non-empty heap graph is a single variable, say x with pointers to a sequence $\mathbf{y}$ of finitely many other variables. We write $x \rightarrowtail \mathbf{y}$ to denote this *single-allocation heap graph* $\langle \{x \mapsto \mathbf{y}\}, \{x\} \cup \mathbf{y}, \emptyset \rangle$.

Theorem 1 ([8]). *Every non-empty heap graph of tree width at most k can be constructed from heap graphs $x \rightarrowtail \mathbf{y}$, renaming, composition, and forgetting using at most $k+1$ free variables.*

3 Symbolic Heap Separation Logic

We consider the symbolic heap fragment of separation logic with user-defined inductive predicate definitions. We omit pure formulas to simplify the presentation. Notice, however, that our implementation supports reasoning about symbolic heaps with pure formulas.

Syntax. The syntax of our simplified symbolic heap fragment is then given by the following context-free grammar:

$$\varphi ::= \mathbf{emp} \mid x \mapsto \mathbf{y} \mid \mathsf{pred}(\mathbf{y}) \mid \exists x \colon \varphi \mid \varphi * \varphi,$$

where $x \in \mathbf{Var} \setminus \{\mathbf{null}\}$ is a variable, $\mathbf{y} \in \mathbf{Var}^+$ is a sequence of variables, and $\mathsf{pred}(\mathbf{y})$ is a predicate call. Here, $\mathbf{emp}$ is the *empty heap*, $x \mapsto \mathbf{y}$ asserts that x *points-to* the locations captured by $\mathbf{y}$, $\exists x \colon \varphi$ is *existential quantification*, and $*$ is the *separating conjunction*. Because $*$ is commutative and associative and because existential quantifiers can always be moved to the front, we will always consider symbolic heaps to be of form $\exists \mathbf{y} \colon (x_1 \mapsto \mathbf{y}_1) * \cdots * (x_m \mapsto \mathbf{y}_m) * \mathsf{pred}_1(\mathbf{z}_1) * \cdots * \mathsf{pred}_n(\mathbf{z}_n)$.

Inductive definitions. Before we assign formal semantics to symbolic heaps, we clarify how custom predicates are specified. To this end, a *system of inductive definitions* (SID) is a finite set Φ of rules of the form $\mathsf{pred} \Leftarrow \varphi$, where $\mathsf{pred} \in \mathbf{Preds}$ is a predicate symbol and φ is a symbolic heap. We assume that all symbolic heaps of rules with head pred have the same sequence of free variables $(x_1, \ldots, x_{\mathrm{ar}(\mathsf{pred})})$[2] and collect these variables in the set $\mathsf{fv}(\mathsf{pred})$. Moreover, we collect all predicates that occur in SID Φ in the set $\mathbf{Preds}(\Phi)$ and all rules of SID Φ in the set $\mathbf{Rules}(\Phi)$. Examples of SIDs are found in Figs. 1 and 5.

[2] A variable is in the set $\mathsf{fv}(\varphi)$ of free variables of φ if it is not bound by a quantifier.

Semantics. We define the semantics of symbolic heaps φ for a given SID Φ in terms of a force relation $\models_\Phi$, which determines whether a heap graph $\mathcal{M}$ satisfies φ. To this end, let $\varphi[\mathbf{x}/\mathbf{y}]$ denote the symbolic heap φ in which every free occurrence of variable $\mathbf{x}[i]$ is substituted by variable $\mathbf{y}[i]$, where $1 \leq i \leq |\mathbf{x}| = |\mathbf{y}|$. Then the relation $\models_\Phi$ is defined inductively on the syntax of symbolic heaps:

$$
\begin{aligned}
\mathcal{M} \models_\Phi \mathbf{emp} &\text{ iff ex. } \mathbf{x} \in \mathbf{Var}^* \text{ s.t. } \mathcal{M} = \langle \emptyset, \mathbf{x}, \emptyset \rangle \\
\mathcal{M} \models_\Phi x \mapsto \mathbf{y} &\text{ iff ex. } \mathbf{z} \supseteq \{x\} \cup \mathbf{y} \text{ s.t. } \mathcal{M} = \langle \{x \mapsto \mathbf{y}\}, \mathbf{z}, \emptyset \rangle \\
\mathcal{M} \models_\Phi \mathsf{pred}(\mathbf{y}) &\text{ iff ex. } \mathbf{z} \supseteq \mathbf{y} \text{ s.t. } \mathcal{M} \cong \langle \emptyset, \mathbf{z}, \{\mathsf{pred}(\mathbf{y})\} \rangle \\
&\quad \text{or ex. } (\mathsf{pred} \Leftarrow \psi) \in \mathbf{Rules}(\Phi) \text{ s.t. } \mathcal{M} \models_\Phi \psi[\mathsf{fv}(\mathsf{pred})/\mathbf{y}] \\
\mathcal{M} \models_\Phi \exists x \colon \varphi &\text{ iff ex. } y \in \mathbf{Var} \text{ s.t. } \langle \mathsf{Ptr}_\mathcal{M}, \mathsf{FV}_\mathcal{M} \cup \{y\}, \mathsf{calls}_\mathcal{M} \rangle \models_\Phi \varphi[x/y] \\
\mathcal{M} \models_\Phi \varphi_1 * \varphi_2 &\text{ iff ex. } \mathcal{M}_1, \mathcal{M}_2 \text{ s.t. } \mathcal{M} \cong \mathcal{M}_1 \bullet \mathcal{M}_2 \\
&\quad \text{and } \mathcal{M}_1 \models_\Phi \varphi_1 \text{ and } \mathcal{M}_2 \models_\Phi \varphi_2
\end{aligned}
$$

The above semantics coincides with the standard least fixed-point semantics of symbolic heaps (cf. [4]) for stack-heap pairs if we restrict ourselves to concrete heap graphs. Moreover, there is a strong relationship between our SL semantics and the operations on heap graphs defined in Sect. 2.

Lemma 1. *Let* $\varphi = \exists \mathbf{y} \colon (x_1 \mapsto \mathbf{y}_1) * \cdots * (x_m \mapsto \mathbf{y}_m) * \mathsf{pred}_1(\mathbf{z}_1) * \cdots * \mathsf{pred}_n(\mathbf{z}_n)$ *be a symbolic heap.* $\mathcal{M} \models_\Phi \varphi$ *iff there exist* $\mathcal{M}_1, \ldots, \mathcal{M}_{m+n}$ *such that (1)* $\mathcal{M}_i \models_\Phi x \mapsto \mathbf{y}_i$ *for* $1 \leq i \leq m$*, (2)* $\mathcal{M}_{m+j} \models_\Phi \mathsf{pred}_j(\mathsf{fv}(\mathsf{pred}_j))$ *for* $1 \leq j \leq n$*, and (3)* $\mathcal{M} = \mathsf{forget}_\mathbf{y}(\mathcal{M}_1 \bullet \cdots \bullet \mathcal{M}_m \bullet \mathsf{rename}_{\mathsf{fv}(\mathsf{pred}_1),\mathbf{z}_1}(\mathcal{M}_{m+1}) \bullet \cdots \bullet \mathsf{rename}_{\mathsf{fv}(\mathsf{pred}_n),\mathbf{z}_n}(\mathcal{M}_{m+n}))$.

Symbolic heaps with bounded tree-width. Our goal is to develop a decision procedure for symbolic heaps with inductive definitions in the bounded tree-width fragment developed by Iosif et al. [10]. This fragment imposes three conditions on SIDs, which we assume for all SIDs Φ considered in the following:

1. *Progress:* Every rule *allocates* exactly one variable x, i.e. every rule contains exactly one points-to assertion $x \mapsto \mathbf{y}$.
2. *Connectivity:* Every predicate call $\mathsf{pred}(\mathbf{z})$ of a rule has a parameter $\mathbf{z}[i]$ that is referenced by the rule's allocated variable, i.e., $\mathbf{z}[i] \in \mathbf{y}$. Moreover, the i-th free variable of predicate pred must be allocated in all rules $\mathsf{pred} \Leftarrow \varphi$ of Φ.
3. *Establishment:* All existentially quantified variables are eventually allocated.

Assumptions. We make two further assumptions for all SIDs throughout this paper: (1) *Predicates are called with pairwise different parameters.* (2) *Unfolding predicates (iteratively substituting predicate calls* $\mathsf{pred}(\mathbf{y})$ *with the right-hand sides* $\varphi[\mathsf{fv}(\mathsf{pred})/\mathbf{y}]$ *of rules* $\mathsf{pred} \Leftarrow \varphi$*) always yields satisfiable symbolic heaps.* SIDs can be transformed automatically to satisfy (1) and (2) before applying our decision procedure (cf. [1,14]). The SIDs in Figs. 1 and 5 satisfy all assumptions.

4 Profiles: An Abstraction for Concrete Heap Graphs

Entailment problem. We present our approach for entailments $\mathsf{pred}_1(\mathbf{x}) \models_\Phi \mathsf{pred}_2(\mathbf{y})$ between predicate calls $\mathsf{pred}_1(\mathbf{x})$, and $\mathsf{pred}_2(\mathbf{y})$ of an SID Φ. We discuss the treatment of more general entailments at the end of Sect. 5. Formally, the entailment $\mathsf{pred}_1(\mathbf{x}) \models_\Phi \mathsf{pred}_2(\mathbf{y})$ holds iff for all concrete heap graphs $\mathcal{M}$, we have $\mathcal{M} \models_\Phi \mathsf{pred}_1(\mathbf{x})$ implies $\mathcal{M} \models_\Phi \mathsf{pred}_2(\mathbf{y})$.

Model reconstruction. Recall from Lemma 1 that $\mathcal{M} \models_\Phi \mathsf{pred}_1(\mathbf{x})$ can be interpreted as being able to construct $\mathcal{M}$ as a model of $\mathsf{pred}_1(\mathbf{x})$ using the rules of SID Φ and our operations on heap graphs introduced in Sect. 2. To prove the entailment $\mathsf{pred}_1(\mathbf{x}) \models_\Phi \mathsf{pred}_2(\mathbf{y})$, we then have to "reconstruct" any such $\mathcal{M}$ as a model of $\mathsf{pred}_2(\mathbf{y})$. Since infinitely many model reconstructions might be required—after all there might be infinitely many $\mathcal{M}$ with $\mathcal{M} \models_\Phi \mathsf{pred}_1(\mathbf{x})$—we now develop an abstraction of heap graphs such that finitely many abstract model reconstructions suffice to cover all models of $\mathsf{pred}_1(\mathbf{x})$.

Running example. To sharpen our intuition, we present the technical details of our abstraction together with a running example: Fig. 5 shows an SID Φ_{lists} specifying predicates for various singly-linked list segments. The predicate $\mathtt{sll}(x_1, x_2)$ specifies non-empty singly-linked list segments with head x_1 and tail x_2. Similarly, the predicates $\mathtt{odd}(x_1, x_2)$ and $\mathtt{even}(x_1, x_2)$ restrict such list segments to odd and even length, respectively. In the remainder of this and the next section, we will use our abstraction to show that the entailment $\mathtt{sll}(x_1, x_2) \models_{\Phi_{\text{lists}}} \mathtt{odd}(x_1, x_2)$ does *not* hold.

$$\begin{array}{l} \mathtt{sll}(x_1, x_2) \Leftarrow x_1 \mapsto x_2 \\ \mathtt{sll}(x_1, x_2) \Leftarrow \exists y\colon x_1 \mapsto y * \mathtt{sll}(y, x_2) \end{array} \qquad \begin{array}{l} \mathtt{odd}(x_1, x_2) \Leftarrow x_1 \mapsto x_2 \\ \mathtt{odd}(x_1, x_2) \Leftarrow \exists y\colon x_1 \mapsto y * \mathtt{even}(y, x_2) \\ \mathtt{even}(x_1, x_2) \Leftarrow \exists y\colon x_1 \mapsto y * \mathtt{odd}(y, x_2) \end{array}$$

Fig. 5. SIDs Φ_{sll} (left) and $\Phi_{\text{o/e}}$ (right) specifying singly-linked list segments with head x_1 and tail x_2. Moreover, we define $\Phi_{\text{lists}} = \Phi_{\text{sll}} \cup \Phi_{\text{o/e}}$.

4.1 Context Profiles as an Abstract Domain

Contexts. Our proposed abstraction is based on *contexts*. Intuitively, every context describes an extension of a concrete heap graph by predicate calls such that the resulting graph satisfies a fixed predicate call. Thus, contexts reveal what is missing in a concrete heap graph to reconstruct models of predicate calls.

Definition 2 (Context). *A triple* $\mathcal{C} = \langle \mathsf{V}, \mathsf{pred}(\mathbf{x}), \mathsf{calls} \rangle$ *is a* context *of a concrete heap graph* $\mathcal{M}$ *w.r.t. SID* Φ *if (1)* $\mathsf{V} = \mathsf{FV}_{\mathcal{M}}$*, (2)* $\langle \mathsf{Ptr}_{\mathcal{M}}, \mathbf{x}, \mathsf{calls} \rangle \models_\Phi \mathsf{pred}(\mathbf{x})$*, and (3) neither* $\mathbf{x}$ *nor* calls *contain auxiliary variables of* $\mathcal{M}$*. Moreover, we define the set of* free variables *of context* $\mathcal{C}$ *as* $\mathsf{fv}(\mathcal{C}) := \mathsf{V}$*. We call variables in* $\mathbf{x}$ *or* calls*, but not in* $\mathsf{fv}(\mathcal{C})$*, the* auxiliary variables *of* $\mathcal{C}$*.* △

Example 3. Figure 6 shows contexts for two concrete heap graphs $\mathcal{M}_{\text{odd}}$ and $\mathcal{M}_{\text{even}}$ of odd and even length (without dashes), respectively. The extension by calls from the contexts is illustrated by dashed lines. Intuitively, context $\mathcal{C}_1$ states that no extension of $\mathcal{M}_{\text{odd}}$ is needed to obtain a model of predicate $\texttt{odd}(x_1, x_2)$. Context $\mathcal{C}_2$ states that—in order to obtain an odd list segment from x_1 to a, where a is an additional free variable—we have to add an even list segment from x_2 to a. Similarly, we obtain an even list segment from x_1 to some fresh variable a by adding an odd list segment from x_2 to a. The interpretation of contexts $\mathcal{C}_4$, $\mathcal{C}_5$, and $\mathcal{C}_6$ of $\mathcal{M}_{\text{even}}$ is analogous. △

Contexts decompositions. A context of heap graph $\mathcal{M}$ stores the free variables of $\mathcal{M}$. These variables are important, because additional free variables might allow to split a heap graph into several smaller ones. For example, the additional free variable b in Fig. 4 (read from right to left) allows to decompose a list into two lists. Since our goal is to develop a compositional abstraction, we have to take contexts of decompositions of heap graphs into account. In general, these decompositions are relevant for entailment when considering more complicated SIDs, e.g., doubly-linked binary trees or trees with linked leaves. We thus have to compute decompositions $\mathcal{M}_1 \bullet \ldots \bullet \mathcal{M}_k$, $k \geq 1$, of a concrete heap graph $\mathcal{M}$ and then consider a context for each component.

Definition 3 (Context decomposition). *A* context decomposition *of a concrete heap graph $\mathcal{M}$ w.r.t. SID Φ is a set $\mathcal{E} = \{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$ such that $\mathcal{M} = \mathcal{M}_1 \bullet \ldots \bullet \mathcal{M}_k$, $k \geq 1$, is a decomposition of $\mathcal{M}$ and $\mathcal{C}_1, \ldots, \mathcal{C}_k$ are contexts of the concrete heap graphs $\mathcal{M}_1, \ldots, \mathcal{M}_k$ w.r.t. Φ, respectively. Moreover, we define the set of* free variables *of context decomposition $\mathcal{E}$ as $\mathsf{fv}(\mathcal{E}) := \bigcup_{\mathcal{C} \in \mathcal{E}} \mathsf{fv}(\mathcal{C})$.* △

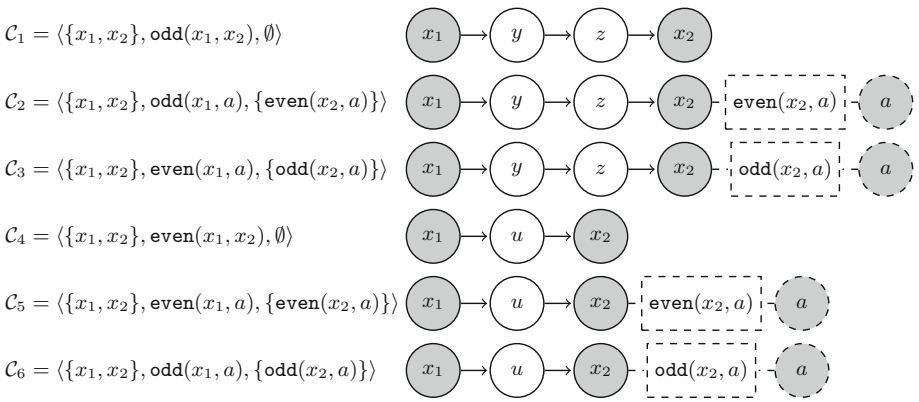

Fig. 6. Contexts of concrete heap graphs $\mathcal{M}_{\text{odd}}$ (first graph) and $\mathcal{M}_{\text{even}}$ (fourth graph). The extensions by a context are drawn in dashed lines.

Example 4. The concrete heap graph $\mathcal{M}_{\text{odd}}$ in Fig. 6 cannot be decomposed into smaller graphs due to a lack of free variables. Hence, context decompositions of $\mathcal{M}_{\text{odd}}$ are singletons consisting of $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ in Fig. 6, respectively. △

Profiles. As the above example shows, concrete heap graphs may have multiple context decompositions. We thus abstract a concrete heap graph $\mathcal{M}$ by the set of all context decompositions of $\mathcal{M}$:

Definition 4 (Profiles). *The* profile $\mathsf{profile}_\Phi(\mathcal{M})$ *of a concrete heap graph* $\mathcal{M}$ *w.r.t. SID* Φ *is the set of all context decompositions of* $\mathcal{M}$ *w.r.t.* Φ*. Moreover, since all* $\mathcal{E} \in \mathcal{P}$ *have the same free variables, we define the free variables of* $\mathcal{P}$ *as* $\mathsf{fv}(\mathcal{P}) := \mathsf{fv}(\mathcal{E})$ *for some* $\mathcal{E} \in \mathcal{P}$*.* △

Refinement property. We propose profiles as a suitable abstraction for deciding entailments. We will argue that they comply with the four essential correctness properties discussed in Sect. 1: refinement, finiteness, compositionality, and effectiveness. Refinement means that two concrete heap graphs with the same profiles entail the same SID predicates. Hence, for each profile and predicate pred, it suffices to find a single model of pred with that profile. Formally,

Lemma 2. *Let* $\mathcal{M}, \mathcal{M}'$ *be concrete heap graphs with* $\mathsf{profile}_\Phi(\mathcal{M}) = \mathsf{profile}_\Phi(\mathcal{M}')$*. Then, for all* $\mathsf{pred} \in \mathbf{Preds}(\Phi)$*, we have* $\mathcal{M} \models_\Phi \mathsf{pred}(\mathbf{x})$ *iff* $\mathcal{M}' \models_\Phi \mathsf{pred}(\mathbf{x})$*.*

Finiteness. In general, the set of profiles of concrete heap graphs is infinite due to different names for additional free variables, e.g., variable a in Fig. 6. To obtain a finite set of profiles, we thus (a) limit the total number of free variables, (b) consider profiles up to renaming of additional free variables, and (c) exploit the *connectivity condition*. Notice that condition (a) is not a restriction, because the number of free variables for every SID and thus every entailment query is bounded. For condition (b), we have to lift the notion of isomorphism from heap graphs to profiles. Formally, contexts $\mathcal{C}_1 = \langle \mathbf{z}_1, \mathsf{pred}_1(\mathbf{x}_1), \mathsf{calls}_1 \rangle$ and $\mathcal{C}_2 = \langle \mathbf{z}_2, \mathsf{pred}_2(\mathbf{x}_2), \mathsf{calls}_2 \rangle$ are isomorphic iff $\mathbf{z}_1 = \mathbf{z_2}$, $\mathsf{pred}_1 = \mathsf{pred}_2$ and there exists a bijective function $f\colon \mathbf{Var} \to \mathbf{Var}$ such that (1) for all $z \in \mathbf{z}_1$, $f(z) = z$, (2) $f(\mathbf{x}_1) = \mathbf{x_2}$, and (3) $\mathsf{calls}_2 = \{\mathsf{pred}(f(\mathbf{y})) \mid \mathsf{pred}(\mathbf{y}) \in \mathsf{calls}_1\}$. Moreover, two context decompositions $\mathcal{E}_1$, $\mathcal{E}_2$ are isomorphic iff for all $i \in \{1, 2\}$ and contexts $\mathcal{C} \in \mathcal{E}_i$ there is a context $\mathcal{C}' \in \mathcal{E}_{3-i}$ that is isomorphic to $\mathcal{C}$. Analogously, two profiles $\mathcal{P}_1$, $\mathcal{P}_2$ are isomorphic iff for all $i \in \{1, 2\}$ and context decompositions $\mathcal{E} \in \mathcal{P}_i$ there exists a context decomposition $\mathcal{E}' \in \mathcal{P}_{3-i}$ that is isomorphic to $\mathcal{E}_i$.

Throughout this paper, we do not distinguish between isomorphic contexts, context decompositions, or profiles.

Lemma 3. *For every SID* Φ *and variable sequence* $\mathbf{x} \in \mathbf{Var}^*$*, the set of profiles* $\mathbf{Profiles}^{\mathbf{x}}(\Phi) = \{\mathsf{profile}_\Phi(\mathcal{M}) \mid \mathcal{M}$ *concrete heap graph,* $\mathsf{fv}(\mathsf{profile}_\Phi(\mathcal{M})) \subseteq \mathbf{x}\}$ *is finite up to profile isomorphism.*

Example 5. Recall from Fig. 5 the SID $\Phi_{\mathrm{o/e}}$. Moreover, recall from Fig. 6 the concrete heap graphs $\mathcal{M}_{\mathrm{odd}}$ and $\mathcal{M}_{\mathrm{even}}$ and their contexts $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ and $\mathcal{C}_4$, $\mathcal{C}_5$, $\mathcal{C}_6$, respectively. Then the profiles of $\mathcal{M}_{\mathrm{odd}}$ and $\mathcal{M}_{\mathrm{even}}$ w.r.t. $\Phi_{\mathrm{o/e}}$ are (up to isomorphism) $\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{odd}}) = \{\{\mathcal{C}_1\}, \{\mathcal{C}_2\}, \{\mathcal{C}_2\}\}$ and $\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{even}}) = \{\{\mathcal{C}_4\}, \{\mathcal{C}_5\}, \{\mathcal{C}_6\}\}$. In fact, the profile of every singly-linked list segment from x_1

to x_2 of odd (even) length is isomorphic to $\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{odd}})$ ($\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{even}})$). Hence, the profile of every model of the singly-linked list predicate $\mathtt{sll}(x_1, x_2)$ is either $\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{odd}})$ or $\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{even}})$. △

4.2 Computation of Profiles

Due to Lemmas 2 and 3, we can decide an entailment $\mathsf{pred}_1(\mathbf{x}) \models_\Phi \mathsf{pred}_2(\mathbf{x})$, once the profiles of all models of $\mathsf{pred}_1(\mathbf{x})$ with respect to the rules relevant for $\mathsf{pred}_2(\mathbf{x})$ are known. The key insight underlying our entailment checker is that profiles can be computed automatically in a compositional manner. To this end, recall from Theorem 1 that every concrete heap graph can be constructed from single-allocation heap graphs $x \rightarrowtail \mathbf{y}$ by means of renaming, forgetting, and composition. We exploit this by (1) devising an algorithm to compute $\mathsf{profile}_\Phi(x \rightarrowtail \mathbf{y})$ and (2) lifting the operations $\mathsf{rename}_{\mathbf{x},\mathbf{y}}$, $\mathsf{forget}_{\mathbf{x}}$, and $\bullet$ for renaming, forgetting, and composition of heap graphs to operations $\overline{\mathsf{rename}}_{\mathbf{x},\mathbf{y}}$, $\overline{\mathsf{forget}}_{\mathbf{x}}$, and $\bar{\bullet}$ on profiles.

Profiles of single allocations. Since single allocations $x \rightarrowtail \mathbf{y}$ cannot be further decomposed, every context decomposition of $x \rightarrowtail \mathbf{y}$ w.r.t. an SID Φ is a singleton. Due to the progress condition, every rule of Φ contains exactly one points-to assertion. For each SID rule $\mathsf{pred} \Leftarrow \exists \mathbf{z}\colon x' \mapsto \mathbf{y}' * \mathsf{pred}_1(\mathbf{y}_1) * \cdots * \mathsf{pred}_k(\mathbf{y}_k)$, the corresponding context $\langle \{x'\} \cup \mathbf{y}', \mathsf{pred}(\mathbf{x}), \{\mathsf{pred}_1(\mathbf{y}_1), \ldots, \mathsf{pred}_k(\mathbf{y}_k)\}\rangle$ must be in the profile of $x \rightarrowtail \mathbf{y}$ iff $x \rightarrowtail \mathbf{y}$ is a model of $\exists \mathbf{z}\colon x' \mapsto \mathbf{y}'$. Hence:

Lemma 4. *Profiles of single allocations, i.e.,* $\mathsf{profile}_\Phi(x \rightarrowtail \mathbf{y})$*, are computable.*

Rename for profiles. We lift the operation $\mathsf{rename}_{\mathbf{x},\mathbf{y}}$, which renames each variable in $\mathbf{x}$ to the corresponding variable in $\mathbf{y}$ according to their position, from heap graphs to contexts, context decompositions, and profiles by componentwise application. That is, for a context $\mathcal{C} = \langle \mathbf{z}, \mathsf{pred}(\mathbf{u}), \mathsf{calls}\rangle$, a context decomposition $\mathcal{E}$, and a profile $\mathcal{P}$, we define:

$$\begin{aligned}
\mathsf{rename}_{\mathbf{x},\mathbf{y}}(\mathcal{C}) &:= \langle \mathsf{rename}_{\mathbf{x},\mathbf{y}}(\mathbf{z}), \mathsf{pred}(\mathsf{rename}_{\mathbf{x},\mathbf{y}}(\mathbf{u})), \\
&\qquad \{\mathsf{pred}'(\mathsf{rename}_{\mathbf{x},\mathbf{y}}(\mathbf{v}) \mid \mathsf{pred}'(\mathbf{v}) \in \mathsf{calls}\}\rangle \\
\mathsf{rename}_{\mathbf{x},\mathbf{y}}(\mathcal{E}) &:= \{\mathsf{rename}_{\mathbf{x},\mathbf{y}}(\mathcal{C}) \mid \mathcal{C} \in \mathcal{E}\} \\
\overline{\mathsf{rename}}_{\mathbf{x},\mathbf{y}}(\mathcal{P}) &:= \{\mathsf{rename}_{\mathbf{x},\mathbf{y}}(\mathcal{E}) \mid \mathcal{E} \in \mathcal{P}\}
\end{aligned}$$

Forget for profiles. Next, we lift the operation $\mathsf{forget}_{\mathbf{x}}$, which removes variables in $\mathbf{x}$ from the set of free variables, to contexts, context decompositions, and profiles. For a profile, forgetting a free variable means that some of its constituting context decompositions do not have to be considered anymore, because the composition of their underlying models is no longer defined. Hence, these decompositions are removed. Formally, for a context $\mathcal{C} = \langle \mathbf{z}, \mathsf{pred}(\mathbf{u}), \mathsf{calls}\rangle$, a context decomposition $\mathcal{E}$, and a profile $\mathcal{P}$, we define:

$$\mathsf{forget}_{\mathbf{x}}(\mathcal{C}) := \langle \mathbf{z} \setminus \mathbf{x}, \mathsf{pred}(\mathbf{u}), \mathsf{calls} \rangle \qquad \mathsf{forget}_{\mathbf{x}}(\mathcal{E}) := \{\mathsf{forget}_{\mathbf{x}}(\mathcal{C}) \mid \mathcal{C} \in \mathcal{E}\}$$
$$\overline{\mathsf{forget}}_{\mathbf{x}}(\mathcal{P}) := \{\mathsf{forget}_{\mathbf{x}}(\mathcal{E}) \mid \mathcal{E} \in \mathcal{P} \text{ and } \mathbf{x} \cap \mathsf{usedvs}(\mathcal{E}) = \emptyset\}$$
$$\mathsf{usedvs}(\mathcal{E}) := \bigcup_{\mathcal{C} \in \mathcal{E}} \mathsf{usedvs}(\mathcal{C}) \qquad \mathsf{usedvs}(\mathcal{C}) := \mathbf{u} \cup \bigcup_{\mathsf{pred}'(\mathbf{y}) \in \mathsf{calls}} \mathbf{y}$$

Composition for profiles. It remains to lift heap graph composition to profiles. This is formalized as substituting predicate calls of contexts by other contexts:

Definition 5 (Context substitution). *Let* $\mathcal{C}_1 = \langle \mathbf{x}_1, \mathsf{pred}_1(\mathbf{z}_1), \mathsf{calls}_1 \rangle$ *and* $\mathcal{C}_2 = \langle \mathbf{x}_2, \mathsf{pred}_2(\mathbf{z}_2), \mathsf{calls}_2 \rangle$ *be contexts such that (1)* $\mathsf{pred}_1(\mathbf{z}_1) \in \mathsf{calls}_2$ *and (2) no auxiliary variable of* $\mathcal{C}_2$ *is a free variable of* $\mathcal{C}_1$ *and vice versa. Then the* substitution *of* $\mathsf{pred}_1(\mathbf{z})$ *in* $\mathcal{C}_2$ *by* $\mathcal{C}_1$ *is given by*

$$\mathcal{C}_2\,[\mathcal{C}_1] := \langle\, \mathbf{x_1} \cup \mathbf{x_2},\, \mathsf{pred}_2(\mathbf{z_2}),\, (\mathsf{calls}_2 \setminus \{\mathsf{pred}_1(\mathbf{z}_1)\}) \cup \mathsf{calls}_1 \rangle. \qquad \triangle$$

To compose profiles, we attempt to substitute the underlying contexts with each other in all possible ways. Formally, a context decomposition $\mathcal{E}_1$ *derives* a context decomposition $\mathcal{E}_2$, written $\mathcal{E}_1 \triangleright \mathcal{E}_2$, iff there exist contexts $\mathcal{C}_1, \mathcal{C}_2 \in \mathcal{E}_1$ such that $\mathcal{E}_2 = (\mathcal{E}_1 \setminus \{\mathcal{C}_1, \mathcal{C}_2\}) \cup \{\mathcal{C}_2\,[\mathcal{C}_1]\}$.[3] We denote by $\triangleright^*$ the reflexive-transitive closure of the derivation relation $\triangleright$. The composition of two profiles then consists of all context decompositions derivable from some decompositions of both profiles:

Definition 6 (Composition of profiles). *Let* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *be profiles w.r.t.* Φ. *Then the* composition $\mathcal{P}_1 \mathbin{\overline{\bullet}} \mathcal{P}_2$ *of* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *is defined as*

$$\mathcal{P}_1 \mathbin{\overline{\bullet}} \mathcal{P}_2 := \{\mathcal{E} \mid \exists \mathcal{E}_1 \in \mathcal{P}_1, \mathcal{E}_2 \in \mathcal{P}_2 \colon \mathcal{E}_1 \cup \mathcal{E}_2 \triangleright^* \mathcal{E}\}. \qquad \triangle$$

Compositionality. Our lifted heap graph operations satisfy the compositionality property mentioned in Sect. 1. That is,

Theorem 2. *For all concrete heap graphs* $\mathcal{M}$, $\mathcal{M}'$ *and every SID* Φ, *we have*

$$\begin{aligned}
\overline{\mathsf{rename}}_{\mathbf{x},\mathbf{y}}(\mathsf{profile}_\Phi(\mathcal{M})) &= \mathsf{profile}_\Phi(\mathsf{rename}_{\mathbf{x},\mathbf{y}}(\mathcal{M})) \\
\overline{\mathsf{forget}}_{\mathbf{x}}(\mathsf{profile}_\Phi(\mathcal{M})) &= \mathsf{profile}_\Phi(\mathsf{forget}_{\mathbf{x}}(\mathcal{M})) \\
\mathsf{profile}_\Phi(\mathcal{M}) \mathbin{\overline{\bullet}} \mathsf{profile}_\Phi(\mathcal{M}') &= \mathsf{profile}_\Phi(\mathcal{M} \bullet \mathcal{M}')
\end{aligned}$$

provided that $\mathsf{rename}_{\mathbf{x},\mathbf{y}}(\mathcal{M})$, $\mathsf{forget}_{\mathbf{x}}(\mathcal{M})$, *and* $\mathcal{M} \bullet \mathcal{M}'$ *are defined, respectively.*

Example 6. Recall from Fig. 6 the heap graphs $\mathcal{M}_{\text{odd}}$ and $\mathcal{M}_{\text{even}}$ whose profiles w.r.t. $\Phi_{\text{o/e}}$ capture all singly-linked lists. We can construct a concrete heap graph $\mathcal{M}$ representing a list of length five from x_1 to x_2 by computing

$$\mathcal{M} := \mathsf{rename}_{v,x_2}\left(\mathsf{forget}_{x_2}\left(\mathcal{M}_{\text{odd}} \bullet \mathsf{rename}_{(x_1,x_2),(x_2,v)}(\mathcal{M}_{\text{even}})\right)\right).$$

[3] Recall that all definitions are to be read up to isomorphism, i.e., auxiliary variables of $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{E}_2$ may be renamed prior to the substitution.

Then, by Theorem 2, the corresponding profile $\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(M)$ is given by:

$$\overline{\mathsf{rename}}_{v,x_2}\Big(\overline{\mathsf{forget}}_{x_2}\Big(\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{odd}}) \mathbin{\overline{\bullet}} \overline{\mathsf{rename}}_{(x_1,x_2),(x_2,v)}(\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{even}}))\Big)\Big)$$

This profile, in turn, coincides with the profile of $\mathcal{M}_{\mathrm{odd}}$, i.e., we have

$$\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}) \;=\; \mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{odd}}).$$

In particular, notice that without the forget statement, we would obtain a heap graph $\mathcal{M}'$ with an additional free variable. The additional free variable would also influence the profile of $\mathcal{M}'$, because there exist more decompositions of $\mathcal{M}'$ into heap graphs $\mathcal{M}_1 \bullet \mathcal{M}_2$. Consequently, there are also more context decompositions of $\mathcal{M}'$ and thus $\mathcal{M}'$ has a larger profile. △

5 An Effective Decision Procedure for Entailment

Profile analysis. We now exploit our abstract domain to develop a decision procedure for entailments of the form $\mathsf{pred}_1(\mathbf{a}) \models_\Phi \mathsf{pred}_2(\mathbf{b})$. Let us first consider the case in which the parameters $\mathbf{a}$ and $\mathbf{b}$ coincide with the free variables in the rules of the SID, i.e., $\mathbf{a} = \mathsf{fv}(\mathsf{pred}_1) =: \mathbf{x_1}$ and $\mathbf{b} = \mathsf{fv}(\mathsf{pred}_2) =: \mathbf{x_2}$. Our key observation is then that analyzing profiles of the entailment's left-hand side suffices to discharge it: The entailment $\mathsf{pred}_1(\mathbf{x_1}) \models_\Phi \mathsf{pred}_2(\mathbf{x_2})$ holds iff the profile of every model $\mathcal{M}$ of $\mathsf{pred}_1(\mathbf{x_1})$ contains a context decomposition stating that a model of $\mathsf{pred}_2(\mathbf{x_2})$ can be reconstructed from $\mathcal{M}$. Formally,

Theorem 3. *The entailment* $\mathsf{pred}_1(\mathbf{x_1}) \models_\Phi \mathsf{pred}_2(\mathbf{x_2})$ *holds iff for all concrete heap graphs* $\mathcal{M}$ *with* $\mathcal{M} \models \mathsf{pred}_1(\mathbf{x_1})$, $\{\langle \mathsf{FV}_{\mathcal{M}}, \mathsf{pred}_2(\mathbf{x_2}), \emptyset\rangle\} \in \mathsf{profile}_\Phi(\mathcal{M})$.

Example 7. Recall the profiles $\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{even}})$ and $\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{odd}})$ from Example 5 computed for models of $\mathtt{sll}(x_1, x_2)$ w.r.t. SID $\Phi_{\mathrm{o/e}}$ (Fig. 5). We now use these profiles to disprove the entailment $\mathtt{sll}(x_1, x_2) \models_{\Phi_{\mathrm{lists}}} \mathtt{odd}(x_1, x_2)$: First, observe that all predicates relevant for constructing models of $\mathtt{odd}(x_1, x_2)$ belong to $\Phi_{\mathrm{o/e}} \subseteq \Phi_{\mathrm{lists}}$. Second, the $\mathsf{profile}_{\Phi_{\mathrm{o/e}}}(\mathcal{M}_{\mathrm{even}})$ does not contain a context decomposition $\{\langle\{x_1, x_2\}, \mathtt{odd}(x_1, x_2), \emptyset\rangle\}$. Hence, by Theorem 3, the entailment does not hold as we cannot reconstruct $\mathcal{M}_{\mathrm{even}}$ as a model of predicate $\mathtt{odd}(x_1, x_2)$. △

Computing profiles. By Theorem 3, to decide whether $\mathsf{pred}_1(\mathbf{x_1}) \models_\Phi \mathsf{pred}_2(\mathbf{x_2})$ holds, it suffices to compute the finite (by Lemma 3) set of all profiles of models of $\mathsf{pred}_1(\mathbf{x_1})$. This is performed by the procedure $\mathsf{abstractSID}(\Phi)$ shown in Algorithm 1. To understand how the algorithm works, recall how predicates can be unrolled to compute a model: We select an SID rule and replace all of its predicate calls with previously computed models. By Lemma 1, this amounts to performing heap graph operations. That is, we first rename the free variables of previously computed models to match the parameters of predicate calls. After

Algorithm 1: The algorithm $\mathsf{abstractSID}(\Phi)$ computes a function f that maps each predicate $\mathsf{pred} \in \mathbf{Preds}(\Phi)$ to the set of profiles $\{\mathsf{profile}_\Phi(\mathcal{M}) \mid \mathcal{M} \models_\Phi \mathsf{pred}(\mathsf{fv}(\mathsf{pred}))\}$.

```
1  f_curr := λpred . ∅;
2  repeat
3  |   f_prev := f_curr;
4  |   for pred ∈ Preds(Φ) do
5  |   |   for (pred ⇐ ∃y: x ↦ z_0 * pred_1(z_1) * ··· * pred_k(z_k)) ∈ Rules(Φ) do
6  |   |   |   P_0 := profile_Φ(x ⇀ z_0);
7  |   |   |   for F_1 ∈ f_prev(pred_1), ..., F_k ∈ f_prev(pred_k) do
8  |   |   |   |   for i ∈ {1, ..., k} do
9  |   |   |   |   |   P_i := rename_{fv(pred_i),z_i}(F_i);
10 |   |   |   |   P := forget_y(P_0 ∗ P_1 ∗ ··· ∗ P_k);
11 |   |   |   |   f_curr(pred) := f_curr(pred) ∪ {P};
12 until f_curr = f_prev;
13 return f_curr
```

that, the resulting models and the single allocation (due to the progress condition) of the rule are composed into a single heap graph. Finally, we apply a forget operation to remove free variables that have been existentially quantified.

Algorithm 1 behaves analogously. However, instead of applying operations on heap graphs, it applies our *abstract* operations on profiles (cf. Theorem 2): We select an SID rule $\mathsf{pred} \Leftarrow \varphi$ in line 5. By Lemma 4, we can compute the profile of the single allocation in φ. (l. 6). We then select previously computed profiles for the predicate rules and rename their free variables to match the parameters of the predicate calls in φ (l. 7–9). Finally, the selected profiles are composed and added to the computed profiles of predicate pred (l. 10, 11). The algorithm then proceeds by computing profiles until a fixed point is reached (l. 12).

Correctness. Algorithm 1 is guaranteed to terminate due to the finiteness of our abstract domain (Lemma 3). Moreover, it computes the desired set of profiles:

Theorem 4. $\mathsf{abstractSID}(\Phi)(\mathsf{pred}) = \{\mathsf{profile}_\Phi(\mathcal{M}) \mid \mathcal{M} \models_\Phi \mathsf{pred}(\mathsf{fv}(\mathsf{pred}))$ *and* $\mathsf{FV}_\mathcal{M} \subseteq \mathsf{fv}(\mathsf{pred})\}$.

To check entailments $\mathsf{pred}_1(\mathbf{a}) \models_\Phi \mathsf{pred}_2(\mathbf{b})$, where $\mathbf{a}$ and $\mathbf{b}$ do not coincide with the free variables of pred_1 and pred_2 in the rules of Φ, it suffices to apply an additional rename operation. Hence, by combining Theorems 3 and 4, we obtain a constructive decidability proof for entailments between predicate calls. Moreover, a close inspection of the size of the set of profiles and the runtime of Algorithm 1 reveals that our decision procedure runs in time doubly exponential in the size of a given SID. A detailed analysis is found in [1, Sect. 7.4].

Corollary 1. *It is decidable in doubly exponential time whether the entailment* $\mathsf{pred}_1(\mathbf{a}) \models_\Phi \mathsf{pred}_2(\mathbf{b})$ *holds.*

Generalizations. Several of our assumptions about SIDs and entailments have been made purely to simplify the presentation. In fact, Corollary 1 can be generalized to (1) decide entailments $\varphi \models_\Phi \psi$ for symbolic heaps φ, ψ (instead of predicate calls) and (2) SIDs with pure formulas. Both extensions are supported by our implementation. Further details are found in [1].

6 Experiments

We implemented our decision procedure for entailment in the separation logic prover HARRSH [1,15], which is written in Scala. HARRSH supports the full $\mathrm{SL}_{\mathrm{btw}}$ fragment, including pure formulas, parameter repetitions, and entailments between symbolic heaps (as opposed to single predicate calls). Table 1 summarizes the results of our evaluation for a selection of entailments and SIDs. Our full collection of 101 benchmarks and all experimental results are available online [1].

Methodology. We compared HARRSH against SONGBIRD [19], the winner of the SID entailment category of this year's separation logic competition, SL-COMP'18; and against SLIDE [11], the tool that is most closely related to our approach but that is complete only for a subclass of $\mathrm{SL}_{\mathrm{btw}}$. Experiments were conducted using the popular benchmarking harness JMH on an Intel® Core™ i7-7500U CPU running at 2.70 GHz with a memory limit of 4 GB. We report the average run times obtained by running JMH on each benchmark for 100 s.

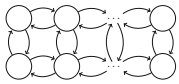

Fig. 7. dlgrid

Benchmarks. Besides the running example (with `sll`, `even` and `odd` as in Fig. 5) and the entailments for doubly-linked trees discussed in the introduction (with `ltree`, `rtree` as defined in Fig. 1), we show results on standard data-structure specifications from the SL literature: Several variants of trees with linked leaves (`tll` [10], `atll`, $\texttt{tll}^{lin}$) and doubly-linked lists (`dllht` [18] defining lists from head to tail, `dllth` from tail to head). Beyond lists and trees, we checked an entailment between *doubly-linked* 2*-grid segments* (see Fig. 7) defined forwards `dlgridr` and backwards `dlgridl`.[4]

Size of the abstraction. Beside the run times, we report the size of the abstraction computed by HARRSH. More specifically, we report (1) the total number of profiles in the fixed point of abstractSID (#P), (2) the total number of context decompositions across all profiles (#D), and (3) the total number of contexts across all decompositions of all profiles (#C). This shows that even though the abstract domain $\mathbf{Profiles}^{\mathbf{x}}(\Phi)$ is very large in general, HARRSH typically only needs to explore a small portion of it to decide an entailment.

[4] Formal definitions of all SIDs are found in the supplementary material [1].

Table 1. The performance of HARRSH (HRS), SONGBIRD (SB) and SLIDE (SLD) on a variety of SIDs; and the size of the abstraction computed by HARRSH. The timeout (TO) was 180,000 ms. Termination before the timeout but without result is denoted (U). Wrong results/crashes are marked (X).

Benchmark		Time (ms)			Profiles		
Query	Status	HRS	SB	SLD	#P	#D	#C
$\texttt{sll}(x_1, x_2) \models \texttt{odd}(x_1, x_2)$	false	4	11	43	2	6	6
$\texttt{even}(x_1, x_2) \models \texttt{sll}(x_1, x_2)$	true	2	26	43	2	4	4
$\texttt{rtree}(x_1, x_2, x_3) \models \texttt{ltree}(x_1, x_2, x_3)$	false	16	(U)	53	3	14	21
Entailment (♣) (Sect. 1), left to right	true	393	TO	53	7	70	116
Entailment (♣) (Sect. 1), right to left	true	532	1274	54	9	57	87
$\texttt{atll}(x_1, x_2, x_3) \models \texttt{tll}(x_1, x_2, x_3)$	true	9	8519	TO	2	2	2
$\texttt{tll}(x_1, x_2, x_3) \models \texttt{atll}(x_1, x_2, x_3)$	false	2	119	TO	2	1	1
$\texttt{tll}^{lin}(x_1, x_2, x_3) \models \texttt{tll}(x_1, x_2, x_3)$	true	2	34	(X)	3	3	4
$\texttt{dllht}(x_1, x_2, x_3, x_4) \models \texttt{dllth}(x_3, x_4, x_1, x_2)$	true	16	37	50	3	27	45
$\texttt{dllth}(x_1, x_2, x_3, x_4) \models \texttt{dllht}(x_3, x_4, x_1, x_2)$	true	16	37	50	3	27	45
$\texttt{dlgridr}(x_1, \ldots, x_8) \models \texttt{dlgridl}(x_1, \ldots, x_8)$	true	172	TO	(X)	5	87	208

Results. Table 1 reveals that our decision procedure—being the first implemented decision procedure that is complete for the entire SL fragment $\mathrm{SL}_{\mathrm{btw}}$—is not only of theoretical interest, but can also solve challenging entailment problems efficiently in practice. While SLIDE was faster on some benchmarks that fall into the fragment defined in [11], as well as on some SIDs outside of that fragment, HARRSH was able to solve several benchmarks on which SLIDE failed. Two benchmarks led to errors: One wrong result and one program crash (the first and the second entries marked by (X) in Table 1, respectively). We are unsure whether the timeouts encountered on the TLL benchmarks are caused by a bug in SLIDE, as SLIDE is quite efficient on other TLL variants (see [11, Table 1]). Furthermore, note that HARRSH significantly outperformed SONGBIRD, providing further evidence of the effectiveness of our profile-based abstraction.

7 Conclusion

We presented an alternative proof for decidability of entailment in separation logic with bounded tree width [10]. In contrast to the original proof, we give a direct model theoretic construction. We implemented the resulting decision procedure in the tool HARRSH and obtained promising experimental results. For future work, we plan to extend our approach to the bi-abduction problem.

References

1. Supplementary material. The webpage below provides access to proofs, our tool, its source code, and our benchmarks. https://github.com/katelaan/harrsh
2. Antonopoulos, T., Gorogiannis, N., Haase, C., Kanovich, M.I., Ouaknine, J.: Foundations for decision problems in separation logic with general inductive predicates. In: Muscholl, A. (ed.) FoSSaCS 2014. LNCS, vol. 8412, pp. 411–425. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54830-7_27
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30538-5_9
4. Brotherston, J.: Formalised inductive reasoning in the logic of bunched implications. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 87–103. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_6
5. Brotherston, J., Distefano, D., Petersen, R.L.: Automated cyclic entailment proofs in separation logic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 131–146. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_12
6. Calcagno, C., Distefano, D.: Infer: an automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 459–465. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_33
7. 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)
8. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach, Encyclopedia of Mathematics and Its Applications, vol. 138. Cambridge University Press, Cambridge (2012)
9. Enea, C., Lengál, O., Sighireanu, M., Vojnar, T.: SPEN: a solver for separation logic. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 302–309. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_22
10. Iosif, R., Rogalewicz, A., Simácek, J.: The tree width of separation logic with recursive definitions. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 21–38. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_2
11. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding entailments in inductive separation logic with tree automata. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 201–218. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_15
12. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, 17–19 January 2001, pp. 14–26 (2001)
13. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4

14. Jansen, C., Katelaan, J., Matheja, C., Noll, T., Zuleger, F.: Unified reasoning about robustness properties of symbolic-heap separation logic. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 611–638. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_23
15. Katelaan, J., Matheja, C., Noll, T., Zuleger, F.: Harrsh: a tool for unified reasoning about symbolic-heap separation logic. In: Proceedings of the 13th International Workshop on the Implementation of Logics (IWIL) (2018)
16. Pek, E., Qiu, X., Madhusudan, P.: Natural proofs for data structure manipulation in C using separation logic. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom, 09–11 June 2014, pp. 440–451 (2014)
17. Piskac, R., Wies, T., Zufferey, D.: GRASShopper - complete heap verification with mixed specifications. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 124–139. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_9
18. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings 17th IEEE Symposium on Logic in Computer Science (LICS 2002), Copenhagen, Denmark, 22–25 July 2002, pp. 55–74 (2002)
19. Ta, Q., Le, T.C., Khoo, S., Chin, W.: Automated mutual explicit induction proof in separation logic. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 659–676. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_40
20. Ta, Q., Le, T.C., Khoo, S., Chin, W.: Automated lemma synthesis in symbolic-heap separation logic. PACMPL **2**(POPL), 9:1–9:29 (2018)