Form Methods Syst Des manuscript No.
(will be inserted by the editor)

Compositional Entailment Checking for a Fragment of Separation Logic

Constantin Enea · Ondřej Lengál · Mihaela Sighireanu · Tomáš Vojnar

Received: September 3, 2017/ Accepted: date

Abstract We present a decision procedure for checking entailment between separation logic formulas with inductive predicates specifying complex data structures corresponding to finite nesting of various kinds of singly linked lists: acyclic or cyclic, nested lists, skip lists, etc. The decision procedure is compositional in the sense that it reduces the problem of checking entailment between two arbitrary formulas to the problem of checking entailment between a formula and an atom. Subsequently, in case the atom is a predicate, we reduce the entailment to testing membership of a tree derived from the formula in the language of a tree automaton derived from the predicate. The procedure is later also extended to doubly linked lists. We implemented this decision procedure and tested it successfully on verification conditions obtained from programs using both singly and doubly linked nested lists as well as skip lists.

Keywords program analysis · separation logic · decision procedure · tree automata

1 Introduction

Automatic verification of programs manipulating dynamic linked data structures is highly challenging since it requires one to reason about complex program configurations having the form of graphs of an unbounded size. For that, a highly expressive formalism is needed. Moreover, in order to scale to large programs, the use of such a formalism within program analysis should be highly efficient. In this context, *separation logic* (SL) [13,18] has emerged as one of the most promising formalisms, offering both high expressiveness and scalability. The latter is due to its support of *compositional reasoning* based on the separating conjunction $*$ and the frame rule, which states that if a Hoare triple $\{\phi\}P\{\psi\}$ holds and P does not alter free variables in σ, then $\{\phi * \sigma\}P\{\psi * \sigma\}$ holds too. Therefore, when reasoning about P, one has to manipulate only specifications for the heap region altered by P.

C. Enea and M. Sighireanu
IRIF, University Paris Diderot and CNRS, 8 place Aurélie Nemours, F-75013 Paris, France
E-mail: {sighirea,cenea}@liafa.univ-paris-diderot.fr

O. Lengál and T. Vojnar
FIT, Brno University of Technology, IT4I Centre of Excellence, Božetěchova 2, 61266 Brno, Czech Rep.
E-mail: {lengal,vojnar}@fit.vutbr.cz

Usually, SL is combined together with higher-order *inductive definitions* that specify the data structures manipulated by the program. If we consider general inductive definitions, then SL is undecidable [5]. Various decidable fragments of SL have been introduced in the literature [1,11,16,3] by restricting the syntax of the inductive definitions and the Boolean structure of the formulas.

In this work, we focus on a fragment of SL with inductive definitions that allows one to specify program configurations (heaps) containing finite nestings of various kinds of singly linked lists (acyclic, cyclic, skip lists, etc.) that are common in practice. (In Section 8, we show that the procedure can also be generalised to doubly linked lists.) This fragment contains formulas of the form $\exists \vec{X} : \Pi \wedge \Sigma$ where $\vec{X}$ is a set of variables, Π is a conjunction of (dis)equalities, and Σ is a set of *spatial atoms* connected by the separating conjunction. Spatial atoms can be *points-to atoms*, which describe values of pointer fields of a given heap location, or *inductively defined predicates*, which describe data structures of an unbounded size. We propose a novel decision procedure for checking the validity of entailments of the form $\varphi \Rightarrow \psi$ where φ may contain existential quantifiers and ψ is a quantifier-free formula. Such a decision procedure can be used in Hoare-style reasoning to check inductive invariants but also in program analysis frameworks to decide termination of fixpoint computations. As usual, checking entailments of the form $\bigvee_i \varphi_i \Rightarrow \bigvee_j \psi_j$ can be soundly reduced to checking that for each i there exists j such that $\varphi_i \Rightarrow \psi_j$.

The key insight of our decision procedure is the idea to use the semantics of the separating conjunction in order to reduce the problem of checking $\varphi \Rightarrow \psi$ to the problem of checking a set of simpler entailments where the right-hand side is an inductively-defined predicate $P(\ldots)$. This reduction shows that the compositionality principle holds not only for deciding the validity of Hoare triples but also for deciding the validity of entailments between two formulas. The reduction requires one to infer (dis)equalities implied by spatial atoms. This inference is done by deriving from the initial SL formulas equi-satisfiable Boolean formulas and checking their unsatisfiability. Boolean unsatisfiability checking is **co-NP** complete, but efficient decision procedures are implemented in the existing SAT solvers.

Further, to check entailments $\varphi \Rightarrow P(\ldots)$ resulting from the above reduction, we define a decision procedure based on the membership problem for tree automata (TAs). In particular, we reduce the above simple entailment to testing membership of a tree derived from φ in the language of a TA $\mathcal{A}[P]$ derived from $P(\ldots)$. The tree encoding of φ preserves some edges of the Gaifman graph of φ, called *backbone edges*, and re-directs other to new nodes, related to the original destination by special symbols. Roughly, such a symbol may be a variable labelling the original destination, or it may show how to reach the original destination using backbone edges only.

Our decision procedure is sound and complete for the considered fragment (defined formally in Section 2). Its time complexity is polynomial in the size of the formula, modulo an oracle for deciding validity of Boolean formulas. We also implemented the procedure and tested it successfully on verification conditions obtained from programs using singly (and doubly) linked nested lists as well as skip lists. The results show that our procedure does not only have a theoretically favorable complexity (for the given context), but also behaves nicely in practice (our implementation in the tool SPEN won one gold and two silver medals in the first competition of SL solvers SL-COMP'14 [19]). At the same time, the procedure offers the addi-

tional benefit of compositionality, which can be exploited within larger verification frameworks caching the simpler entailment queries.

Related Work. Several decision procedures for fragments of SL have been introduced in the literature [1,5,6,9,11,12,15,16,4]. Some of these works [1,5,6,15] consider a fragment of SL that uses only a single predicate describing singly linked lists, which is a much more restricted setting than what is considered in this work. In particular, Cook *et al.* [6] prove that the satisfiability/entailment problem can be solved in polynomial time. Piskac *et al.* [16] show that the Boolean closure of this fragment can be translated to a decidable fragment of first-order logic, and in this way they prove that the satisfiability/entailment problem can be decided in **NP/co-NP**. Furthermore, they consider the problem of combining SL formulas with constraints on data using the Nelson-Oppen theory combination framework. Adding constraints on data to SL formulas is considered also in Qiu *et al* [17].

A fragment of SL covering overlaid nested lists was considered by Enea *et al* [9]. Compared with it, we currently do not consider overlaid lists, but we have enlarged the set of inductively-defined predicates to allow nesting of cyclic lists and doubly linked lists (DLLs). We also provide a novel and more efficient TA-based procedure for checking simple entailments.

Brotherston *et al.* [4] define a generic automated theorem prover relying on the notion of cyclic proofs and instantiate it to prove entailments in a fragment of SL with inductive definitions and disjunctions more general than what we consider here. They do not, however, provide a fragment for which completeness is guaranteed. Another incomplete procedure based on proof searching is [10]. It considers inductive definitions specifying nested lists and trees storing integer data, but no circular nested lists. The proof search applies lemmas satisfied by the inductive definitions. These lemmas can be generated automatically due to some syntactic restrictions on inductive definitions. Our fragment allows to define more general shapes for nested lists. Iosif *et al.* [11] also introduce a decidable fragment of SL that can describe more complex data structures than the fragment presented in this work, including, e.g., trees with parent pointers or trees with linked leaves. The mentioned work reduces the entailment problem to MSO on graphs with a bounded tree width, resulting in a multiple-exponential complexity.

The work [12] considers a more restricted fragment than [11] (incomparable with ours). The work proposes a more practical, purely TA-based decision procedure, which reduces the entailment problem to *language inclusion* on TAs, establishing **EXPTIME** completeness of the considered fragment. Our decision procedure deals with the Boolean structure of SL formulas using SAT solvers, thus reducing the entailment problem to the problem of entailment between a formula and an atom. Such simpler entailments are then checked using a polynomial decision procedure based on the *membership problem* for TAs. The approach of [12] can deal with various forms of trees and with entailment of structures with skeletons based on different selectors (e.g. DLLs viewed from the beginning and linked by the `next` selector as the main field, and DLLs viewed from the end linked by the `prev` selector as the main field). On the other hand, the procedure in [12] currently cannot deal with structures of zero length and with some forms of structure concatenation (such as concatenation of two DLL segments), which we can handle.

The recent work [3] has proposed a reduction of SL formula satisfiability checking to Boolean satisfiability. As discussed above, our procedure also uses a reduction to Boolean satisfiability to infer (dis)equalities in the SL formulas. The inductive definitions we consider are, however, less general, and we use a different reduction to check satisfiability as a part of our decision procedure for entailment checking.

This work is an extended version of [7]. We consider a slightly smaller class of inductive definitions to obtain a simpler presentation. This simplification, to the best of our knowledge, does not, however, remove any inductive definition used to model the heap of data structures with practical interest. We also give an improved construction for the complete decision procedure, decreasing its time complexity from exponential to polynomial. Moreover, we also optimised our implementation and provide an updated evaluation of experiments.

Contribution. Overall, the contribution of this work is a novel decision procedure for a rich class of verification conditions with singly (extended also to doubly) linked lists, nested lists, and skip lists. As discussed in more detail in the previous paragraph, existing works that can efficiently deal with fragments of SL capable of expressing verification conditions for programs handling complex dynamic data structures are still rare. Indeed, we are not aware of any techniques that could decide the class of verification conditions considered in this work at the same level of efficiency as our procedure. In particular, compared with other approaches using TAs [11, 12], our procedure is compositional as it uses TAs recognising models of predicates, not models of entire formulas. Moreover, our TAs recognise in fact formulas that entail a given predicate, reducing SL entailment to the (**PTIME**) membership problem for TAs, not the more expensive (**EXPTIME** complete) inclusion problem as in other works.

2 Separation Logic Fragment

Our logic is a fragment of the *symbolic heaps fragment* [1] of Separation Logic [18]. The fragment specifies sets of configurations of programs manipulating the heap. A program configuration is given by the state of its stack and of its heap. We consider a memory model where the heap is abstracted by a collection of disjoint memory regions, called *records*. We denote by $Locs$ the set of locations at which heap records are stored. Records are sets of *fields*, each field storing a reference to a record location. The record types are fixed by type definitions that also define $\mathbb{F}$, the set of *field names*. Wlog, we assume that different record types declare pairwise disjoint sets of field names. A program manipulates the heap by creating records, setting and accessing their fields, and freeing them. For this, it uses a set of of *program variables* $Vars$ stored on the program stack. We assume that $Vars$ contains the `null` constant.

2.1 Syntax

The syntax of the Separation Logic fragment we consider is given in Fig. 1. Record locations that are not stored in program variables are addressed using a set of *logical variables* $LVars$ disjoint from $Vars$.

An SL formula is an existentially quantified conjunction of a pure formula Π and a spatial formula Σ. Wlog, we assume that existentially quantified logical variables

$x, y \in \mathit{Vars}$	program variables	$f \in \mathbb{F}$	fields
$X, Y \in \mathit{LVars}$	logical variables	$P \in \mathbb{P}$	predicates
$\vec{F} \in (\mathit{Vars} \cup \mathit{LVars})^*$	vectors of variables	$E, F ::= x \mid X$	
		$\rho ::= (f, E) \mid \rho, \rho$	records

$$\Pi ::= E = F \mid E \neq F \mid \Pi \wedge \Pi \quad \text{pure formulas}$$
$$\Sigma ::= \mathit{emp} \mid E \mapsto \{\rho\} \mid P(E, \vec{F}) \mid \Sigma * \Sigma \quad \text{spatial formulas}$$
$$\varphi ::= \exists \vec{X} : \Pi \wedge \Sigma \quad \text{formulas}$$

Fig. 1 The syntax of the considered separation logic fragment

have unique names. The set of program variables used in a formula φ is denoted by $pv(\varphi)$. By $\varphi(\vec{E})$ (resp. $\rho(\vec{E})$), we denote a formula (resp. a set of field-variable pairs) whose set of free variables is $\vec{E}$, and we use $\mathit{free}(\varphi(\vec{E}))$ and $\mathit{free}(\rho(\vec{E}))$ to denote $\vec{E}$.

Pure formulas characterise the stack of the program using (dis)equalities between location variables. Given a formula φ, $\mathit{pure}(\varphi)$ denotes its pure part Π. We allow set operations to be applied on vectors, i.e., vectors can be treated as sets of their elements. Moreover, $E \neq \vec{F}$ is a shorthand for $\bigwedge_{F_i \in \vec{F}} E \neq F_i$.

The atomic spatial formula emp denotes an empty heap. The *points-to atom* $E \mapsto \{(f_i, F_i)\}_{i \in \mathcal{I}}$ denotes a heap containing a record at the location labelled by E whose field f_i points to F_i, for all i. Wlog, we assume that each field f_i appears at most once in the set of pairs $\{(f_i, F_i)\}_{i \in \mathcal{I}}$. The *separating conjunction* $*$ specifies the union of two disjoint heaps. The *predicate atom* $P(E, \vec{F})$ specifies a heap fragment described by the predicate P and delimited by its arguments, i.e., all locations it represents are reachable from E and allocated on the heap, except the locations in $\vec{F}$.

The fragment is parameterised by a set $\mathbb{P}$ of *inductively defined predicates*. An *inductive definition* of $P \in \mathbb{P}$ is a finite set of rules of the form $P(X, \vec{Y}) ::= \exists \vec{Z} : \Pi \wedge \Sigma$. In this work, we consider only inductive definitions for possibly empty *nested list segments*, defined formally in Section 2.3.

2.2 Semantics

Formulas of our SL fragment are interpreted over pairs (S, H) where S models the program stack and H the program heap. The stack $S : \mathit{Vars} \cup \mathit{LVars} \to \mathit{Locs}$ maps variables to locations. The heap $H : \mathit{Locs} \times \mathbb{F} \rightharpoonup \mathit{Locs}$ is a partial function that defines values of fields for some of the locations in Locs. The domain of H is denoted by $dom(H)$, and the set of locations in the domain of H is denoted by $ldom(H)$. We say that a location ℓ is *allocated* in (S, H) or that (S, H) allocates ℓ iff ℓ belongs to $ldom(H)$, and we say that a variable E is allocated iff the location $S(E)$ is allocated. A location (resp. variable) which is not allocated is called *dangling*. A *sub-model* of (S, H) is a pair (S', H') such that $S \subseteq S'$, $H \subseteq H'$, and for any $\ell \in ldom(H')$ and $f \in \mathbb{F}$, it holds that $H'(\ell, f) = H(\ell, f)$, i.e., a location in the domain of a sub-model is included with all its fields defined in the model.

The set of models satisfying a formula φ is given by the relation $(S, H) \models \varphi$ defined in Fig. 2. The semantic rules are standard except the predicate atom where the model satisfying a predicate $P(E, \vec{F})$ *cannot* allocate any variable in $\vec{F}$ as these vari-

$$
\begin{array}{lll}
(S,H) \models E = F & \text{iff} & S(E) = S(F) \\
(S,H) \models E \neq F & \text{iff} & S(E) \neq S(F) \\
(S,H) \models \varphi \wedge \psi & \text{iff} & (S,H) \models \varphi \text{ and } (S,H) \models \psi \\
(S,H) \models emp & \text{iff} & dom(H) = \emptyset \\
(S,H) \models E \mapsto \{\rho\} & \text{iff} & dom(H) = \{(S(E), f_i) \mid (f_i, E_i) \in \{\rho\}\} \text{ and} \\
 & & \text{for every pair } (f_i, E_i) \in \{\rho\}, \text{ it holds that} \\
 & & H(S(E), f_i) = S(E_i) \\
(S,H) \models \Sigma_1 * \Sigma_2 & \text{iff} & \text{there exist } H_1 \text{ and } H_2 \text{ s.t.} \\
 & & ldom(H) = ldom(H_1) \uplus ldom(H_2), \\
 & & (S,H_1) \models \Sigma_1, \text{ and } (S,H_2) \models \Sigma_2 \\
(S,H) \models P(E, \vec{F}) & \text{iff} & \text{there exists a rule } (P(X, \vec{Y}) ::= \exists \vec{Z} : \Pi \wedge \Sigma) \in \mathbb{P} \text{ s.t.} \\
 & & (S,H) \models \exists \vec{Z} : (\Pi \wedge \Sigma)[E/X, \vec{F}/\vec{Y}] \text{ and} \\
 & & ldom(H) \cap \{S(F) \mid F \in \vec{F}\} = \emptyset \\
(S,H) \models \exists X : \varphi & \text{iff} & \exists \ell \in Locs \text{ s.t. } (S[X \leftarrow \ell], H) \models \varphi
\end{array}
$$

Fig. 2 The $\models$ relation ($\uplus$ denotes the disjoint union of sets, $\mathbb{P}$ is the set of inductively defined predicates, $[X/Y]$ denotes a substitution of Y by X, and $S[X \leftarrow \ell]$ denotes the function S' such that $S'(X) = \ell$ and $S'(Y) = S(Y)$ for any $Y \neq X$)

ables are considered not to be in its domain (which differs, e.g., from the semantics in [1]). A model satisfying this property is called *well-formed wrt the atom* $P(E, \vec{F})$.

The set of models of a formula φ is denoted by $[\![\varphi]\!]$. Given two formulas φ_1 and φ_2, we say that φ_1 entails φ_2, denoted by $\varphi_1 \Rightarrow \varphi_2$, iff $[\![\varphi_1]\!] \subseteq [\![\varphi_2]\!]$. By an abuse of notation, $\varphi \Rightarrow E = F$ (resp. $\varphi \Rightarrow E \neq F$) denotes the fact that E and F are interpreted to the same location (resp. different locations) in all models of φ.

2.3 Inductive Definitions for Nested Lists

We consider a class of restricted inductive definitions that are expressive enough to deal with intricate singly linked lists (including simple lists, lists of circular lists, skip lists of fixed depth, etc.) while also enabling efficient entailment checking (we also extend our fragment to doubly linked lists in Section 8). We define this class by requiring the following restrictions on the general inductive definitions. Examples of inductive predicates conforming to our restrictions are in Fig. 3.

Constraint 1 (*Linearity*): Each predicate $P \in \mathbb{P}$ has at least two formal parameters and exactly two rules: (i) *an empty base rule* of the form $P(E, F, \vec{B}) ::= E = F \wedge emp$ specifying an empty list segment, and (ii) *an inductive rule* with the following syntax, where Σ' does not contain occurrences of the atom P:

$$P(E, F, \vec{B}) ::= \exists X_{\mathtt{tl}}, \vec{Z} : E \neq \{F\} \cup \vec{B} \wedge \underbrace{E \mapsto \{\rho\} * \Sigma'}_{mat(P)} * P(X_{\mathtt{tl}}, F, \vec{B}) \quad (1)$$

The parameters are divided into three categories: the *source* (or *root*) parameter E, the *target* parameter F, and the vector of *border* parameters $\vec{B}$. The formula $E \mapsto \{\rho\} * \Sigma'$ is called *the matrix of* P and denoted by $mat(P)$.

We use the constraint $E \neq \{F\} \cup \vec{B}$ to syntactically denote the semantic constraint that locations for target and border parameters are not allocated in a non-empty heap specified by the predicate (see Fig. 2). Intuitively, the inductive rule of P defines a heap composed of a sequence of sub-heaps specified by the matrix of P between the locations given by the actual source and target parameters.

singly linked lists:

$$\mathtt{ls}(E,F) ::= \exists X_{\mathtt{t1}} : E \neq F \wedge E \mapsto \{(f, X_{\mathtt{t1}})\} * \mathtt{ls}(X_{\mathtt{t1}}, F)$$

lists of acyclic lists:

$$\mathtt{nll}(E,F,B) ::= \exists X_{\mathtt{t1}}, Z : E \neq \{F,B\} \wedge E \mapsto \{(s, X_{\mathtt{t1}}), (h, Z)\} * \mathtt{ls}(Z,B) * \mathtt{nll}(X_{\mathtt{t1}}, F, B)$$

lists of cyclic lists:

$$\mathtt{nlcl}(E,F) ::= \exists X_{\mathtt{t1}}, Z : E \neq F \wedge E \mapsto \{(s, X_{\mathtt{t1}}), (h, Z)\} * \circlearrowleft^{1+} \mathtt{ls}[Z] * \mathtt{nlcl}(X_{\mathtt{t1}}, F)$$

skip lists with three levels:

$$\mathtt{skl}_3(E,F) ::= \exists X_{\mathtt{t1}}, Z_1, Z_2 : E \neq F \wedge E \mapsto \{(f_3, X_{\mathtt{t1}}), (f_2, Z_2), (f_1, Z_1)\} * \mathtt{skl}_1(Z_1, Z_2) * \mathtt{skl}_2(Z_2, X_{\mathtt{t1}}) * \mathtt{skl}_3(X_{\mathtt{t1}}, F)$$

$$\mathtt{skl}_2(E,F) ::= \exists X_{\mathtt{t1}}, Z_1 : E \neq F \wedge E \mapsto \{(f_3, \mathtt{null}), (f_2, X_{\mathtt{t1}}), (f_1, Z_1)\} * \mathtt{skl}_1(Z_1, X_{\mathtt{t1}}) * \mathtt{skl}_2(X_{\mathtt{t1}}, F)$$

$$\mathtt{skl}_1(E,F) ::= \exists X_{\mathtt{t1}} : E \neq F \wedge E \mapsto \{(f_3, \mathtt{null}), (f_2, \mathtt{null}), (f_1, X_{\mathtt{t1}})\} * \mathtt{skl}_1(X_{\mathtt{t1}}, F)$$

Fig. 3 Examples of inductive definitions used throughout this paper (we omit all base rules $P(E,F,\vec{B}) ::= E = F \wedge \mathit{emp}$ for all predicates P)

Constraint 2 (*Root atom*): For each predicate $P \in \mathbb{P}$, the formula Σ' does not contain points-to atoms. The atom $E \mapsto \{\rho\}$ is called the *root atom*, and it is denoted by $\mathit{root}(P)$. Furthermore, free variables of ρ contain all existentially quantified variables of the inductive rule, and they can only also contain border parameters, i.e., $\{X_{\mathtt{t1}}\} \cup \vec{Z} \subseteq \mathit{free}(\rho) \subseteq \{X_{\mathtt{t1}}\} \cup \vec{Z} \cup \vec{B}$.

Intuitively, this constraint requires that the occurrence of $mat(P)$ in the next unfolding of P, which has $X_{\mathtt{t1}}$ as the root, is pointed by at least one field from E. This condition is satisfied by all inductive definitions in Fig. 3, but it forbids the following inductive definition defining lists segments of even length (we often omit the base rule in what follows):

$$\mathtt{evenls}(E,F) ::= \exists X_1, X_{\mathtt{t1}} : E \neq F \wedge E \mapsto \{(f, X_1)\} * X_1 \mapsto \{(f, X_{\mathtt{t1}})\} * \mathtt{evenls}(X_{\mathtt{t1}}, F).$$

Also, this restriction forbids inductive definitions that are not compositional (see Property 4, pg. 11), such as list segments with fast-forward pointers to the end node:

$$\mathtt{lstf}(E,F) ::= \exists X_{\mathtt{t1}} : E \neq F \wedge E \mapsto \{(f, X_{\mathtt{t1}}), (g, F)\} * \mathtt{lstf}(X_{\mathtt{t1}}, F).$$

Note that this is not at a loss of expressiveness because such list segments may be obtained using the inductive definition below that defines list segments with all elements pointing to some border location B:

$$\mathtt{lsb}(E,F,B) ::= \exists X_{\mathtt{t1}} : E \neq \{F,B\} \wedge E \mapsto \{(f, X_{\mathtt{t1}}), (g, B)\} * \mathtt{lsb}(X_{\mathtt{t1}}, F, B).$$

Then, $\mathtt{lsb}(E,F,F)$ specifies list segments with pointers to the end node of the list.

Moreover, the constraint forbids inductive definitions where the matrix uses an existentially quantified variable Z not pointed by the root atom, such as, e.g.:

$$\mathtt{nfls}(E,F,B) ::= \exists X_{\mathtt{t1}}, Y, Z : E \neq \{F,B\} \wedge E \mapsto \{(f, X_{\mathtt{t1}}), (g, Y), (h, B)\} * \mathtt{ls}(Y,Z) * \mathtt{nfls}(X_{\mathtt{t1}}, F, B).$$

Constraint 3 (*Nested list segments*): For each $P \in \mathbb{P}$, the matrix of P contains the root atom $*$-connected with formulas of the following form (for $Q \neq P$):

$$\Sigma' ::= Q(Z, U, \vec{Y}) \mid \circlearrowleft^{1+} Q[Z, \vec{Y}] \mid \Sigma' * \Sigma' \mid emp \quad (2)$$
$$\text{for } Z \in \vec{Z}, U \in \vec{Z} \cup \vec{B} \cup \{E, X_{\mathtt{tl}}\}, \vec{Y} \subseteq \vec{B} \cup \{E, X_{\mathtt{tl}}\}$$
$$\circlearrowleft^{1+} Q[Z, \vec{Y}] \equiv \exists Z' : mat(Q(Z, Z', \vec{Y})) * Q(Z', Z, \vec{Y}) \quad (3)$$

Notice that F does not appear in the matrix of P. The macro $\circlearrowleft^{1+} Q[Z, \vec{Y}]$ is used to represent a *non-empty* cyclic (nested) list segment on Z whose shape is described by the predicate Q. We call predicate atoms $Q(Z, U, \vec{Y})$ and macros $\circlearrowleft^{1+} Q[Z, \vec{Y}]$ *extended predicate atoms*.

Intuitively, this constraint requires nested lists to have their sources in $\vec{Z}$, i.e., in a variable referenced by a field from the location of E (due to the previous constraint). Except for $X_{\mathtt{tl}}$, the target of these nested list segments is either a location pointed by the fields of E (e.g., $\mathtt{skl}_3$ in Fig. 3), a border location in $\vec{B}$ (e.g., $\mathtt{nll}$), or E. The $\circlearrowleft^{1+} Q[Z, \vec{Y}]$ macro is needed to define nested (non-empty) circular lists; defining them as $Q(Z, Z, \vec{Y})$ would make them empty (the only rule allowed for instances of predicates with matching source and target parameter is the empty base rule).

The next constraint on the matrix of P is defined using its Gaifman graph. Let Σ be the matrix of some inductive definition $P \in \mathbb{P}$. The *Gaifman graph* of Σ, denoted $Gf[\Sigma]$, is a labelled graph where:

- The set of vertices is given by the set of free and existentially quantified variables in Σ, i.e., $\{E, X_{\mathtt{tl}}\} \cup \vec{B} \cup \vec{Z}$.
- Edges represent spatial atoms as follows: let $E \mapsto \{\rho\}$ be the root atom of Σ, then for every (f, X) in $\{\rho\}$, $Gf[\Sigma]$ contains an edge from E to X labelled by f; for every $Q(Z, U, \vec{Y})$, $Gf[\Sigma]$ contains an edge from Z to U labelled by Q; and for every macro $\circlearrowleft^{1+} Q[Z, \vec{Y}]$, $Gf[\Sigma]$ contains a self-loop on Z labelled by Q.

Constraint 4 (*Matrix connectedness*): Let Σ be the matrix of $P \in \mathbb{P}$. Then all *infinite* paths of $Gf[\Sigma]$ either form a cycle going through E or start in E and end in a self-loop built from some macro $\circlearrowleft^{1+} Q[Z, \vec{Y}]$, and all maximal *finite* paths start in E and end in a node from $\vec{B} \cup \{X_{\mathtt{tl}}\}$. Moreover, we require that every vertex of $Gf[\Sigma]$ has at most one outgoing edge labelled by a predicate.

Intuitively, the constraint requires that every existential variable in an inductive rule appears as the source parameter of exactly one extended predicate atom. This ensures that every existential variable Z is either allocated in the matrix (when the list segment starting from Z is non-empty or it ends in E) or it aliases the target or a border parameter. The inductive definitions given in Fig. 3 satisfy the above constraint. The following inductive definition is, however, forbidden because it contains a dangling existential variable Z:

$$\mathtt{npls}(E, F, B) ::= \exists X_{\mathtt{tl}}, Y, Z : E \neq \{F, B\} \wedge E \mapsto \{(f, X_{\mathtt{tl}}), (g, Y), (h, Z)\} * \mathtt{ls}(Y, B) * \mathtt{npls}(X_{\mathtt{tl}}, F, B).$$

The constraint also forbids the following inductive definition:

$$\mathtt{nls2}(E, F, B) ::= \exists X_{\mathtt{tl}}, Y, Z : E \neq \{F, B\} \wedge E \mapsto \{(f, X_{\mathtt{tl}}), (f_1, Y), (f_2, Z)\} * \mathtt{ls}(Y, Z) * \mathtt{ls}(Z, Y) * \mathtt{nls2}(X_{\mathtt{tl}}, F, B).$$

This is because the Gaifman graph of its matrix contains a loop which is not a self-loop—it traverses the inner vertices represented by variables Y and Z. Such loops are forbidden because they may produce dangling variables when the list segments composing the loop are all empty. Dangling variables are problematic because they may be aliased with any variable occurring outside the occurrence of a predicate, which is difficult to encode in our procedure.

The following inductive definition satisfies the matrix constraint because the list segment from Z is a self-loop:

$$\texttt{nlls}(E,F,B) \ ::= \ \exists X_{\texttt{tl}}, Y, Z : E \neq \{F,B\} \ \wedge \ E \mapsto \{(f, X_{\texttt{tl}}), (f_1, Y), (f_2, Z)\} \\ * \ \texttt{ls}(Y,Z) \ * \ \circlearrowleft^{1+} \texttt{ls}[Z] \ * \ \texttt{nlls}(X_{\texttt{tl}}, F, B).$$

Finally, the following restrictions limit the use of predicate atoms and fields in inductive definitions of $\mathbb{P}$. For this, we define the relation $\prec_{\mathbb{P}}$ on $\mathbb{P}$ by $P_1 \ \prec_{\mathbb{P}} \ P_2$ iff P_2 appears in the matrix of P_1. For example, if $\mathbb{P} = \{\texttt{skl}_1, \texttt{skl}_2, \texttt{skl}_3\}$, then $\texttt{skl}_3 \prec_{\mathbb{P}} \texttt{skl}_2 \prec_{\mathbb{P}} \texttt{skl}_1 \ \wedge \ \texttt{skl}_3 \prec^*_{\mathbb{P}} \texttt{skl}_1$. Here, $\prec^*_{\mathbb{P}}$ is the reflexive and transitive closure of $\prec_{\mathbb{P}}$.

Constraint 5 (*No mutual recursion*): Given a set of inductive definitions $\mathbb{P}$, $\prec^*_{\mathbb{P}}$ is a partial order.

Let $\mathbb{F}_{\mapsto}(P)$ denote the set of fields occurring in $root(P)$. For example, in the inductive definitions in Fig. 3, it holds that $\mathbb{F}_{\mapsto}(\texttt{nll}) = \{s, h\}$ and $\mathbb{F}_{\mapsto}(\texttt{skl}_3) = \mathbb{F}_{\mapsto}(\texttt{skl}_1) = \{f_3, f_2, f_1\}$. Also, let $\mathbb{F}^*_{\mapsto}(P)$ denote the union of $\mathbb{F}_{\mapsto}(P')$ for all $P \prec^*_{\mathbb{P}} \ P'$. For example, $\mathbb{F}^*_{\mapsto}(\texttt{nll}) = \{s, h, f\}$.

Constraint 6 (*No shared fields*): For any two predicates P_1 and P_2 that are incomparable wrt $\prec^*_{\mathbb{P}}$, it holds that $\mathbb{F}_{\mapsto}(P_1) \cap \mathbb{F}_{\mapsto}(P_2) = \emptyset$.

Therefore, we forbid predicates named differently but having exactly the same set of models. Moreover, we require the existence of a total ordering on fields, denoted $\prec_{\mathbb{F}}$, which complies with the inductive definition of predicates in $\mathbb{P}$. Intuitively, $\prec_{\mathbb{F}}$ shall reflect the order in which the unfolding of the inductive definition of P is done. Therefore, fields used in the root atom $E \mapsto \{\rho\}$ of the matrix of P are ordered before fields of any other predicate called by P. Fields appearing in ρ and going "one-step forward" (i.e. occurring in a pair $(f, X_{\texttt{tl}})$) are ordered before fields leading to "inner" locations (i.e. occurring in a pair (f, Z) with $Z \in \vec{Z}$), which are ordered before fields going to the border parameters (i.e. occurring in a pair (f, B) with $B \in \vec{B}$). We note that `null` is considered a constant, not a border variable.

Formally, for a predicate P defined by an inductive rule as in Equation (1) (pg. 6), we partition $\mathbb{F}_{\mapsto}(P)$ in four sub-sets: (a) $\mathbb{F}_{\mapsto X_{\texttt{tl}}}(P)$ is the set of fields f occurring in a pair $(f, X_{\texttt{tl}})$ of ρ, (b) $\mathbb{F}_{\mapsto \mathbf{Z}}(P)$ is the set of fields f occurring in a pair (f, Z) with $Z \in \vec{Z}$, (c) $\mathbb{F}_{\mapsto \mathbf{B}}(P)$ is the set of fields f occurring in a pair (f, X) with $X \in \vec{B} \setminus \{\texttt{null}\}$, and (d) $\mathbb{F}_{\mapsto \texttt{null}}(P)$ is the set of fields f occurring in a pair $(f, \texttt{null})$.

Constraint 7 (*Total ordered fields*): There exists a total order $\prec_{\mathbb{F}}$ on $\mathbb{F}$ such that for all P, P_1, and P_2 in $\mathbb{P}$:

$$\forall f_1 \in \mathbb{F}_{\mapsto X_{\texttt{tl}}}(P), \ \forall f_2 \in \mathbb{F}_{\mapsto \mathbf{Z}}(P), \ \forall f_3 \in \mathbb{F}_{\mapsto \mathbf{B}}(P) : f_1 \prec_{\mathbb{F}} f_2 \prec_{\mathbb{F}} f_3 \text{ and} \quad (4)$$

$$(f_1 \in \mathbb{F}_{\mapsto}(P_1) \wedge f_2 \in \mathbb{F}_{\mapsto}(P_2) \wedge f_1 \neq f_2 \wedge P_1 \prec_{\mathbb{P}} P_2) \Rightarrow f_1 \prec_{\mathbb{F}} f_2. \quad (5)$$

For instance, if $\mathbb{P} = \{\texttt{nll}, \texttt{ls}\}$ or $\mathbb{P} = \{\texttt{nlcl}, \texttt{ls}\}$, then $s \prec_{\mathbb{F}} h \prec_{\mathbb{F}} f$ satisfies the constrains above. Also, if $\mathbb{P} = \{\texttt{skl}_2, \texttt{skl}_1\}$, then both $f_2 \prec_{\mathbb{F}} f_1 \prec_{\mathbb{F}} f_3$ and $f_3 \prec_{\mathbb{F}} f_2 \prec_{\mathbb{F}} f_1$ are correct total orderings of fields. Only the last one, however, complies with the constraint above for $\mathbb{P} = \{\texttt{skl}_3, \texttt{skl}_2, \texttt{skl}_1\}$. So, fields in $\mathbb{F}_{\mapsto \texttt{null}}(P)$ shared with another predicate Q are ordered to agree with the ordering of fields in Q; in absence of any ordering constraint, they may be ordered by $\prec_{\mathbb{F}}$ in any possible way.

2.4 Properties of Models for Predicate Atoms

The constraints on the inductive definitions, together with the basic syntax and semantics of the introduced SL fragment, including the restriction to well-formed models, induce some properties of the considered models of predicate atoms that are important for the soundness of our procedure. These properties are given in this section.

Some of the properties on well-formed models (S, H) of a predicate atom $P(E, F, \vec{B})$ are expressed using their representation as labelled directed graphs. The *heap graph* of a model (S, H) has as vertices the locations in $ldom(H) \cup img(H)$; these locations are labelled by sets of program and logic variables using S^{-1}. The heap graph edges are labelled by fields such that (ℓ, f, ℓ') is an edge iff $H(\ell, f) = \ell'$.

Property 1 (*Reachability from root*): Any location ℓ in (S, H) is reachable from the location $S(E)$.

Proof Constraints 1 and 2 ensure there is a path from the source to the target of a predicate edge and Constraint 4 ensures connectedness of the predicate's matrix. □

Property 2 (*No inner dangling*): Any maximal path of (S, H) starting in $S(E)$ is either cyclic or ends in a location labelled by a variable in $\{F\} \cup \vec{B}$. Therefore, only locations labelled by $\{F\} \cup \vec{B}$ are dangling.

Proof Follows from Property 1 and Constraint 4. □

The next property is a consequence of the semantics of formulas—in particular, the restriction to well-formed models of predicate atoms.

Property 3 (*Precise assertions*): For any model (S, H) of a formula φ including some predicate atom $P(E, F, \vec{B})$, there exists at most one well-formed sub-model (S', H') of (S, H) such that $(S', H') \models P(E, F, \vec{B})$.

Proof By contradiction. Suppose (S', H') is the smallest well-formed sub-model (there is at most one due to determinism of heaps) of (S, H) such that $(S', H') \models P(E, F, \vec{B})$ and that there exists another well-formed sub-model (S'', H'') such that $(S'', H'') \models P(E, F, \vec{B})$ and (S', H') is a proper sub-model of (S'', H''). It follows that (S'', H'') contains an allocated location ℓ that is not in (S', H'). From Property 1, it follows that ℓ is reachable from $S(E)$, and from the fact that heaps are deterministic, we know that there is an allocated location ℓ' in (S'', H'') such that it is a dangling node of (S', H'), and ℓ is reachable from ℓ'. From Property 2, it follows that $\ell' \in S^{-1}(\{F\} \cup \vec{B})$. Therefore, (S'', H'') allocates a node from $\{F\} \cup \vec{B}$, so it is not a well-formed model of $P(E, F, \vec{B})$, which is a contradiction. □

Constraints 2–4 imply that inductive definitions are *compositional*:

Property 4 (*Compositional List Segments*): For any $P \in \mathbb{P}$ and any model (S, H) such that $(S, H) \models P(E, F, \vec{B}) * P(F, G, \vec{B})$ and G is not allocated in (S, H), it holds that $(S, H) \models P(E, G, \vec{B})$.

Proof By induction on the length of the left-hand side occurrence of P. For the base rule $E = F$, a model $(S_1, H_1) \models E = F \wedge emp * P(F, G, \vec{B})$ is also a model of $P(E, G, \vec{B})$. For the inductive rule, assume that if $(S_2, H_2) \models P(X_{\mathtt{tl}}, F, \vec{B}) * P(F, G, \vec{B})$ and (S_2, H_2) does not allocate G, then it holds that $(S_2, H_2) \models P(X_{\mathtt{tl}}, G, \vec{B})$. Let us consider $P(E, F, \vec{B}) * P(F, G, \vec{B})$ such that if we once unfold $P(E, F, \vec{B})$, we obtain $\psi ::= \exists X_{\mathtt{tl}}, \vec{Z} \,.\, E \neq \{F\} \cup \vec{B} \wedge mat(P) * P(X_{\mathtt{tl}}, F, \vec{B}) * P(F, G, \vec{B})$. Suppose (S, H) is a model of ψ that does not allocate G. Due to the induction hypothesis, we infer that $(S, H) \models \exists X_{\mathtt{tl}}, \vec{Z} \,.\, E \neq \{G\} \cup \vec{B} \wedge mat(P) * P(X_{\mathtt{tl}}, G, \vec{B})$. From the inductive rule for P, it follows that $(S, H) \models P(E, G, \vec{B})$. □

The key for the encoding of SL formulas entailing predicate atoms $P(E, F, \vec{B})$ as trees (see Section 6) is given by the following properties. We call a path simple if it does not pass through the same node repeatedly.

Property 5 (*Joining paths*): Let (S, H) be a well-formed model of $P(E, F, \vec{B})$ and ℓ be an allocated location in (S, H) with multiple incoming edges such that $\ell \neq S(E)$. Then there is a unique edge $\ell' \xrightarrow{f} \ell$ with f minimal wrt $\prec_{\mathbb{F}}$. Moreover, the other incoming edges are last edges of simple paths starting from ℓ' or ℓ.

Proof Constraints 1 and 3 imply that there are two cases: (a) ℓ corresponds to the first node of a predicate atom $R(\ldots)$ (resp. a macro $\circlearrowleft^{1+}R(\ldots)$) s.t. $P \prec^{*}_{\mathbb{P}} R$, or (b) ℓ is an internal (i.e. not the first) node of $R(\ldots)$ (resp. $\circlearrowleft^{1+}R(\ldots)$) or $P(E, F, \vec{B})$.

Case (a): From Constraint 1, ℓ corresponds to some variable $Z \in \vec{Z}$ from the non-empty rule of a predicate T s.t. $P \prec^{*}_{\mathbb{P}} T \prec_{\mathbb{P}} R$. Then, from Constraint 2, the root atom $root(T)$ contains a pair (f, Z) s.t. f is, by Constraint 7, the least label of edges entering ℓ (wrt $\prec_{\mathbb{F}}$). Moreover, also by Constraint 7, there are no more f-edges entering ℓ. Hence, the source of $root(T)$ plays the role of ℓ'.

Case (b): From Constraints 2 and 7, there is a pair $(f, X_{\mathtt{tl}})$ in $root(R)$ or $root(P)$, respectively, such that f is smaller (wrt $\prec_{\mathbb{F}}$) than any other edge entering ℓ (which may be, e.g., edges in nested list segments from some $\vec{Z}$ variables). Hence, ℓ' is the predecessor of ℓ according to f.

The last sentence of the property follows from Constraints 3 and 4, in particular from requirements on the use of inductive predicates and the macro $\circlearrowleft^{1+}Q[Z, \vec{Y}]$, respectively, in the matrix of an inductive rule. □

A corollary of the previous property is that for any allocated location ℓ with several incoming edges, there exist paths σ, σ', a location ℓ' that is not an internal location of σ', and an edge label f such that the following holds: $S(E) \overset{\sigma}{\rightsquigarrow} \ell' \xrightarrow{f} \ell$ and either (i) $S(E) \overset{\sigma}{\rightsquigarrow} \ell' \overset{\sigma'}{\rightsquigarrow} \ell$, or (ii) $S(E) \overset{\sigma}{\rightsquigarrow} \ell' \xrightarrow{f} \ell \overset{\sigma'}{\rightsquigarrow} \ell$. Then, given $\sigma\sigma'$ (resp. $\sigma f \sigma'$) and f, we can unambiguously determine σf. In particular, for (i), we can obtain σf by traversing (S, H) from ℓ backwards along $\sigma\sigma'$ up to the first node (in fact, ℓ') that defines an f edge to a non-null location. Similarly for (ii). This property is important for the selection of aliasing relations in encoding graphs as trees in Section 6.

Property 6 (*Minimal path*): For any allocated location ℓ in (S, H), there is a path σ_{min} in its heap graph from $S(E)$ to ℓ such that for any edge $\ell_i \xrightarrow{f_i} \ell_{i+1}$ in σ_{min}, the label f_i is the least (wrt $\prec_{\mathbb{F}}$) label among labels of edges entering ℓ_{i+1}.

Proof Follows from Properties 1 and 5. □

Due to this property, removing edges entering a node that are not labelled by the minimal field keeps the model connected, so we can represent it using a tree.

3 Compositional Entailment Checking

We now provide our procedure for reducing the problem of checking validity of entailment between two formulas to the problem of checking validity of entailment between a formula and an atom. In particular, we consider the problem of deciding validity of entailments $\varphi_1 \Rightarrow \varphi_2$ where φ_2 is free of quantifiers and $\mathit{free}(\varphi_2) \subseteq \mathit{free}(\varphi_1)$, which usually suffices for checking verification conditions in practice. We assume $pv(\varphi_2) \subseteq pv(\varphi_1)$; otherwise, the entailment is trivially invalid.

The main steps of the reduction are given in Algorithm 1. The reduction starts by a normalisation step (described in Section 4), which adds to each of the two formulas all (dis-)equalities implied by their spatial sub-formulas and removes all atoms $P(E, F, \vec{B})$ representing *empty* list segments, i.e. those where $E = F$ occurs in the pure part. The normalisation of a formula returns false iff the formula is unsatisfiable.

Algorithm 1: Compositional entailment checking of $\varphi_1 \Rightarrow \varphi_2$ for $\prec$ being any total order compatible with $\prec^*_{\mathbb{P}}$

```
1  φ1^n ← norm(φ1);   // normalisation
2  φ2^n ← norm(φ2);
3  if φ1^n = false then return true;
4  if φ2^n = false then return false;
   // pure parts
5  if pure(φ1^n) ⇏ pure(φ2^n) then return false;
   // points-to atoms
6  foreach points-to atom a2 in φ2^n do
7  |  φ1^n[a2] ← select(φ1^n, a2);
8  |  if φ1^n[a2] ⇏ a2 then return false;
9  |  mark(φ1^n[a2]);
   // predicate atoms
10 for P2 ← max_≺(ℙ) downto min_≺(ℙ) do
11 |  foreach a2 = P2(E, F, B⃗) in φ2^n s.t.
   |    pure(φ1^n) ⇏ E = F do
12 |  |  φ1^n[a2] ← select(φ1^n, a2);
13 |  |  if φ1^n[a2] ⇏_sh a2 then return false;
14 |  |  mark(φ1^n[a2]);
15 return isMarked(φ1^n);
```

In the second step, the procedure tests entailment between the pure parts of the normalised formulas. This can be done using any decision procedure for quantifier-free formulas in the first-order theory with equality.

Next, for the spatial parts, the procedure uses the function `select`, described in Sect. 5, to build a mapping from spatial atoms of φ_2^n to sub-formulas of φ_1^n. The sub-formula of φ_1^n to which an atom a_2 of φ_2^n is mapped in this way is denoted as $\varphi_1^n[a_2]$. The mapping is built by first enumerating the points-to atoms of φ_2^n and only then by enumerating its predicate atoms, in a decreasing order wrt $\prec^*_{\mathbb{P}}$. The decreasing order is important for the completeness of the procedure (see Section 9). Intuitively, the formula $\varphi_1^n[a_2]$ associated to an atom a_2 of φ_2^n describes the region of a heap modelled by φ_1^n that should satisfy a_2.

The construction of $\varphi_1^n[a_2]$ may fail, implying that the entailment $\varphi_1 \Rightarrow \varphi_2$ is not valid. In such a case, `select` returns emp, causing the algorithm to return false.

For predicate atoms $a_2 = P_2(E, F, \vec{B})$, handled in the second loop of the algorithm, `select` is called only if there exists a model of φ_1^n where the heap region that should satisfy a_2 is non-empty, i.e. $E = F$ does not occur in φ_1^n. In this case, `select` also checks that for any model of φ_1^n, the sub-heap corresponding to the atoms in $\varphi_1^n[a_2]$ is well-formed wrt a_2. This check is needed since all heaps described by a_2 are well-formed (see Section 2.2).

Note that in the well-formedness check above, one cannot speak about $\varphi_1^n[a_2]$ alone. Indeed, without the rest of φ_1^n, the formula $\varphi_1^n[a_2]$ may have models that are not well-formed wrt a_2 even if the sub-heap corresponding to $\varphi_1^n[a_2]$ is well-formed for any model of φ_1^n. For example, let $\varphi_1^n = \mathtt{ls}(x, y) * \mathtt{ls}(y, z) * z \mapsto \{(f, t)\}$, $a_2 = \mathtt{ls}(x, z)$, and $\varphi_1^n[a_2] = \mathtt{ls}(x, y) * \mathtt{ls}(y, z)$. If we take models of φ_1^n only, the sub-heaps corresponding to $\varphi_1^n[a_2]$ are all well-formed wrt a_2, i.e. they do not allocate the location bound to z. The formula $\varphi_1^n[a_2]$ alone has, however, lasso-shaped models where the location bound to z is allocated on the path between x and y.

Once $\varphi_1^n[a_2]$ is obtained, one needs to check that all sub-heaps modelled by $\varphi_1^n[a_2]$ are also models of a_2. For points-to atoms a_2, this boils down to a syntactic identity (modulo some renaming given by the equalities in the pure part of φ_1^n). For predicate atoms a_2, a special entailment operator $\Rightarrow_{sh}$ is used. We cannot use the usual entailment $\Rightarrow$ since $\varphi_1^n[a_2]$ may have models that are not sub-heaps of models of φ_1^n (as we have seen in the example above).

Definition 1 $\varphi_1^n[a_2] \Rightarrow_{sh} a_2$ iff all models of $\varphi_1^n[a_2]$ that are well-formed wrt a_2 are also models of a_2.

Given a formula φ and an atom $P(E, F, \vec{B})$, the entailment $\varphi \Rightarrow_{sh} P(E, F, \vec{B})$ is checked as follows: (1) $G[\varphi]$ is transformed into a *tree* $\mathcal{T}[\varphi]$ by splitting nodes that have multiple incoming edges, (2) the inductive definition of $P(E, F, \vec{B})$ is used to construct a TA $\mathcal{A}[P]$ such that $\mathcal{T}[\varphi]$ belongs to the language of $\mathcal{A}[P]$ only if $\varphi \Rightarrow_{sh} P(E, F, \vec{B})$. The transformation of graphs $G[\varphi]$ into trees $\mathcal{T}[\varphi]$ is presented in Section 6 while the construction of the TA $\mathcal{A}[P]$ is introduced in Section 7.

If there exists an atom a_2 of φ_2^n that is not entailed by the associated sub-formula, then $\varphi_1 \Rightarrow \varphi_2$ is not valid. By the semantics of the separating conjunction, the sub-formulas of φ_1^n associated with two different atoms of φ_2^n must not share spatial atoms. In order to avoid such a scenario, the spatial atoms obtained from each application of `select` are marked by the algorithm and cannot be reused in the future. If all entailments between formulas and atoms are valid, then $\varphi_1 \Rightarrow \varphi_2$ holds provided that all spatial atoms of φ_1^n are marked (which is tested by `isMarked`).

Graph representations. Some of the sub-procedures mentioned above work on a graph representation of the input formulas, called *SL graphs* (which are different from the Gaifman graphs of matrices of inductive definitions).

Definition 2 (SL graph) Given a formula φ, its SL graph, denoted by $G[\varphi]$, is a directed labelled graph where:

– Each node n represents an equivalence class over the set of variables, i.e., it represents a maximal set of variables equal wrt the pure part of φ, and it is labelled

by the set of variables $\texttt{Var}(n)$ it represents. For every variable E, we then use $\texttt{Node}(E)$ to denote the node n such that $E \in \texttt{Var}(n)$.

– The following edges can appear: (1) Undirected *disequality edges* from $\texttt{Node}(E)$ to $\texttt{Node}(F)$ encoding disequalities $E \neq F$. (2) Directed *points-to edges* from $\texttt{Node}(E)$ to $\texttt{Node}(E_i)$ labelled by f_i that encode spatial atoms $E \mapsto \{(f_1, E_1), \ldots, (f_n, E_n)\}$, for $1 \leq i \leq n$. (3) Directed *predicate edges* from $\texttt{Node}(E)$ to $\texttt{Node}(F)$ labelled by $P(\vec{B})$ encoding spatial atoms $P(E, F, \vec{B})$.

For simplicity, we confuse a formula φ with its graph representation $G[\varphi]$.

Running example. Below, we use as a running example the entailment $\psi_1 \Rightarrow \psi_2$ between the following formulas:

$$\begin{aligned}\psi_1 \equiv \exists Y_1, Y_2, Y_3, Y_4, Z_1, Z_2, Z_3 : x \neq z \wedge Z_2 \neq z \wedge x \mapsto \{(s, Z_2), (h, Z_1)\} * \\ Z_2 \mapsto \{(s, y), (h, Z_3)\} * \texttt{ls}(Z_1, z) * \texttt{ls}(Z_3, z) * \texttt{ls}(y, Y_1) * \\ \texttt{skl}_2(y, Y_3) * \texttt{ls}(Y_1, Y_2) * Y_3 \mapsto \{(f_2, t), (f_1, Y_4)\} * t \mapsto \{(s, Y_2)\} * \\ Y_4 \mapsto \{(f_2, \texttt{null}), (f_1, t)\}\end{aligned} \tag{6}$$

$$\psi_2 \equiv y \neq t \wedge \texttt{nll}(x, y, z) * \texttt{skl}_2(y, t) * t \mapsto \{(s, y)\}$$

The graph representations of these formulas are shown in Fig. 4(a) and (b)[1].

The formula ψ_1 specifies a heap including a cell whose location is referenced by the (program) variable x and whose fields s and h point to locations Z_2 and Z_1 (atom $x \mapsto \{(s, Z_2), (h, Z_1)\}$). The list cell at location Z_2 contains a field s referencing the location stored in the program variable y, and a field h referencing the location of Z_3. The nodes Z_1 and Z_3 are initial nodes of two disjoint singly linked list segments ending in the location z (atoms $\texttt{ls}(Z_1, z)$ and $\texttt{ls}(Z_3, z)$). The node y is the beginning of a singly linked list segment ending in the location of Y_1 (atom $\texttt{ls}(y, Y_1)$) and a skip list segment ending in the location of Y_3 (atom $\texttt{skl}_2(y, Y_3)$). The heap between Y_3 and t is a fragment of a two-level skip list with a single element on the ground level. Moreover, the variable t references a cell with the field s pointing to the location of the end of the list segment starting from Y_1. The only explicit non-aliasing constraint on program variables is $x \neq z$.

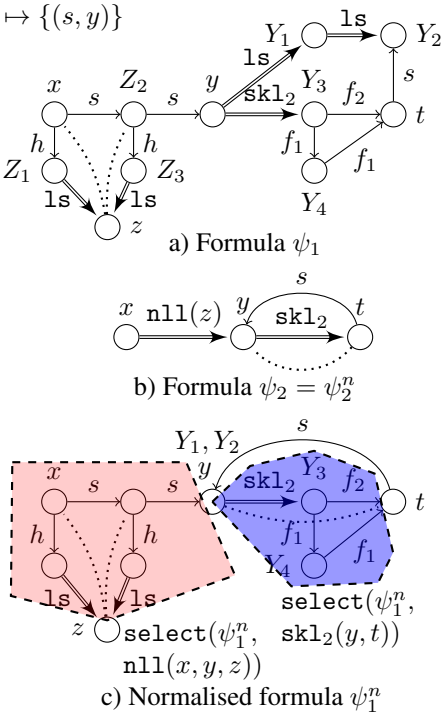


Fig. 4 A running example of an entailment test $\psi_1 \implies \psi_2$.

[1] Points-to edges are depicted as simple lines, predicate edges as double lines, and disequality edges as dotted lines. For readability, we omit some of the labelling with existentially-quantified variables and some of the disequality edges in the normalised graphs.

$$F_{\Pi} ::= \bigwedge_{E=F \in \Pi} [E=F] \ \wedge \bigwedge_{E \neq F \in \Pi} \neg[E=F] \qquad F_{*} ::= \bigwedge_{E,F \text{ variables in } \varphi,\ a \neq a' \text{ atoms in } \Sigma} \big([E=F] \wedge [E,a]\big) \Rightarrow \neg[F,a']$$

$$F_{\Sigma} ::= \bigwedge_{a=E \mapsto \{\rho\} \in \Sigma} [E,a] \ \wedge \bigwedge_{a=P(E,F,\vec{B}) \in \Sigma} ([E,a] \oplus [E=F]) \ \wedge \ ([E,a] \Rightarrow \bigwedge_{B \in \vec{B}} \neg[E=B])$$

$$F_{=} ::= \bigwedge_{E_1,E_2,E_3 \text{ variables in } \varphi} \big([E_1=E_1] \wedge \big([E_1=E_2] \Leftrightarrow [E_2=E_1]\big) \wedge \big([E_1=E_2] \wedge [E_2=E_3]\big) \Rightarrow [E_1=E_3] \big)$$

Fig. 5 Definition of the components of $\mathsf{BoolAbs}[\varphi]$ with $\oplus$ denoting xor

The formula ψ_2 specifies a heap with a nested list segment between locations x and y where all nested list segments end in z (atom $\mathtt{nll}(x,y,z)$) and a skip list segment between locations y and t. It also requires y and t be not aliased.

4 Normalisation

Given a formula φ, the normalisation procedure `norm` computes a new formula φ' that contains all (dis-)equalities among the variables of φ that are implied by the existing ones in φ and the semantics of separating conjunction. This process may discover that φ contains contradictory constraints, i.e., it is unsatisfiable. To infer the implicit (dis-)equalities in a formula, we adapt the Boolean abstraction proposed in [9] for the fragment considered in this paper.

Definition 3 (Boolean abstraction) Given a formula $\varphi \triangleq \exists \vec{X} : \Pi \wedge \Sigma$, we define the Boolean formula $\mathsf{BoolAbs}[\varphi] ::= F_{\Pi} \wedge F_{\Sigma} \wedge F_{=} \wedge F_{*}$ where the components of $\mathsf{BoolAbs}[\varphi]$ are defined in Fig. 5. The set $BV(\varphi)$ of Boolean variables occurring in $BV(\varphi)$ is defined as:

- $[E=F] \in BV(\varphi)$ for every two variables E and F occurring in φ,
- $[E,a] \in BV(\varphi)$ for every variable E and a spatial atom of the form $a = E \mapsto \{\rho\}$ or $a = P(E,F,\vec{B})$ in φ.

Intuitively, the variable $[E=F]$ denotes the equality between E and F, while $[E,a]$ denotes the fact that the atom a describes a heap where E is allocated. The components of $\mathsf{BoolAbs}[\varphi]$, defined in Fig. 5, have the following meaning: F_{Π} and F_{Σ} encode the atoms of φ, $F_{=}$ encodes reflexivity, symmetry, and transitivity of equality, and F_{*} encodes the semantics of the separating conjunction.

Proposition 1 *Let φ be a formula. Then, $\mathsf{BoolAbs}[\varphi]$ is equi-satisfiable with φ, and, for any variables E and F of φ, $\mathsf{BoolAbs}[\varphi] \Rightarrow [E=F]$ (resp. $\mathsf{BoolAbs}[\varphi] \Rightarrow \neg[E=F]$) iff $\varphi \Rightarrow E=F$ (resp. $\varphi \Rightarrow E \neq F$).*

For the formula ψ_1 in our running example, i.e. Equation (6), $\mathsf{BoolAbs}[\psi_1]$ is a conjunction of several formulas including:

1. $[y, \mathtt{skl}_2(y,Y_3)] \oplus [y=Y_3]$, which encodes the atom $\mathtt{skl}_2(y,Y_3)$,
2. $[Y_3, Y_3 \mapsto \{(f_2,t),(f_1,Y_4)\}]$ and $[t, t \mapsto \{(s,Y_2)\}]$, encoding points-to atoms of ψ_1,
3. $\big([t=y] \wedge [t, t \mapsto \{(s,Y_2)\}]\big) \Rightarrow \neg[y, \mathtt{skl}_2(y,Y_3)]$, which encodes the separating conjunction between $t \mapsto \{(s,Y_2)\}$ and $\mathtt{skl}_2(y,Y_3)$,

4. $\big([t = Y_3] \wedge [t, t \mapsto \{(s, Y_2)\}]\big) \Rightarrow \neg[Y_3, Y_3 \mapsto \{(f_2, t), (f_1, Y_4)\}]$, which encodes the separating conjunction between $t \mapsto \{(s, Y_2)\}$ and $Y_3 \mapsto \{(f_2, t), (f_1, Y_4)\}$.

If $\mathsf{BoolAbs}[\varphi]$ is unsatisfiable, $\texttt{norm}(\varphi)$ returns false. Otherwise, the output of $\texttt{norm}(\varphi)$ is the formula φ' obtained from φ by (1) adding all (dis-)equalities $E = F$ (resp. $E \neq F$) such that $[E = F]$ (resp. $\neg[E = F]$) is implied by $\mathsf{BoolAbs}[\varphi]$ and (2) removing all predicate atoms $P(E, F, \vec{B})$ s.t. $E = F$ occurs in the pure part.

For example, the normalisations of ψ_1 and ψ_2 are given in Fig. 4(c) and (b). Note that the `ls` atoms reachable from y are removed because $\mathsf{BoolAbs}[\psi_1]$ implies that Y_1 and Y_2 are aliasing y, and thus the list segments between y, Y_1, and Y_2 are empty. Moreover, $\mathsf{BoolAbs}[\psi_1]$ implies that y is different from t and z. $\mathsf{BoolAbs}[\psi_2]$ does not imply additional (dis-)equalities, so ψ_2 is unchanged after normalisation.

The following result is important for the completeness of the `select` procedure.

Proposition 2 *Let* $\texttt{norm}(\varphi)$ *be the result of the normalisation of a formula* φ*. For any two* distinct *nodes* n *and* n' *in the SL graph of* $\texttt{norm}(\varphi)$*, there cannot exist two* disjoint *sets of atoms* A *and* A' *in* $\texttt{norm}(\varphi)$ *such that both* A *and* A' *form paths between* n *and* n'*.*

Proof Suppose that $\texttt{norm}(\varphi)$ contains two such sets of atoms between nodes n and n' labelled by variables E and F respectively. By the semantics of the separating conjunction, it holds that one of the paths is empty, so that $\varphi \Rightarrow E = F$. Therefore, $\texttt{norm}(\varphi)$ does not include all φ-implied equalities, contradicting its construction. □

5 Selection of Spatial Atoms

After normalisation and testing entailment of pure parts of the checked formulas, the algorithm starts matching every spatial atom from φ_2^n to a set of atoms of φ_1^n. For this, it uses the `select` procedure described in this section.

Points-to atoms. Let $\varphi_1 ::= \exists \vec{X} : \Pi_1 \wedge \Sigma_1$ be a normalised formula. The procedure $\texttt{select}(\varphi_1, E_2 \mapsto \{\rho_2\})$ outputs either the sub-formula $\exists \vec{X} : E_1 = E_2 \wedge E_1 \mapsto \{\rho_1\}$ if $E_1 = E_2$ occurs in Π_1, or the sub-formula emp otherwise. The procedure `select` is called only if φ_1 is satisfiable. Consequently, because of the semantics of the separating conjunction, φ_1 cannot contain two different atoms $E_1 \mapsto \{\rho_1\}$ and $E_1' \mapsto \{\rho_1'\}$ such that $E_1 = E_1' = E_2$. Also, if there exists no such points-to atom, then $\varphi_1 \Rightarrow \varphi_2$ is not valid. Indeed, since φ_2 does not contain existentially quantified variables, a points-to atom in φ_2 could be entailed only by a points-to atom in φ_1.

In the running example, $\texttt{select}(\psi_1^n, t \mapsto \{(s, y)\}) = \exists Y_2 : y = Y_2 \wedge \ldots \wedge t \mapsto \{(s, Y_2)\}$. For readability, we have omitted some existential variables and pure atoms.

Predicate atoms. Given an atom $a_2 = P_2(E_2, F_2, \vec{B_2})$, the call to $\texttt{select}(\varphi_1, a_2)$ first builds a sub-graph G' of $G[\varphi_1]$, which is a candidate for representing a *partial* unfolding of a_2 in φ_1, and then it checks whether the sub-heaps described by G' are well-formed wrt a_2. If this is not true or if G' is empty, then $\texttt{select}(\varphi_1, a_2)$ outputs emp. Otherwise, it outputs the formula $\exists \vec{X} : \Pi_1' \wedge \Sigma'$ where Σ' consists of all atoms represented by edges of the sub-graph G', and Π_1' contains all equalities $E_1 = E_2$ of Π_1 where either E_1 or E_2 occur in G'.

We now have a look at the construction of G' in more detail. It is based on Constraint 4. Let $\texttt{Dangling}[a_2] = \texttt{Node}(F_2) \cup \{\texttt{Node}(B) \mid B \in \overrightarrow{B_2}\}$. Notice that $\{F_2\} \cup \overrightarrow{B_2}$ are also free variables of φ_1. The sub-graph G' is defined as the union of all paths of $G[\varphi_1]$ that (1) start in the node labelled by E_2, (2) consist of edges labelled by fields in $\mathbb{F}^*_{\mapsto}(P_2)$ or predicates Q with $P_2 \prec^*_{\mathbb{P}} Q$, and (3) end either in a node from $\texttt{Dangling}[a_2]$ or in a cycle, in both cases not traversing through nodes in $\texttt{Dangling}[a_2]$. Therefore, G' does not contain edges that start in a node from $\texttt{Dangling}[a_2]$, but shall contain a path from $\texttt{Node}(E_2)$ to each node in $\texttt{Dangling}[a_2]$. In the running example, the subgraphs returned by $\texttt{select}(\psi_1^n, \texttt{nll}(x, y, z))$ and $\texttt{select}(\psi_1^n, \texttt{skl}_2(y, t))$ are highlighted in Fig. 4(c).

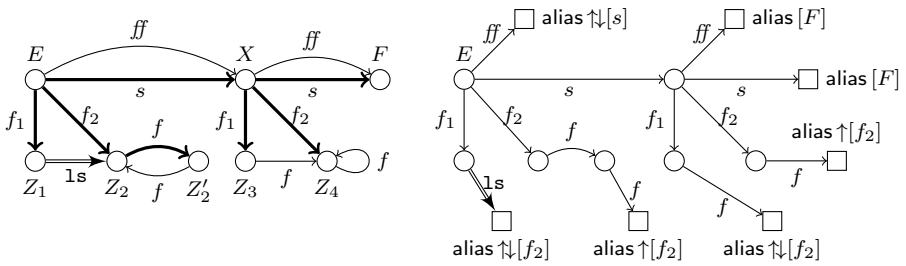
If the construction of G' succeeds, the procedure `select` checks that, in every model (S_1, H_1) of φ_1, the sub-heap (S_1, H_1') described by G' is well-formed wrt a_2, i.e., nodes of $\texttt{Dangling}[a_2]$ are not interpreted by S_1 in $ldom(H_1')$, the set of allocated locations in H_1'. For our running example, for any model of ψ_1, in the sub-heap modelled by the graph $\texttt{select}(\psi_1^n, \texttt{skl}_2(y, t))$ in Fig. 4(c), the variable t should not be (1) interpreted as an allocated location in the list segment $\texttt{skl}_2(y, Y_3)$ or (2) aliased to one of nodes labelled by Y_3 and Y_4. The well-formedness test is performed using the below proposition.

Proposition 3 (Well-formedness test) *Let a graph G' represent a sub-formula of φ_1 and $a_2 = P_2(E_2, F_2, \overrightarrow{B_2})$ be a predicate atom such that $free(G') \supseteq free(a_2)$. G' is well-formed wrt a_2 iff the following conditions hold for each variable $V \in \{F_2\} \cup \overrightarrow{B_2}$:*

1. *For every variable V' labelling the source of a points-to edge of G', it holds that $\varphi_1 \Rightarrow V \neq V'$.*
2. *For every predicate edge e included in G' that does not end in $\texttt{Node}(V)$, V is allocated in all models of $E \neq F \wedge (\varphi_1 \setminus G')$ where E and F are variables labelling the source and the destination of e, respectively, and $\varphi_1 \setminus G'$ is a formula obtained from φ_1 by deleting all* spatial *atoms represented by edges of G'.*

Proof ($\Rightarrow$) If G' is well-formed, then condition (1) is trivially satisfied. Notice that, if G' contains only one predicate edge e, it shall end in $\texttt{Dangling}[a_2]$ (by construction of G'), so condition (2) is trivially true. Otherwise, let $V \in \{F_2\} \cup \overrightarrow{B_2}$ and e be a predicate edge of G' such that $\texttt{Node}(V) \notin \texttt{Dangling}[e]$ as in condition (2). Let (S_1, H_1) be a model of φ_1 s.t. the sub-heap described by e is not empty. From Constraint 1, it follows that (S_1, H_1) is also a model of $E \neq F \wedge \varphi_1$, where E and F are the destination and target parameters of e respectively. The hypothesis implies that, for the sub-heap H_1' described by G', $S_1(V) \notin ldom(H_1')$. From the semantics of separating conjunction, $ldom(H_1) = ldom(H_1') \uplus ldom(H_1'')$, where H_1'' is the sub-heap described by $\varphi_1 \setminus G'$. Thus, $S(V)$ shall belong to $ldom(H_1'')$ because it is the only set disjoint from $ldom(H_1')$ in any model of φ_1.

($\Leftarrow$) Condition (1) guarantees that V is different from all allocated locations represented by sources of points-to edges in G'. Condition (2) guarantees that V is not interpreted as an allocated location in a list segment described by a predicate edge of G' (this trivially holds for predicate edges ending in $\texttt{Node}(V)$). If V were not allocated in some model (S_1, H_1'') of $E \neq F \wedge (\varphi_1 \setminus G')$, then one could construct a model (S_1, H_1') of G' where e would be interpreted to a non-empty list and $S(V)$



a) An SL graph G b) The tree encoding $\mathcal{T}[G]$ of G

Fig. 6 An example of encoding an SL graph into a tree

would equal an allocated location inside this list. Therefore, there would exist a model of φ_1, defined as the union of (S_1, H_1') and (S_1, H_1''), in which the heap region described by G' would not be well-formed wrt a_2. □

The following proposition provides a test for checking that variables are allocated based on checking unsatisfiability of SL formulas. Note that, by Proposition 1, unsatisfiability of formulas can be decided using the Boolean abstraction BoolAbs.

Proposition 4 *Let $\varphi ::= \exists \vec{X} : \Pi \wedge \Sigma$ be a formula and V a program variable such that $V \in pv(\varphi)$. Let V_1 and f_1 be symbols not occurring in φ. V is allocated in every model of φ iff $\exists \vec{X} : \Pi \wedge \Sigma * V \mapsto \{(f_1, V_1)\}$ is unsatisfiable.*

6 Representing SL Graphs as Trees

We define a canonical representation of SL graphs in the form of trees, which we use for checking $\Rightarrow_{sh}$. In this representation, disequality edges are ignored because they have been dealt with previously when checking entailment of pure parts.

Example: We start by explaining the main concepts of the tree encoding using the labelled graph G in Fig. 6(a), which is well-formed wrt some predicate atom $P(E, F)$ where P specifies some special kind of list segments with nested circular lists (chosen to completely illustrate all the needed issues). We assume that all nodes in G are reachable from the node labelled by E, which is guaranteed for the graphs constructed by `select` because of Property 1.

To construct a tree representation of G, we start with its spanning tree (highlighted using bold edges) built using minimal paths as in Property 6. Then, any node with at least two incoming edges, called a *join node*, is split into several copies, one for each incoming edge not contained in the spanning tree. The obtained tree is given in Fig. 6(b). In order not to lose any information, the copies of nodes are labelled with the identity of the original node, which is kept in the spanning tree. If the original node is labelled by a program variable, say x, the original node and its copies are labelled by $\mathsf{alias}\,[x]$. Otherwise, since the representation does not use node identities, we assign to every copy of the node a "routing" label describing how the copy can reach the original node using paths in the spanning tree.

For example, if a node n has the label $\mathsf{alias}\uparrow[f_2]$, this means that n is a copy of some join node m, where m is the first ancestor of n with an incoming edge labelled by f_2. Further, n labelled by $\mathsf{alias}\uparrow\downarrow[f_2]$ means that the original node m can be reached from n by going up in the tree until the first node with an outgoing edge labelled by f_2, and then down via the f_2-labelled edge. The exact definition of these labels can be found later in this section. Intuitively, a label of the form $\mathsf{alias}\uparrow[f]$ will be used when breaking loops, while a label of the form $\mathsf{alias}\uparrow\downarrow[f]$ will be used when breaking parallel paths between nodes. Due to Property 5, this set of routing labels is enough to convert an SL graph into a canonical tree representation that can entail a spatial atom from the considered fragment; for arbitrary graphs, this is not the case.

Let G be an SL graph well-formed wrt the predicate atom $P(E, F, \vec{B})$ such that all nodes of G are reachable from the node $Root$ labelled by E. An *f-edge* of an SL graph is a points-to edge labelled by f or a predicate edge labelled by $Q(\vec{Y})$ such that the minimum field in $\mathbb{F}_{\mapsto}(Q)$ wrt $\prec_{\mathbb{F}}$ is f. The tree encoding of G is computed by the procedure $\texttt{toTree}(G, P(E, F, \vec{B}))$ that consists of four consecutive steps that are presented below.

Node marking. First, `toTree` computes a mapping $\mathbb{M}$, called *node marking*, that maps each node n to a field in $\mathbb{F}$ as follows:

$$\mathbb{M}(n) ::= \begin{cases} \min_{\prec_{\mathbb{F}}}(\mathbb{F}_{\mapsto}(P)) & \text{if } n = Root, \\ \min_{\prec_{\mathbb{F}}}\{f \mid f\text{-edge enters } n\} & \text{otherwise.} \end{cases} \tag{7}$$

This means that $\mathbb{M}(n)$ is the minimum field wrt $\prec_{\mathbb{F}^*}$ among the fields of (points-to or predicate) edges entering node n. For technical reasons, we add the minimum field (wrt $\prec_{\mathbb{F}}$) in $\mathbb{F}_{\mapsto}(P)$ as the marking of node $Root$.

For any join node n not labelled by a variable in $\{E, F\} \cup \vec{B}$, the spanning tree edge is the f-edge (m, n) such that $\mathbb{M}(n) = f$; for $Root$, no incoming edge is in the spanning tree. The soundness of this construction is obtained due to Property 6, which ensures that in any model of $P(E, F, \vec{B})$, all allocated nodes are reachable via paths built using only minimum fields.

Splitting join nodes. The way join nodes are split depends on whether they are labelled by variables in $\{E, F\} \cup \vec{B}$ or not. First, a graph G' is obtained from G by replacing any edge (m, n) with n labelled by some $V \in \{E, F\} \cup \vec{B}$ by an edge (m, n') with the same label, where n' is a fresh copy of n labelled by $\mathsf{alias}\,[V]$. In our example, the node labelled with F in Fig. 6(a) is split, and we obtain three nodes labelled by $\mathsf{alias}\,[F]$ in Fig. 6(b).

Subsequently, G' is transformed into a tree by splitting the remaining join nodes as follows. Let n be a join node and (m, n) an edge not in the spanning tree of G' (and G). The edge (m, n) is replaced in the tree by an edge (m, n') with the same edge label, where n' is a fresh copy of n labelled by:

- $\mathsf{alias}\uparrow[\mathbb{M}(n)]$ if m is reachable from n in G' and n is the first predecessor of n' marked with $\mathbb{M}(n)$. In Fig. 6(a), this labelling is used on cutting the edge from Z_2' to Z_2, and substituting it by an edge to a node labelled $\mathsf{alias}\uparrow[f_2]$ in Fig. 6(b).

– $\mathsf{alias}{\uparrow\downarrow}[\mathbb{M}(n)]$ if there is a node p that is the first predecessor of n' with a (non-null) successor over edge $\mathbb{M}(n)$, and the successor is n. In Fig. 6(a), this labelling is illustrated on cutting the edge from Z_1 to Z_2, and substituting it by an edge to a node labelled $\mathsf{alias}{\uparrow\downarrow}[f_2]$ in Fig. 6(b).

If the relation between n and n' does not satisfy the constraints mentioned above, i.e. the formula does not belong to the considered fragment, the result of this step is an error represented by the $\bot$ tree. Also note that in the example in Fig. 6, edges over *ff* were split in two ways, depending on whether the target node is labelled by a variable or not. This will later be important in the construction of the TA recognising unfoldings of predicates. We denote the set of all aliasings over variables $Vars \cup LVars$ and fields $\mathbb{F}$ with ALIAS, formally, $\mathsf{ALIAS} = \{\mathsf{alias}[X] \mid X \in Vars \cup LVars\} \cup \{\mathsf{alias}{\uparrow}[f], \mathsf{alias}{\uparrow\downarrow}[f] \mid f \in \mathbb{F}\}$.

At the end of these steps, we obtain a tree with labels on edges (using fields $f \in \mathbb{F}$ or predicates $Q(\vec{B})$) and labels on nodes of the form $\mathsf{alias}[\ldots]$; the root of the tree is labelled by E.

Updating the labels. In the last step, two transformations are done on the tree. First, the labels of predicate edges are changed in order to replace each argument X from the set $\{F\} \cup \vec{B}$ by $\mathsf{alias}[X]$, and the rest of arguments by $\mathsf{alias}{\uparrow}[\mathbb{M}(n)]$ or $\mathsf{alias}{\uparrow\downarrow}[\mathbb{M}(n)]$, depending on the position of the node n labelled by X wrt the source node of the predicate edge. In the case this is not possible, the algorithm returns $\bot$.

Second, as the generated trees will be tested for membership in the language of a TA that accepts node-labelled trees only, the labels of edges are moved to the labels of their source nodes and concatenated in the order given by $\prec_{\mathbb{F}}$ (predicates in the labels are ordered according to the minimum field in their matrix).

We now formally define the structure of the output of the algorithm. Let $\mathbb{L}$ denote the set of possible node labels obtained in the previous transformation, i.e. elements of $\mathbb{F}^*$ (ordered wrt $\prec_{\mathbb{F}}$), elements of ALIAS, and predicates $P(\vec{B})$ for all $P \in \mathbb{P}$ and $\vec{B} \in \mathsf{ALIAS}^*$. Then the output of $\mathtt{toTree}(G, P(E, F, \vec{B}))$ is a tree over labels of the tree encoding, i.e. a mapping $t : \mathbb{N}^* \to \mathbb{L}$ such that $dom(t)$ is prefix-closed with the following conditions. Let $chlds(n)$ be the set $\{i \mid ni \in dom(t)\}$. Then,

– if $t(n) = f_1 \cdots f_k \in \mathbb{F}^*$, then $chlds(n) = \{1, \ldots, k\}$,
– if $t(n) = P(\vec{B})$ for some $P \in \mathbb{P}$ and $\vec{B} \in \mathsf{ALIAS}^*$, then $chlds(n) = \{1\}$, and
– if $t(n) \in \mathsf{ALIAS}$, then $chlds(n) = \emptyset$.

The following property ensures the completeness of the entailment procedure:

Proposition 5 *Let $P(E, F, \vec{B})$ be a predicate atom and G an SL graph. If the procedure $\mathtt{toTree}(G, P(E, F, \vec{B}))$ returns $\bot$, then $G \not\Rightarrow_{sh} P(E, F, \vec{B})$.*

Proof It follows from Properties 5 and 6 that a model of a predicate in our fragment can be translated into a tree using the considered aliasing relations. Therefore, if the procedure $\mathtt{toTree}(G, P(E, F, \vec{B}))$ returns $\bot$, then G can only correspond to a model of a predicate not in the considered fragment. □

7 Tree Automata Recognising Tree Encodings of SL Graphs

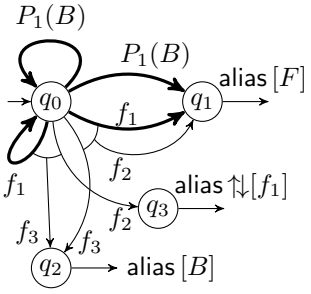
Next, we proceed to the construction of tree automata $\mathcal{A}[P]$ that recognise tree encodings of SL graphs that entail atoms of the form $P(E, F, \vec{B})$. After defining TAs, we continue with an intuitive description on a typical example, and give a full description of the TA construction later.

Definition 4 (Tree automata) A (nondeterministic) *tree automaton* (TA) recognising tree encodings of SL graphs is a tuple $\mathcal{A} = (Q, q_0, \Delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, and Δ is a finite set of transitions of the form $(q, a_1 \cdots a_n, q_1 \cdots q_n)$ or (q, a, ϵ), where $n > 0$, $q, q_1, \ldots, q_n \in Q$, a_i is an SL graph edge label (we assume them to be ordered wrt the same ordering of fields $\prec_{\mathbb{F}}$ as for tree encodings), and $a \in \mathsf{ALIAS}$. We use $q \hookrightarrow a_1(q_1), \ldots, a_n(q_n)$ to denote $(q, a_1 \cdots a_n, q_1 \cdots q_n)$ and $q \hookrightarrow a$ to denote (q, a, ϵ).

A tree encoding $t : \mathbb{N}^* \to \mathbb{L}$ is accepted by $\mathcal{A}$ if there exists a mapping $\rho : dom(t) \to Q$ such that: (i) $\rho(\epsilon) = q_0$, and (ii) for all $n \in dom(t)$, if $chlds(n) = \{1, \ldots, k\}$, then $(\rho(n), t(n), \rho(n{\cdot}1) \cdots \rho(n{\cdot}k)) \in \Delta$. The set of trees $L(\mathcal{A})$ accepted by $\mathcal{A}$ is called the *language* of $\mathcal{A}$.

7.1 Overview of the Construction

The tree automaton $\mathcal{A}[P]$ is constructed by a procedure starting from the inductive definition of P. If P does not call other predicates, the TA simply recognises the tree encodings of the SL graphs that are obtained by "concatenating" a sequence of either Gaifman graphs representing the matrix of P, $\Sigma(E, X_{\mathtt{t1}}, \vec{B})$, or predicate edges $P(E, X_{\mathtt{t1}}, \vec{B})$. In these sequences, occurrences of both types can be mixed in an arbitrary order and in an arbitrary number due to Property 4 (compositional list segments) of inductive definitions in our fragment. Intuitively, this corresponds to a partial unfolding of the predicate P in which there appear concrete segments described by points-to edges as well as (possibly multiple) segments described by predicate edges. Concatenating two Gaifman graphs means that the node labelled by $X_{\mathtt{t1}}$ in the first graph is merged with the node labelled by E in the other graph. We first illustrate this on a simplified example.



(1) $q_0 \hookrightarrow f_1(q_0), f_2(q_3), f_3(q_2)$
(2) $q_3 \hookrightarrow \mathsf{alias}\Updownarrow[f_1]$
(3) $q_2 \hookrightarrow \mathsf{alias}\,[B]$
(4) $q_0 \hookrightarrow f_1(q_1), f_2(q_1), f_3(q_2)$
(5) $q_1 \hookrightarrow \mathsf{alias}\,[F]$
(6) $q_0 \hookrightarrow P_1(B)(q_0)$
(7) $q_0 \hookrightarrow P_1(B)(q_1)$

Fig. 7 Automaton $\mathcal{A}[P_1]$

Consider a predicate $P_1(E, F, B)$ that does not invoke any other predicates and whose matrix is $\Sigma_1 ::= E \mapsto \{(f_1, X_{\mathtt{t1}}), (f_2, X_{\mathtt{t1}}), (f_3, B)\}$. The tree automaton $\mathcal{A}[P_1]$ for $P_1(E, F, B)$ has transitions given in Fig. 7. Transitions 1–3 recognise the tree encoding of the Gaifman graph of Σ_1, assuming the following total order on the fields: $f_1 \prec_{\mathbb{F}} f_2 \prec_{\mathbb{F}} f_3$. Transition 4 is used to distinguish the "last" instance of this

tree encoding, which ends in the node labelled by $\mathsf{alias}\,[F]$ accepted by Transition 5. Finally, Transitions 6 and 7 recognise predicate edges labelled by $P_1(B)$. As in the previous case, we distinguish the predicate edge that ends in the node labelled by $\mathsf{alias}\,[F]$. Note that the TA given above exhibits the simple and generic skeleton of TAs accepting tree encodings of list segments of our SL fragment: The initial state q_0 is used in a loop to traverse over an arbitrary number of folded (Transition 6) and unfolded (Transition 1) occurrences of list segments, and the state q_1 is used to recognise the end of the backbone (Transition 5). The other states (here, q_2 and q_3) are used to accept alias labels only.

When P invokes other predicates, the automaton recognises tree encodings of concatenations of more general SL graphs, obtained from $Gf[mat(P)]$ by replacing predicate edges with unfoldings of these predicates. On the level of TAs, this operation corresponds to a substitution of transitions labelled by predicates with TAs for the nested predicates. During this substitution, $\mathsf{alias}\,[\ldots]$ labels occurring in the TA for the nested predicate need to be modified, in particular, labels of the form $\mathsf{alias}\,[V]$ are substituted by the marking of $\mathtt{Node}(V)$ wrt the higher-level matrix.

7.2 Basic Algorithm for Non-Empty List Segments

We present our algorithm for translating a predicate into a TA in two steps. In this section, we start with the basic algorithm for a predicate that for each nested predicate allows *at least one* unfolding, and in the next section, we extend the construction to allow *empty* nested predicates.

Consider the definition of the matrix of a predicate $P(E, F, \vec{B})$ as given in Equations (1) and (2) in Section 2.3. The construction of the automaton $\mathcal{A}[P]$ is described in the following. To ease its presentation, let us suppose that the matrix of P is of the form $\Sigma(E, X_{\mathtt{tl}}, \vec{B}) ::= \exists \vec{Z} : E \mapsto \{(f_1, Z_1), \ldots, (f_n, Z_n)\} * \Sigma'$. Wlog, we further assume that $f_1 \prec_{\mathbb{F}} \cdots \prec_{\mathbb{F}} f_n$, i.e., f_1 is the minimum field in $\mathbb{F}_{\mapsto}(P)$.

The construction uses the SL graph of the formula that represents two unfoldings of the recursive definition of the predicate:[2]

$$\exists X_{\mathtt{tl}} : \Sigma(E, X_{\mathtt{tl}}, \vec{B}) * \Sigma(X_{\mathtt{tl}}, F, \vec{B}). \tag{8}$$

The unfolding is done twice in order to capture all markings that may appear in tree encodings that shall be recognised by $\mathcal{A}[P]$, including the ones of the nodes allocated inside the list segment (cf. the *ff* edge in the example in Fig. 6). We obtain a graph G by transforming the formula in Equation (8) to its SL graph (macros of the form $\circlearrowleft^{1+}Q[Z, \vec{Y}]$ are first expanded according to Equation (3)). In the following, we use variables $Z_1, \ldots, Z_n$ to denote existentially quantified variables from the first unfolding $\Sigma(E, X_{\mathtt{tl}}, \vec{B})$ and variables $Z'_1, \ldots, Z'_n$ to denote existentially quantified variables from the second unfolding $\Sigma(X_{\mathtt{tl}}, F, \vec{B})$.

In the following step, we get $\mathcal{T}[G]$, the tree encoding of G, and check that it is not equal to $\bot$, otherwise we abort the procedure. Notice that the variable $X_{\mathtt{tl}}$ is existentially quantified in the formula, so $\mathcal{T}[G]$ does not use the aliasing relation $\mathsf{alias}\,[X_{\mathtt{tl}}]$.

[2] Note that in the example in Fig. 7, we performed some manual minimization of the result.

Instead, a node that is a copy of the node labelled with $X_{\mathtt{t1}}$ in G needs to use either the relation $\mathsf{alias}\uparrow[f_1]$ or the relation $\mathsf{alias}\updownarrow[f_1]$, because the marking of $\mathtt{Node}(X_{\mathtt{t1}})$ is f_1. Recall also that the nodes of G labelled by parameters or existentially quantified variables are kept in the structure of $\mathcal{T}[G]$ (the tree encoding only cuts some edges and adds new nodes). Therefore, we overload the notation $\mathtt{Node}(Z)$ in the following to denote the node of $\mathcal{T}[G]$ obtained from the node of G labelled by Z.

The construction starts with an empty automaton $\mathcal{A}[P]$. It calls the procedure $\mathtt{buildTA}(P, \sigma, q_0, q_1, m_0)$, which adds states and transitions to $\mathcal{A}[P]$ to recognise tree encodings of unfoldings of the atom $P(E, F, \vec{B})$. This procedure is recursive, because it is called for all atoms $Q(U, V, \vec{W})$ inside the formula in Equation (8). The arguments of `buildTA` are the following: P is the predicate called, σ is the mapping of the formal parameters of the predicate to an aliasing relation, q_0 and q_1 are the states to be used for the source resp. the continuation of the construction, and m_0 is the marking of the state q_0. The state q_0 is chosen as the initial state of $\mathcal{A}[P]$.

Let $\sigma = \{E \mapsto \mathsf{alias}\,[E], F \mapsto \mathsf{alias}\,[F], \overrightarrow{B \mapsto \mathsf{alias}\,[B]}\}$ where $\overrightarrow{B \mapsto \mathsf{alias}\,[B]}$ denotes the set of mappings $\{B \mapsto \mathsf{alias}\,[B] \mid B \in \vec{B}\}$. The procedure $\mathtt{buildTA}(P, \sigma, q_0, q_1, f_1)$ consists of the following four steps.

I. Importing the tree encoding $\mathcal{T}[G]$. In the first step, we construct the *skeleton* of $\mathcal{A}[P]$ by taking $\mathcal{T}[G]$ and transforming it in the following way:

(a) For each node u of $\mathcal{T}[G]$, we create a unique state $q(u)$ in $\mathcal{A}[P]$, except for the nodes $\mathtt{Node}(E)$ and $\mathtt{Node}(F)$, for which we use the states q_0 and q_1 respectively.
(b) If the node u is labelled in $\mathcal{T}[G]$ with an aliasing relation $r \in \mathsf{ALIAS}$, we add the transition $q(u) \hookrightarrow \sigma(r)$ if r is of the form $\mathsf{alias}\,[B]$ for any $B \in \vec{B}$ and $q(u) \hookrightarrow r$ if r is a relation $\mathsf{alias}\,\triangledown[m]$ for $\triangledown \in \{\uparrow, \updownarrow\}$.
(c) If there is a predicate edge from u to v labelled with $Q(\vec{Y})$, we add the transition $q(u) \hookrightarrow Q(\beta'(\vec{Y}, \sigma))(q(v))$ where $\beta'(\vec{Y}, \sigma)$ changes every Y in $\vec{Y}$ according to the following rules:
   - If Y is an argument of `buildTA`, it is changed to $\sigma(Y)$;
   - if Y is an existentially quantified variable in the formula in Equation (8), m is the marking of $\mathtt{Node}(Y)$, and the relation between u and $\mathtt{Node}(Y)$ is $\mathsf{alias}\,\triangledown[m]$ for $\triangledown \in \{\uparrow, \updownarrow\}$, we change Y to $\mathsf{alias}\,\triangledown[m]$;
   - otherwise, we abort the procedure.
(d) If the node u is the source of points-to edges $e_1, \ldots, e_k$ labelled with the fields $h_1, \ldots, h_k$ respectively, assuming that $h_1 \prec_{\mathbb{F}} \cdots \prec_{\mathbb{F}} h_k$, and entering nodes $v_1, \ldots, v_k$ in this order, we add the transition $q(u) \hookrightarrow h_1(q(v_1)), \ldots, h_k(q(v_k))$. Note that this rule also creates the backbone transitions

$$q_0 \hookrightarrow f_1(q(\mathtt{Node}(X_{\mathtt{t1}}))), f_2(q(Z_2)), \ldots, f_n(q(Z_n)), \tag{9}$$

$$q(\mathtt{Node}(X_{\mathtt{t1}}))) \hookrightarrow f_1(q_1, f_2(q(Z_2')), \ldots, f_n(q(Z_n')). \tag{10}$$

(e) If the call to `buildTA` is not nested, we also add the transition $q_1 \hookrightarrow \sigma(F)$.

Observe that the created skeleton is able to accept precisely two unfoldings of the predicate P between E and F such that nested predicates are not unfolded.

II. Accepting non-empty list segments. Next, we make $\mathcal{A}[P]$ accept an arbitrary number of these unfoldings along the minimum field, i.e. f_1, of the predicate P. To do this, we add in state q_0 the following transitions:

(a) a transition that accepts exactly one unfolding:
$q_0 \hookrightarrow f_1(q_1), f_2(q(Z'_2)), \ldots, f_n(q(Z'_n))$,
(b) a looping transition that allows to insert arbitrarily many unfoldings:
$q_0 \hookrightarrow f_1(q_0), f_2(q(Z_2)), \ldots, f_n(q(Z_n))$.

III. Interleave with predicate edges. We add transitions allowing an arbitrary interleaving of folded and unfolded occurrences of the predicate P:

$$q_0 \hookrightarrow P(\sigma(\vec{B}))(q_0) \tag{11}$$

$$q_0 \hookrightarrow P(\sigma(\vec{B}))(q(\texttt{Node}(X_{\texttt{tl}}))) \tag{12}$$

$$q(\texttt{Node}(X_{\texttt{tl}})) \hookrightarrow P(\sigma(\vec{B}))(q_1). \tag{13}$$

Moreover, if the call to `buildTA` is not nested, we also add the transition

$$q_0 \hookrightarrow P(\sigma(\vec{B}))(q_1) \tag{14}$$

to accept exactly one instance of predicate P.

IV. Inserting tree automata of nested predicate edges. For each transition inserted in $\mathcal{A}[P]$ of the form: $q(\texttt{Node}(R)) \hookrightarrow Q(\vec{Y})(q(\texttt{Node}(S)))$, with $Q \neq P$ representing a nested predicate atom $Q(R, S, \vec{Y})$, we recursively call $\texttt{buildTA}(Q, \sigma', q(\texttt{Node}(R)), q(\texttt{Node}(S)), m_R)$ where $\sigma' = \{E \mapsto r_R, F \mapsto r_S, \overrightarrow{B \mapsto r_Y}\}$ (note that the definition of Q uses E, F, and $\vec{B}$) such that for any $Z \in \{R, S\} \cup \vec{Y}$:

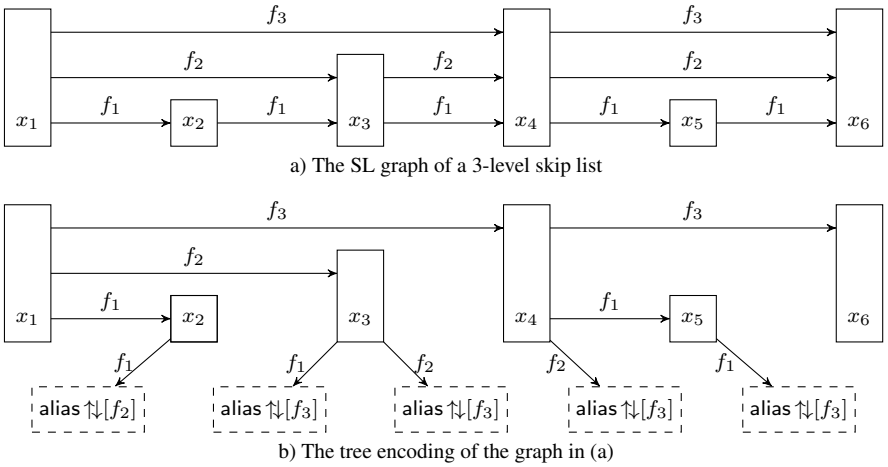
- if $Z \in \{E, F\} \cup \vec{B}$ then r_Z is $\sigma(Z)$,
- if $Z \in \vec{Z}$ (the set of existentially quantified variables in P) then r_Z is the aliasing relation between $\texttt{Node}(R)$ and $\texttt{Node}(Z)$ in $\mathcal{T}[G]$,

Note that the size of $\mathcal{A}[P]$ (number of states and transitions) is polynomial in the size of the inductive definition (number of variables and atoms) of P and of Q with $P \prec^*_{\mathbb{P}} Q$. The procedure itself is also polynomial, and the membership problem for tree automata is solvable in polynomial time (wrt the size of the input). As a consequence, we conclude that the entailment decision procedure described in this section is polynomial in the size of the input.

The following result states the correctness of the tree automata construction.

Theorem 1 *For any predicate atom $P(E, F, \vec{B})$ and any SL graph G, if the tree generated by $\texttt{toTree}(G, P(E, F, \vec{B}))$ is accepted by $\mathcal{A}[P]$, then $G \Rightarrow_{sh} P(E, F, \vec{B})$.*

Proof (Idea) The construction first creates a TA that accepts exactly two unfoldings of P (Step I). The construction then extends the TA with transitions used to accept exactly one unfolding (Step II(a)) and more than two unfoldings (Step II(b)). Step III handles acceptance of partial unfoldings of P (any interleaving of occurrences of unfoldings of P and P itself along the backbone). Finally, Step IV inserts transitions that accept all possible (non-empty) unfoldings for nested predicates. □



a) The SL graph of a 3-level skip list

b) The tree encoding of the graph in (a)

Fig. 8 Illustration of the issue with possibly empty nested list segments on $\mathtt{skl}_3$. The label of the node accessible from x_5 over f_1 (labelled with $\mathsf{alias}\Uparrow\Downarrow[f_3]$) reflects the fact that the second-level skip list from the node x_4 to the node x_6 is empty.

7.3 Extending the Basic Algorithm to Possibly Empty Nested List Segments

We now modify the above algorithm to generate TAs accepting unfoldings of P with not only *non-empty* occurrences of nested predicates, but also *empty* ones. To show the difficulties of this construction, we consider the SL graph in Fig. 8(a), which is an unfolding of the predicate atom $\mathtt{skl}_3(x_1, x_6)$. The skip list segment between nodes x_1 and x_4 contains a non-empty level-2 skip list, while the level-2 skip list between x_4 and x_6 is empty. The emptiness of the second segment requires to use the alias relation $\mathsf{alias}\Uparrow\Downarrow[f_3]$ for the node reachable from x_5 over f_1 instead of $\mathsf{alias}\Uparrow\Downarrow[f_2]$ used in the node reachable from x_2 over f_1. The TA built by the procedure `buildTA` presented in the previous section rejects such trees.

To fix this problem, apart from allowing empty occurrences of nested predicates in the TA returned by `buildTA`, we also need to extend the occurrences of aliasing relations $\mathsf{alias}\Uparrow\Downarrow[\ldots]$ to consider all combinations of empty/non-empty occurrences of nested predicates. Indeed, such aliasing relations are used to address the target node of nested predicate atoms in the tree encoding of the matrix of a predicate. The aliasing relations of the form $\mathsf{alias}\uparrow[\ldots]$ are not considered because they are used to encode the $\circlearrowleft^{1+}Q[Z, \vec{Y}]$ macro, which describes a non-empty list segment. Although it looks that we need to consider an exponential number of possibilities, we provide in the following a polynomial-time construction for the TA. The obtained TA, however, accepts trees that have the right structure but some wrong alias labels; to fix this, an additional polynomial-time check is done on the result of the membership test.

Intuitively, the new procedure has the following steps:

1. The tree encoding of G, $\mathcal{T}[G]$, is computed using $\mathtt{toTree}(G, P(E, F, \vec{B}))$.
2. Then, the TA $\mathcal{A}[P]$ is obtained using `buildTA` given in the previous section.

3. Further, $\mathcal{A}[P]$ is modified in such a way that for every predicate-labelled transition, a parallel ϵ-transition is added. Subsequently, the ϵ-transitions are removed using a standard algorithm for ϵ-transition elimination, obtaining $\mathcal{A}_r$. The automaton $\mathcal{A}_r$ accepts the same trees as $\mathcal{A}[P]$, but also trees obtained from these trees by removing some of the predicate-labelled edges (and for every such a removed edge, merging the source and target nodes together). The aliasing relations in the leaves of these trees may, however, be wrong. For example, $\mathcal{A}_r$ obtained for $\mathtt{skl}_3$ would miss the opportunity to accept trees having $\mathsf{alias}\!\uparrow\!\downarrow[f_3]$ in the nodes accessible from x_5 and x_3 through f_1 in Fig. 8(b).
4. A saturation algorithm is applied on $\mathcal{A}_r$ to obtain $\mathcal{A}'[P]$, where more aliasing transitions are introduced. Some of these added transitions do not, however, correspond to aliasings generated by the presence of empty predicate atoms.
5. For this reason, a modified membership algorithm is applied to $\mathcal{T}[G]$ and $\mathcal{A}'[P]$. It consists of first testing $\mathcal{T}[G] \in \mathcal{A}'[P]$ using a standard algorithm; if it answers false, the procedure returns false. Otherwise, the procedure checks that the aliasing transitions of $\mathcal{A}'[P]$ used in the standard membership test correspond to empty occurrences of predicates in $\mathcal{T}[G]$. If this check succeeds, the final result is true; otherwise the procedure returns false.

The above procedure, further called $\mathit{isIn}(G, P(E, F, \vec{B}))$, runs in the time polynomial to the size of G and of the inductive definitions in $\mathbb{P}$. It improves the procedure in [7], where the TA $\mathcal{A}'[P]$ is built by pumping all (exponentially many) *legal* combinations of empty predicate atoms, and membership is tested in the standard way.

We now formalize steps 3–5 of $\mathit{isIn}(G, P(E, F, \vec{B}))$. In step 3, we first create the TA $\mathcal{A}_\epsilon = (Q, q_0, \Delta_\epsilon)$ from $\mathcal{A}[P]$ in such a way, that for every transition of $\mathcal{A}[P]$ of the form $q \hookrightarrow P'(\vec{B})(r)$ representing a predicate atom $P'(\ldots)$ in the matrix of P, we add an ϵ-transition of the form $q \hookrightarrow \epsilon(r)$. Then, $\mathcal{A}_r = (Q, q_0, \Delta_r)$ is the TA obtained by applying a standard algorithm for removing epsilon transitions on $\mathcal{A}_\epsilon$.

In step 4, the saturation procedure first computes the mapping $\omega : Q \to (\mathbb{F} \cup \vec{B})^*$ such that $\omega(q)$ is the sequence of aliases that an alias accepted at q can (possibly via other aliases) refer to. More precisely, let $q \in Q$ be a state such that Δ_r contains a transition $q \hookrightarrow \mathsf{alias}\!\uparrow\!\downarrow[f]$ with $f \in \mathbb{F}$; due to our construction of $\mathcal{A}[P]$ and $\mathcal{A}_r$, there is at most one such transition from q. Let $r \hookrightarrow g_1(s_1) \cdots g_n(s_n)$ be the first transition obtained by traversing the graph of $\mathcal{A}_r$ backwards from q to q_0 satisfying the following constraints: (i) its right-hand side contains a term $g_i(s_i)$ with $g_i = f$, and (ii) Δ_r contains a transition starting in s_i and expressing an aliasing relation, i.e., it has one of the forms $s_i \hookrightarrow \mathsf{alias}\!\uparrow\!\downarrow[f']$ for $f' \in \mathbb{F}$ or $s_i \hookrightarrow \mathsf{alias}\,[X]$ for $X \in \vec{B} \setminus \{\mathtt{null}\}$ (there is at most one such transition from s_i, by the construction of $\mathcal{A}_r$). Then, we define $\omega(q) = f \cdot \omega(s_i)$. If there is no transition from s_i satisfying the above constraints but Δ_r contains $s_i \hookrightarrow \mathsf{alias}\,[\mathtt{null}]$ (if $\mathtt{null} \in \vec{B}$), then we define $\omega(q) = f \cdot \mathtt{null}$. (We treat $\mathtt{null}$ in a special way in order to to match the definition of $\mathsf{alias}\!\uparrow\!\downarrow[\ldots]$.) For any state $q \in Q$ that does not satisfy the above constraints, $\omega(q) = \epsilon$. Notice that if $\omega(q) = f_1 \cdots f_n$, then f_1 is the marking used in the alias transition from q; if $f_i \in \vec{B}$, then $i = n$, i.e., variables can only occur at the end of the sequence. The saturation returns the TA $\mathcal{A}'[P] = (Q, q_0, \Delta')$ such that $\Delta' = \Delta_r \cup \bigcup_{q \in Q} C(q)$ where $C(q) = \{q \hookrightarrow \mathsf{alias}\!\uparrow\!\downarrow[f_i] \mid f_i \in \omega(q) \cap \mathbb{F}\} \cup \{q \hookrightarrow \mathsf{alias}\,[X] \mid X \in \omega(q) \cap \vec{B}\}$.

In step 5, the modified tree membership checking algorithm creates a partial mapping $\mu : \mathcal{T}[G] \rightharpoonup (\mathbb{F} \cup \vec{B})^*$ that is defined for some leaves of $\mathcal{T}[G]$. Intuitively, μ is used to determine which nested lists of P are assumed to have empty occurrences in $\mathcal{T}[G]$. Formally, let $u \in dom(\mathcal{T}[G])$ be a leaf labelled by $\mathsf{alias}\!\uparrow\!\downarrow[f_i]$ (for $f_i \in \mathbb{F}$) or $\mathsf{alias}\,[f_i]$ (for $f_i \in \vec{B}$). If u is labelled by q in the accepting run of $\mathcal{A}'[P]$ on $\mathcal{T}[G]$, then $\mu(u) = f_1 \cdots f_i$ if $\omega(q) = f_1 \cdots f_n$ for $i \leq n$. Then, for all labelled leaves u of $\mathcal{T}[G]$, the modified membership test performs the following checks. Suppose $\mu(u) = f_1 \cdots f_i$. Then, for all $1 \leq j < i$, we test that the node v accessible via $\mathsf{alias}\!\uparrow\!\downarrow[f_j]$ references the same node as u, i.e. $\mathcal{T}[G](v) = \mathcal{T}[G](u)$. Intuitively, this validates that if an alias relation assumes that there is an empty list segment in $\mathcal{T}[G]$, there really is one. Moreover, if $f_i \in \mathbb{F}$, we also test that the node accessible via $\mathsf{alias}\!\uparrow\!\downarrow[f_i]$ is *not* an alias node. If any of the above checks fails, the procedure $isIn(G, P(E, F, \vec{B}))$ returns $false$, otherwise it returns $true$.

Theorem 2 *For any predicate atom $P(E, F, \vec{B})$ and any SL graph G, the result of $isIn(G, P(E, F, \vec{B}))$ is true iff $G \Rightarrow_{sh} P(E, F, \vec{B})$.*

Proof (Idea) From the idea of the proof of Theorem 1, we know that the TA $\mathcal{A}[P]$ accepts tree encodings of all models of P with no empty predicate occurrences. The new construction of $\mathcal{A}'[P]$ ensures that $\mathcal{A}'[P]$ will also accept any tree $\mathcal{T}$ obtained from a tree accepted by $\mathcal{A}[P]$ by allowing any predicate that occurs in it to be empty. This shows completeness of the method. On the other hand, the construction of $\mathcal{A}'[P]$ accepts trees that do not correspond to any model of P, since some nested list may jump out of the list in which it should be nested. The modified membership test ensures that such trees are rejected, re-establishing soundness of the procedure. □

8 Extension to Doubly Linked Lists

The procedure presented above can be extended to check validity of entailments between formulas using more general inductively defined predicates. In this section, we sketch the main idea for the extension to list segments that are finite nestings of both singly linked and *doubly* linked lists.

To describe doubly linked list segments, we extend the class of inductive definitions allowed by Constraint 1 (page 6) by including the following rules:

$$R_{dl}(E, F, P, L, \vec{B}) ::= E = F \wedge P = L \wedge emp \tag{15}$$

$$R_{dl}(E, F, P, L, \vec{B}) ::= \exists X_{\mathtt{tl}}, \vec{Z} : E \neq \{F\} \cup \vec{B} \wedge P \neq L \wedge \underbrace{E \mapsto \{\rho(\{X_{\mathtt{tl}}, P\} \cup \vec{V})\} * \Sigma'}_{mat(R_{dl}(E, X_{\mathtt{tl}}, P, \vec{B}))} * R_{dl}(X_{\mathtt{tl}}, F, E, L, \vec{B}) \tag{16}$$

where $\vec{V} \subseteq \vec{Z} \cup \vec{B}$ and Σ' from Constraint 3 is changed to

$$\Sigma' ::= Q(Z, U, \vec{Y}) \mid Q_{dl}(Z, U, Z_p, Z_l, \vec{Y}) \mid \circlearrowleft^{1+} Q[Z, \vec{Y}] \mid \circlearrowleft^{1+} Q_{dl}[Z, \vec{Y}] \mid \Sigma' * \Sigma' \mid emp$$

for $Z, Z_p, Z_l \in \vec{Z}$; $U \in \vec{Z} \cup \vec{B} \cup \{E, X_{\mathtt{tl}}, P\}$; $\vec{Y} \subseteq \vec{B} \cup \{E, X_{\mathtt{tl}}, P\}$; and

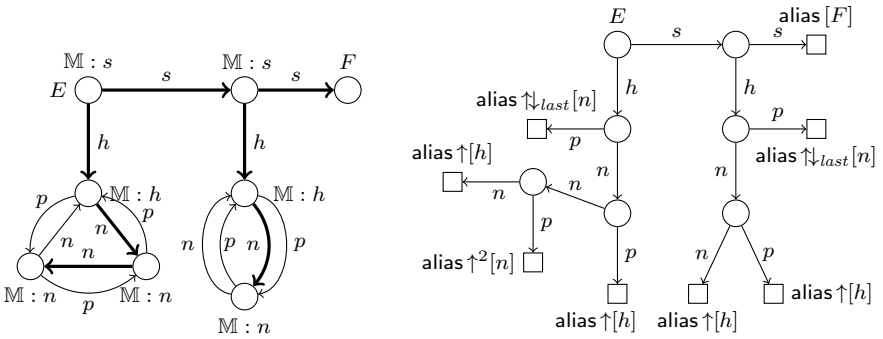


Fig. 9 Tree encodings for lists of nested cyclic doubly linked lists: (left) an SL graph that entails $\texttt{nlcdl}(E, F)$, (right) the tree encoding of the graph from the left

$$\circlearrowleft^{1+} Q[Z, \vec{Y}] \equiv \exists Z' : mat(Q(Z, Z', \vec{Y})) * Q(Z', Z, \vec{Y}),$$
$$\circlearrowleft^{1+} Q_{dl}[Z, \vec{Y}] \equiv \exists Z', Z_p : mat(Q_{dl}(Z, Z', Z_p, \vec{Y})) * Q_{dl}(Z', Z, Z, Z_p, \vec{Y}).$$

In Equation (16), variable P corresponds to the predecessor of E and variable L corresponds to the predecessor of F, i.e. the last element of the list segment. Notice that the above constraints extend the definition used for DLL segments introduced in SL by, e.g., [**?**]. For instance, to describe DLL segments starting in E, ending in L, and going to F, one can use the following inductive rule: $\texttt{dll}(E, F, P, L) ::= \exists X_{\texttt{tl}} : \ E \neq F \wedge P \neq L \wedge E \mapsto \{(n, X_{\texttt{tl}}), (p, P)\} * \texttt{dll}(X_{\texttt{tl}}, F, E, L)\big)$. To describe a singly linked list of cyclic doubly linked lists, one may use the following inductive rule: $\texttt{nlcdl}(E, F) ::= \ \exists X_{\texttt{tl}}, Z : \ E \neq F \wedge E \mapsto \{(s, X_{\texttt{tl}}), (h, Z)\} * \circlearrowleft^{1+} \texttt{dll}[Z] * \texttt{nlcdl}(X_{\texttt{tl}}, F)\big)$. (In both cases, we omitted the base rule.)

To deal with the above introduced class of inductive definitions, the main modification of our decision procedure concerns the conversion of SL graphs to trees, i.e. the *toTree* procedure described in Section 6. More precisely, we have to extend the splitting of join nodes used by this procedure as follows. Recall that, given a join node n in an SL graph G and an edge (m, n) that is not in the spanning tree of G, the splitting operation replace (m, n) by an edge (m, n') with the same edge label and n' being a fresh copy of n. For the new class of inductive definitions, we have to introduce two additional aliasing labels to describe the path from n' to n:

- $\mathsf{alias}\uparrow^2[\mathbb{M}(n)]$ will be used if m is reachable from n in G' and n is the *second* predecessor of n' marked with $\mathbb{M}(n)$. Intuitively, this label is needed to handle inner nodes of doubly linked lists, which have two incoming edges: one from their successor and one from their predecessor (see Fig. 9).
- $\mathsf{alias}\uparrow\!\!\downarrow_{last}[\mathbb{M}(n)]$ will be used if there is a node p that is the first predecessor of n' marked with $\mathbb{M}(n)$, n is reachable from p by going only via $\mathbb{M}(n)$ edges, and n has no non-alias successors with the marking $\mathbb{M}(n)$. Intuitively, the label is needed for a doubly linked cyclic list to allow referring to the predecessor of the head node of the list (see Fig. 9).

The construction of TAs from Section 7 has to be adapted too since it is based on the tree encoding of SL graphs obtained by unfolding the inductive definition of predicates to be represented. In order to generate all the aliasing relations, it turns out that

we have to consider three unfoldings (instead of two) for these predicates. Step I (importing tree encodings) of the algorithm from Section 7.2 can be extended in a trivial way for the new aliasing relations. Step IV (inserting tree automata of nested predicate edges) is adapted in a similar way to the tree encoding of SL graphs. The other steps are not modified because they are independent of the set of aliasing relations.

9 Soundness, Completeness, and Complexity

We can now finally state that Algorithm 1 is a decision procedure for our SL fragment.

Theorem 3 *Let φ_1 and φ_2 be a pair of formulas such that φ_2 is quantifier-free. Then, Algorithm 1 returns $true$ iff $\varphi_1 \Rightarrow \varphi_2$.*

Proof The first part of Algorithm 1 (until line 6) saturates the input formulas with all (non-)aliasing relations between logic variables. It follows from Proposition 1 that this transformation preserves the models of the input formulas. Thus the soundness and completeness of the algorithm is proved for the normalised formulas φ_1^n and φ_2^n.
Soundness, i.e., the fact that if Algorithm 1 returns *true*, then $\varphi_1 \Rightarrow \varphi_2$. The procedure may return *true* either at line 3 or line 15. At line 3, the test of unsatisfiability for φ_1^n is sound (by Proposition 1) and by the semantics of entailment, $\mathit{false} \Rightarrow \varphi_2$. At line 15, the result is *true* if (1) the normalised formulas φ_1^n and φ_2^n are satisfiable, (2) their pure parts satisfy $\mathit{pure}(\varphi_1^n) \Rightarrow \mathit{pure}(\varphi_2^n)$, (3) there is a mapping σ that associates to each spatial atom a_2 of φ_2^n a sub-formula $\sigma(a_2)$ of φ_1^n entailing a_2, and (4) all atoms of φ_1^n are used at most once in the image of σ (i.e. are marked once). Let $M = (S, H)$ be a model of φ_1^n. From Point (2), it follows that $M \models \mathit{pure}(\varphi_2^n)$. Let $a_2^1, \ldots, a_2^k$ be the spatial atoms of φ_2^n. From the semantics of spatial formulas and Points (3) and (4), the heap H may be partitioned in domain-disjoint heaps $H_1, \ldots, H_k$ such that they are well-formed models of sub-formulas of φ_1^n in the image of σ, i.e., for any $1 \leq i \leq k$, (S, H_i) is well-formed wrt a_i and $(S, H_i) \models \sigma(a_i)$. From the soundness of selection and Proposition 5, we obtain that $(S, H_i) \models a_i$. Thus, (S, H) is a model of φ_2^n.
Completeness, meaning that if Algorithm 1 returns *false*, then $\varphi_1 \not\Rightarrow \varphi_2$. At line 4, the procedure return *false* when φ_1^n is satisfiable and φ_2^n is not. By the soundness of the satisfiability checking, then trivially $\varphi_1 \not\Rightarrow \varphi_2$. The next result *false* is obtained at line 5 when the test of the entailment of pure parts fails. This trivially implies $\varphi_1 \not\Rightarrow \varphi_2$. In the first for loop (line 6), the result *false* is returned when a points-to atom $E_2 \mapsto \rho_2$ of φ_2^n cannot be mapped to an unmarked points-to atom a_1 of φ_1^n such that $\mathit{pure}(\varphi_1^n) \wedge a_1 \Rightarrow \mathit{pure}(\varphi_2^n) \wedge a_2$. Notice that, because φ_1^n is satisfiable, it cannot contain two different points-to atoms from the node labelled by E_2 in $G(\varphi_1^n)$. So if there is such an atom but it is already marked, i.e. used for another atom of φ_2^n, the semantics of separating conjunction implies that $\varphi_1 \not\Rightarrow \varphi_2$. If E_2 is not allocated in φ_1^n, i.e. there is no spatial atom in $G(\varphi_1^n)$ having E_2 as origin, then the entailment is also invalid because E_2 is allocated in φ_2^n. If the node labelled by E_2 in $G(\varphi_1^n)$ is the origin of a predicate atom, the entailment is also invalid because the logic cannot constraint the length of list segments to be one. In the second for loop (line 10), the *false* result is returned when the `select` procedure fails to build the sub-formula $\varphi_1^n[a_2]$ with unmarked atoms of φ_1^n such that it is well-formed wrt a_2. From

Proposition 2, φ_1^n cannot contain two disjoint sets of atoms that could correspond to the sub-formula $\varphi_1^n[a_2]$. If such a set of atoms exists, but includes marked atoms, it follows that some of the atoms are shared with the selection for another atom a_2' of φ_2^n; this is excluded by the semantics of separating conjunction, so the entailment is invalid. If such a set does not exist, no model of φ_1^n can include a model of a_2, and Property 1 implies that it cannot be a model of the predicate atom a_2. If `select` returns emp because the well-formedness test failed on the selected set of atoms, then due to the completeness of the test (Proposition 3), no well-formed model of a_2 exists in φ_1. If the selection succeeds but the algorithm proposed for $\varphi_1^n[a_2] \Rightarrow_{sh} a_2$ returns *false*, then (Theorem 2) there are well-formed models of $\varphi_1^n[a_2]$ that are not models of a_2, so the initial entailment is invalid. Finally, if there are unmarked atoms of φ_1^n, the precise semantics of our logic implies that the models of φ_1^n contain more allocated locations than the models of φ_2^n, meaning the entailment is invalid. □

The overall complexity of the decision procedure is dominated by the complexity of (a) the Boolean satisfiability and unsatisfiability checking used in the normalisation and well-formedness tests (in $select$), which are **NP** and **co-NP** complete respectively, and (b) the algorithms presented in Section 7.3 to build tree automata and check tree automaton membership, which are both polynomial-time. In conclusion, the overall complexity of the algorithm is polynomial wrt the sizes of the formulas φ_1 and φ_2 modulo an oracle for deciding (un-)satisfiability of a Boolean formula.

10 Implementation and Experimental Results

We implemented our decision procedure in a solver called SPEN (SeParation logic ENtailment). The tool takes as the input an entailment problem $\varphi_1 \Rightarrow \varphi_2$ (including the definition of the predicates used) encoded in the SMTLIB2 format. For non-valid entailments, SPEN prints the atom of φ_2 which is not entailed by a sub-formula of φ_1. The tool is based on the MINISAT solver for deciding unsatisfiability of Boolean formulas and the VATA library [14] as the tree automata backend.

Table 1 Running SPEN on entailments between well-formed formulas and atoms. Time is given in ms. The column for $\mathcal{A}[\varphi_2]$ gives the numbers of states/transitions and for $T(\varphi_1)$ the numbers of nodes/edges.

φ_2	φ_1	Time	Status	$\mathcal{A}[\varphi_2]$	$T(\varphi_1)$
`nll`	`tc1`	344	valid	6/17	7/7
	`tc2`	335	valid		7/7
	`tc3`	319	invalid		6/7
`nlcl`	`tc1`	318	valid	6/15	10/9
	`tc2`	316	valid		7/7
	`tc3`	317	invalid		6/6
$\texttt{skl}_3$	`tc1`	334	valid	80/193	7/7
	`tc2`	349	valid		8/8
	`tc3`	326	invalid		6/6
`dll`	`tc1`	358	valid	9/16	7/7
	`tc2`	324	valid		7/7
	`tc3`	322	invalid		5/5

We applied SPEN to entailment problems that use various recursive predicates. First, we considered the benchmark provided in [15], which uses only the `ls` predicate. It consists of two classes of entailment problems: the first class contains 110 problems each (split into 11 groups) generated randomly according to the rules specified in [15], whereas the second class contains 100 problems (split into 10 groups) obtained from the verification conditions generated by the tool SMALLFOOT [2]. In all experiments[3], SPEN finished in less than 1 second with the deviation of running

[3] Our experiments were performed on a PC with an Intel Core 2 Duo @2.53 GHz processor and 4 GiB DDR3 @1067 MHz running a virtual machine with Fedora 20 (64-bit).

times ± 100 ms wrt the ones reported for ASTERIX [15], the most efficient tool for deciding entailments of SL formulas with singly linked lists we are aware of.

The TA for the predicate `ls` is quite small, and so the above experiments did not evaluate much the performance of our procedure for checking entailments between formulas and atoms. For a more thorough evaluation, we further considered the experiments listed in Table 1 (among which, $\texttt{skl}_3$ required the extension discussed in Section 7.3). The full benchmark is available with our tool [8]. The entailment problems are extracted from verification conditions of operations like adding or deleting an element at the beginning, in the middle, or at the end of various kinds of list segments. Table 1 gives for each example the running time, whether the entailment is valid or invalid, and the size of the tree encoding and TA for φ_1 and φ_2, respectively. We find the resulting times quite encouraging.

Table 2 Results of SL-COMP'14. The ✗/✓/**?** columns give the numbers of wrong (✗), correct (✓), and unknown (**?**) answers.

Solver	✗	✓	?	Time [s]
`FDB_entl`				
SPEN	0	43	0	0.61
Cyclist-SL	0	19	24	141.78
SLIDE	0	0	43	0.00
SLEEK-06	1	31	11	43.65
`sll0a_entl`				
Asterix	0	292	0	2.98
SPEN	0	292	0	7.58
SLEEK-06	0	292	0	14.13
Cyclist-SL	0	55	237	11.78
`sll0a_sat`				
Asterix	0	110	0	1.06
SPEN	0	110	0	3.27
SLEEK-06	0	110	0	4.99
Cyclist-SL	55	55	0	0.55

Moreover, SPEN participated in three divisions of the first competition of separation logic solvers SL-COMP'14 [19]: division `FDB_entl` containing problems with extended acyclic lists, such as doubly linked lists, nested lists, or skip lists, and divisions `sll0a_entl` and `sll0a_sat` containing problems with singly linked lists. SPEN won division `FDB_entl` by a huge margin, solving the set containing all problems in less than a minute; further, note that SPEN is the only tool that correctly answered all problems in this division. In addition to this, SPEN was also placed second in both divisions with singly linked lists, where the first place was won by Asterix. Detailed results of this competition are in Table 2 (c.f. [19] for complete description).

11 Conclusion

This article presents a novel decision procedure for a fragment of SL with inductive predicates describing various forms of lists (singly or doubly linked, nested, circular, with skip links, etc.). The procedure is compositional in that it reduces the given entailment query to a set of simpler queries between a formula and an atom. For solving them, we proposed a novel reduction to testing membership of a tree derived from the formula in the language of a TA derived from a predicate. We implemented the procedure, and our experiments show that it has not only a favourable theoretical complexity, but also efficiently handles practical verification conditions. Moreover, when compared with other tools which competed in the first competition of separation logic solvers SL-COMP'14 [19], SPEN won the first place in one division (being by several orders of magnitude faster and even more successful in correctly deciding some problems), and the second place in two divisions.

In the future, we plan to investigate extensions of our approach to formulas with a more general Boolean structure or using more general inductive definitions. Concerning the latter, we plan to investigate whether some ideas from [12] could be used

to extend our decision procedure for entailments between formulas and atoms. From a practical point of view, apart from improving the implementation of our procedure, we plan to integrate it into a complete program analysis framework.

Acknowledgement. This work was supported by the French ANR project Vecolib, the Czech Science Foundation (project 14-11384S), and the EU/Czech IT4Innovations Excellence in Science project LQ1602.

References

1. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *Proc. of FSTTCS'04*, volume 3328 of *LNCS*, pages 97–109. Springer, 2005.
2. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proc. of FMCO'05*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.
3. James Brotherston, Carsten Fuhs, Nikos Gorogiannis, and Juan Navarro Pérez. A decision procedure for satisfiability in separation logic with inductive predicates. In *Proc. of CSL-LICS'14*, pages 25:1–25:10. ACM, 2014.
4. James Brotherston, Nikos Gorogiannis, and Rasmus Lerchedahl Petersen. A generic cyclic theorem prover. In *Proc. of APLAS'12*, volume 7705 of *LNCS*, pages 350–367. Springer, 2012.
5. Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *Proc. of FSTTCS'01*, volume 2245 of *LNCS*, pages 108–119. Springer, 2001.
6. Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In *Proc. of CONCUR'11*, volume 6901 of *LNCS*, pages 235–249. Springer, 2011.
7. Constantin Enea, Ondřej Lengál, Mihaela Sighireanu, and Tomáš Vojnar. Compositional entailment checking for a fragment of separation logic. In *Proc. of APLAS'14*, volume 8858 of *LNCS*, pages 314–333. Springer, 2014.
8. Constantin Enea, Ondřej Lengál, Mihaela Sighireanu, and Tomáš Vojnar. SPEN, 2014. Available from `https://www.irif.univ-paris-diderot.fr/~sighirea/spen`.
9. Constantin Enea, Vlad Saveluc, and Mihaela Sighireanu. Compositional invariant checking for overlaid and nested linked lists. In *Proc. of ESOP'13*, volume 7792 of *LNCS*, pages 129–148. Springer, 2013.
10. Constantin Enea, Mihaela Sighireanu, and Zhilin Wu. On automated lemma generation for separation logic with inductive definitions. In *ATVA'15*, volume 9364 of *LNCS*, pages 80–96. Springer, 2015.
11. Radu Iosif, Adam Rogalewicz, and Jiří Šimáček. The tree width of separation logic with recursive definitions. In *Proc. of CADE'13*, volume 7898 of *LNCS*, pages 21–38. Springer, 2013.
12. Radu Iosif, Adam Rogalewicz, and Tomáš Vojnar. Deciding entailments in inductive separation logic with tree automata. In *Proc. of ATVA'14*, volume 8837 of *LNCS*, pages 201–218. Springer, 2014.
13. Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. of POPL'01*, pages 14–26. ACM, 2001.
14. Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. VATA: A library for efficient manipulation of non-deterministic tree automata. In *Proc. of TACAS'12*, volume 7214 of *LNCS*, pages 79–94. Springer, 2012.
15. Juan Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *Proc. of PLDI'11*, pages 556–566. ACM, 2011.
16. Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In *Proc. of CAV'13*, volume 8044 of *LNCS*, pages 773–789. Springer, 2013.
17. Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Parthasarathy Madhusudan. Natural proofs for structure, data, and separation. In *Proc. of PLDI'13*, pages 231–242. ACM, 2013.
18. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS'02*, pages 55–74. IEEE, 2002.
19. Mihaela Sighireanu and David Cok. Report on SL-COMP'14. *JSAT, Journal of Satisfiability*, 1, 2014. Available from `http://smtcomp.sourceforge.net/2014/results-SLCOMP2.shtml`.