Compositional Entailment Checking for a Fragment of Separation Logic

FIT BUT Technical Report Series

Constantin Enea, Ondřej Lengál, Mihaela Sighireanu, and Tomáš Vojnar

Technical Report No. FIT-TR-2014-01
Faculty of Information Technology, Brno University of Technology

Last modified: October 2, 2014

Compositional Entailment Checking for a Fragment of Separation Logic

Constantin Enea[1], Ondřej Lengál[2], Mihaela Sighireanu[1], and Tomáš Vojnar[2]

[1] Univ. Paris Diderot, LIAFA CNRS UMR 7089
[2] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. We present a (semi-)decision procedure for checking entailment between separation logic formulas with inductive predicates specifying complex data structures corresponding to finite nesting of various kinds of linked lists: acyclic or cyclic, singly or doubly linked, 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. We implemented this decision procedure and tested it successfully on verification conditions obtained from programs using singly and doubly linked nested lists as well as skip lists.

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) [14,19] 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.

Usually, SL is used together with higher-order *inductive definitions* that describe 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 [1,13,17,3] 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 linked lists (acyclic or cyclic, singly or doubly linked, skip lists, etc.), which are common in practice. This fragment contains formulas of the form $\exists \overrightarrow{X}.\ \Pi \wedge \Sigma$ where 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 (semi-)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 an 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. To infer (dis)equalities implied by spatial atoms, our reduction to checking simpler entailments is based on boolean unsatisfiability checking, which is in co-NP but can usually be checked efficiently by current 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 (TA). In particular, we reduce the 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 graph, called *backbone edges*, while others are re-directed to new nodes, related to the original destination by special symbols. Roughly, such a symbol may be a variable represented by the original destination, or it may show how to reach the original destination using backbone edges only.

Our procedure is complete for formulas speaking about non-nested singly as well as doubly linked lists. Moreover, it runs in polynomial time modulo an oracle for deciding validity of a boolean formula. The procedure is incomplete for nested list structures due to not considering all possible ways in which targets of inner pointer fields of nested list predicates can be aliased. The construction can be easily extended to become complete even in such cases, but then it becomes exponential. However, even in this case, it is exponential in the size of the inductive predicates used, and not in the size of the formulas, which remains acceptable in practice.

We implemented our decision 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 it also behaves nicely in practice, at the same time offering the additional benefit of compositionality that can be exploited within larger verification frameworks caching the simpler entailment queries.

Contribution. Overall, the contribution of this paper is a novel (semi-)decision procedure for a rich class of verification conditions with singly as well as doubly linked lists, nested lists, and skip lists. As discussed in more detail in Section 9, 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 paper at the same level of efficiency as our procedure. In particular, compared with other approaches using TAs [13,12], our procedure is compositional as it uses TAs recognizing models of predicates, not models of entire formulas (further differences are discussed in the related work section). Moreover, our TAs recognize in fact formulas

that entail a given predicate, reducing SL entailment to the membership problem for TAs, not the more expensive inclusion problem as in other works.

2 Separation Logic Fragment

Let Vars be a set of *program variables*, ranged over using x, y, z, and LVars a set of *logical variables*, disjoint from Vars, ranged over using X, Y, Z. We assume that Vars contains a variable `NULL`. Also, let $\mathbb{F}$ be a set of *fields*.

We consider the fragment of separation logic whose syntax is given below:

$$
\begin{array}{lll}
x, y \in \mathit{Vars} \text{ program variables} & X, Y \in \mathit{LVars} \text{ logical variables} & E, F \ ::= \ x \mid X \\
f \in \mathbb{F} \text{ fields} & \rho \ ::= \ (f, E) \mid \rho, \rho & P \in \mathbb{P} \text{ predicates} \\
\multicolumn{3}{c}{\overrightarrow{B} \in (\mathit{Vars} \cup \mathit{LVars})^* \text{ vectors of variables}} \\
\Pi \ ::= \ E = F \mid E \neq F \mid \Pi \wedge \Pi & & \text{pure formulas} \\
\Sigma \ ::= \ \mathit{emp} \mid E \mapsto \{\rho\} \mid P(E, F, \overrightarrow{B}) \mid \Sigma * \Sigma & & \text{spatial formulas} \\
\varphi \ \triangleq \ \exists \overrightarrow{X}. \, \Pi \wedge \Sigma & & \text{formulas}
\end{array}
$$

W.l.o.g., we assume that existentially quantified logical variables have unique names. The set of program variables used in a formula φ is denoted by $pv(\varphi)$. By $\varphi(\overrightarrow{E})$ (resp. $\rho(\overrightarrow{E})$), we denote a formula (resp. a set of field-variable couples) whose set of free variables is $\overrightarrow{E}$. Given a formula φ, $\mathit{pure}(\varphi)$ denotes its pure part Π. We allow set operations to be applied on vectors. Moreover, $E \neq \overrightarrow{B}$ is a shorthand for $\bigwedge_{B_i \in \overrightarrow{B}} E \neq B_i$.

The *points-to atom* $E \mapsto \{(f_i, F_i)\}_{i \in \mathcal{I}}$ specifies that the heap contains a location E whose f_i field points to F_i, for all i. W.l.o.g., we assume that each field f_i appears at most once in a set of pairs ρ. The fragment is parameterized by a set $\mathbb{P}$ of *inductively defined predicates*; intuitively, $P(E, F, \overrightarrow{B})$ describes a possibly empty *nested list segment* delimited by its arguments, i.e., all the locations it represents are reachable from E and allocated on the heap except the locations in $\{F\} \cup \overrightarrow{B}$.

Inductively defined predicates. We consider predicates defined as

$$
\begin{array}{l}
P(E, F, \overrightarrow{B}) \triangleq (E = F \wedge \mathit{emp}) \vee \\
\qquad \left(E \neq \{F\} \cup \overrightarrow{B} \ \wedge \ \exists X_{\mathtt{tl}}. \, \Sigma(E, X_{\mathtt{tl}}, \overrightarrow{B}) \ * \ P(X_{\mathtt{tl}}, F, \overrightarrow{B})\right)
\end{array}
\tag{1}
$$

where Σ is an existentially-quantified formula, called *the matrix of* P, of the form:

$$
\begin{array}{rcl}
\Sigma(E, X_{\mathtt{tl}}, \overrightarrow{B}) & \triangleq & \exists \overrightarrow{Z}. \, E \mapsto \{\rho(\{X_{\mathtt{tl}}\} \cup \overrightarrow{V})\} * \Sigma' \qquad \text{where } \overrightarrow{V} \subseteq \overrightarrow{Z} \cup \overrightarrow{B} \text{ and} \\
\Sigma' & ::= & Q(Z, U, \overrightarrow{Y}) \mid \circlearrowleft^{1+} Q[Z, \overrightarrow{Y}] \mid \Sigma' * \Sigma' \\
& & \text{for } Z \in \overrightarrow{Z}, U \in \overrightarrow{Z} \cup \overrightarrow{B} \cup \{E, X_{\mathtt{tl}}\}, \overrightarrow{Y} \subseteq \overrightarrow{B} \cup \{E, X_{\mathtt{tl}}\}, \text{ and} \\
\circlearrowleft^{1+} Q[Z, \overrightarrow{Y}] & \triangleq & \exists Z'. \, \Sigma_Q(Z, Z', \overrightarrow{Y}) * Q(Z', Z, \overrightarrow{Y}) \text{ where } \Sigma_Q \text{ is the matrix of } Q.
\end{array}
\tag{2}
$$

The formula Σ specifies the values of the fields defined in E (using the atom $E \mapsto \{\rho(\{X_{\mathtt{tl}}\} \cup \overrightarrow{V})\}$, where the fields in ρ are constants in $\mathbb{F}$) and the (possibly cyclic) nested list segments starting at the locations $\overrightarrow{Z}$ referenced by fields of E. We assume

singly linked lists:

$$\mathtt{ls}(E,F) \triangleq lemp(E,F) \vee (E \neq F \wedge \exists X_{\mathtt{tl}}.\, E \mapsto \{(f, X_{\mathtt{tl}})\} * \mathtt{ls}(X_{\mathtt{tl}}, F))$$

lists of acyclic lists:

$$\mathtt{nll}(E,F,B) \triangleq lemp(E,F) \vee (E \neq \{F,B\} \wedge \exists X_{\mathtt{tl}}, Z.\, E \mapsto \{(s, X_{\mathtt{tl}}), (h, Z)\} * \mathtt{ls}(Z,B) * \mathtt{nll}(X_{\mathtt{tl}}, F, B))$$

lists of cyclic lists:

$$\mathtt{nlcl}(E,F) \triangleq lemp(E,F) \vee (E \neq F \wedge \exists X_{\mathtt{tl}}, Z.\, E \mapsto \{(s, X_{\mathtt{tl}}), (h, Z)\} * \circlearrowleft^{1+} \mathtt{ls}[Z] * \mathtt{nlcl}(X_{\mathtt{tl}}, F))$$

skip lists with three levels:

$$\mathtt{skl}_3(E,F) \triangleq lemp(E,F) \vee (E \neq F \wedge \exists X_{\mathtt{tl}}, Z_1, Z_2.\, E \mapsto \{(f_3, X_{\mathtt{tl}}), (f_2, Z_2), (f_1, Z_1)\} * \mathtt{skl}_1(Z_1, Z_2) * \mathtt{skl}_2(Z_2, X_{\mathtt{tl}}) * \mathtt{skl}_3(X_{\mathtt{tl}}, F))$$

$$\mathtt{skl}_2(E,F) \triangleq lemp(E,F) \vee (E \neq F \wedge \exists X_{\mathtt{tl}}, Z_1.\, E \mapsto \{(f_3, \mathtt{NULL}), (f_2, X_{\mathtt{tl}}), (f_1, Z_1)\} * \mathtt{skl}_1(Z_1, X_{\mathtt{tl}}) * \mathtt{skl}_2(X_{\mathtt{tl}}, F))$$

$$\mathtt{skl}_1(E,F) \triangleq lemp(E,F) \vee (E \neq F \wedge \exists X_{\mathtt{tl}}.\, E \mapsto \{(f_3, \mathtt{NULL}), (f_2, \mathtt{NULL}), (f_1, X_{\mathtt{tl}})\} * \mathtt{skl}_1(X_{\mathtt{tl}}, F))$$

Fig. 1. Examples of inductive definitions ($lemp(E,F) \triangleq E = F \wedge emp$).

that Σ contains a single points-to atom in order to simplify the presentation. Notice that the matrix of a predicate P does not contain applications of P.

The macro $\circlearrowleft^{1+}\ Q[Z, \overrightarrow{Y}]$ is used to represent a *non-empty* cyclic (nested) list segment on Z whose shape is described by the predicate Q.

We consider several restrictions on Σ which are defined using its *Gaifman graph* $Gf[\Sigma]$. The set of vertices of $Gf[\Sigma]$ is given by the set of free and existentially quantified variables in Σ, i.e., $\{E, X_{\mathtt{tl}}\} \cup \overrightarrow{B} \cup \overrightarrow{Z}$. The edges in $Gf[\Sigma]$ represent spatial atoms: for every (f, X) in ρ, $Gf[\Sigma]$ contains an edge from E to X labeled by f; for every predicate $Q(Z, U, \overrightarrow{Y})$, $Gf[\Sigma]$ contains an edge from Z to U labeled by Q; and for every macro $\circlearrowleft^{1+}\ Q[Z, \overrightarrow{Y}]$, $Gf[\Sigma]$ contains a self-loop on Z labeled by Q.

The first restriction is that $Gf[\Sigma]$ contains no cycles other than self-loops built solely of edges labeled by predicates. This ensures that the predicate is *precise*, i.e., for any heap, there exists at most one sub-heap on which the predicate holds. Precise assertions are very important for concurrent separation logic [10].

The second restriction requires that all the maximal paths of $Gf[\Sigma]$ start in E and end either in a self-loop or in a node from $\overrightarrow{B} \cup \{E, X_{\mathtt{tl}}\}$. This restriction ensures that (a) all the heap locations in the interpretation of a predicate are reachable from the head of the list and that (b) only the locations represented by variables in $F \cup \overrightarrow{B}$ are dangling. Moreover, for simplicity, we require that every vertex of $Gf[\Sigma]$ has at most one outgoing edge labeled by a predicate.

For example, the predicates given in Fig. 1 describe singly linked lists, lists of acyclic lists, lists of cyclic lists, and skip lists with three levels.

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. The reflexive and transitive closure of $\prec_{\mathbb{P}}$ is denoted by $\prec^*_{\mathbb{P}}$. For example, if $\mathbb{P} = \{\mathtt{skl}_1, \mathtt{skl}_2, \mathtt{skl}_3\}$, then $\mathtt{skl}_3 \prec_{\mathbb{P}} \mathtt{skl}_2$ and $\mathtt{skl}_3 \prec^*_{\mathbb{P}} \mathtt{skl}_1$.

Given a predicate P of the matrix Σ as in (2), let $\mathbb{F}_{\mapsto}(P)$ denote the set of fields f occurring in a pair (f, X) of ρ. For example, $\mathbb{F}_{\mapsto}(\mathtt{nll}) = \{s, h\}$ and $\mathbb{F}_{\mapsto}(\mathtt{skl}_3) = \mathbb{F}_{\mapsto}(\mathtt{skl}_1) = \{f_3, f_2, f_1\}$. Also, let $\mathbb{F}^*_{\mapsto}(P)$ denote the union of $\mathbb{F}_{\mapsto}(P')$ for all $P \prec^*_{\mathbb{P}} P'$. For example, $\mathbb{F}^*_{\mapsto}(\mathtt{nll}) = \{s, h, f\}$.

We assume that $\prec^*_{\mathbb{P}}$ is a partial order, i.e., there are no mutually recursive definitions in $\mathbb{P}$. Moreover, for simplicity, we assume that for any two predicates P_1 and P_2 which

$$
\begin{array}{ll}
(S,H) \models P(E,F,\overrightarrow{B}) & \text{iff there exists } k \in \mathbb{N} \text{ s.t. } (S,H) \models P^k(E,F,\overrightarrow{B}) \text{ and} \\
 & \quad ldom(H) \cap (\{S(F)\} \cup \{S(B) \mid B \in \overrightarrow{B}\}) = \emptyset \\
(S,H) \models P^0(E,F,\overrightarrow{B}) & \text{iff } (S,H) \models E = F \wedge emp \\
(S,H) \models P^{k+1}(E,F,\overrightarrow{B}) & \text{iff } (S,H) \models E \neq \{F\} \cup \overrightarrow{B} \wedge \exists X_{\mathtt{t1}}.\, \Sigma(E, X_{\mathtt{t1}}, \overrightarrow{B}) \ * \ P^k(X_{\mathtt{t1}}, F, \overrightarrow{B})
\end{array}
$$

Fig. 2. The semantics of inductive predicates.

are incomparable w.r.t. $\prec^*_{\mathbb{P}}$, it holds that $\mathbb{F}_{\mapsto}(P_1) \cap \mathbb{F}_{\mapsto}(P_2) = \emptyset$. This assumption avoids predicates named differently but having exactly the same set of models.

Semantics. Let $Locs$ be a set of *locations*. A *heap* is a pair (S, H) where $S : Vars \cup LVars \rightarrow Locs$ maps variables to locations and $H : Locs \times \mathbb{F} \rightharpoonup Locs$ 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 (resp., a variable E) is *allocated* in the heap (S, H) or that (S, H) allocates ℓ (resp., E) iff ℓ (resp., $S(E)$) belongs to $ldom(H)$.

The set of heaps satisfying a formula φ is defined by the relation $(S, H) \models \varphi$. For brevity, in Fig. 2, we give the definition of $\models$ for inductive predicates only. The complete definition can be found in App. A. Note that a heap satisfying a predicate $P(E, F, \overrightarrow{B})$ should not allocate any variable in $F \cup \overrightarrow{B}$ since these variables are considered not to be a part of its domain. A heap satisfying this property is called *well-formed w.r.t. the atom* $P(E, F, \overrightarrow{B})$. 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, $\varphi_1 \Rightarrow E = F$ (resp., $\varphi_1 \Rightarrow E \neq F$) denotes the fact that E and F are interpreted to the same location (resp., different locations) in all models of φ_1.

3 Compositional Entailment Checking

We define a procedure for reducing the problem of checking the validity of an entailment between two formulas to the problem of checking the validity of an entailment between a formula and an atom. We assume that the right-hand side of the entailment is a quantifier-free formula (which usually suffices for checking verification conditions in practice). The reduction can be extended to the general case, but it becomes incomplete.

3.1 Overview of the Reduction Procedure

We consider the problem of deciding validity of entailments $\varphi_1 \Rightarrow \varphi_2$ with φ_2 quantifier-free. We assume $pv(\varphi_2) \subseteq pv(\varphi_1)$; otherwise, the entailment is not valid.

The main steps of the reduction are given in Fig. 3. The reduction starts by a normalization step (described in Sec. 3.2), which adds to each of the two formulas all (dis-)equalities implied by spatial sub-formulas and removes all atoms $P(E, F, \overrightarrow{B})$ representing *empty* list segments, i.e., those where $E = F$ occurs in the pure part. The normalization of a formula outputs $false$ iff the input formula is unsatisfiable.

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

For the spatial parts, the procedure builds a mapping from spatial atoms of φ_2 to sub-formulas of φ_1. Intuitively, the sub-formula $\varphi_1[a_2]$ associated to an atom a_2 of φ_2,

$\varphi_1 \leftarrow \texttt{norm}(\varphi_1)$; $\varphi_2 \leftarrow \texttt{norm}(\varphi_2)$; // normalization
if $\varphi_1 = \mathit{false}$ **then return** true;
if $\varphi_2 = \mathit{false}$ **then return** false;
if $\mathit{pure}(\varphi_1) \not\Rightarrow \mathit{pure}(\varphi_2)$ **then return** false; // entailment of pure parts
foreach a_2 *: points-to atom in* φ_2 **do** // entailment of shape parts
  $\varphi_1[a_2] \leftarrow \texttt{select}(\varphi_1, a_2)$;
  if $\varphi_1[a_2] \not\Rightarrow a_2$ **then return** false;
for $P_2 \leftarrow \max_{\prec}(\mathbb{P})$ ***down to*** $\min_{\prec}(\mathbb{P})$ **do**
  forall the $a_2 = P_2(E, F, \overrightarrow{B})$ *: predicate atom in* φ_2 *s.t.* $\mathit{pure}(\varphi_1) \not\Rightarrow E = F$ **do**
    $\varphi_1[a_2] \leftarrow \texttt{select}(\varphi_1, a_2)$;
    if $\varphi_1[a_2] \not\Rightarrow_{sh} a_2$ **then return** false;
return $\texttt{isMarked}(\varphi_1)$;

Fig. 3. Compositional entailment checking ($\prec$ is any total order compatible with $\prec^*_{\mathbb{P}}$).

computed by `select`, describes the region of a heap modeled by φ_1 that should satisfy a_2. For predicate atoms $a_2 = P_2(E, F, \overrightarrow{B})$, `select` is called (in the second loop) only if there exists a model of φ_1 where the heap region that should satisfy a_2 is non-empty, i.e., $E = F$ does not occur in φ_1. In this case, `select` does also check that for any model of φ_1, the sub-heap corresponding to the atoms in $\varphi_1[a_2]$ is well-formed w.r.t. a_2 (see Sec. 3.3). This is needed since all heaps described by a_2 are well-formed.

Note that in the well-formedness check above, one cannot speak about $\varphi_1[a_2]$ alone. This is because without the rest of φ_1, $\varphi_1[a_2]$ may have models which are not well-formed w.r.t. a_2 even if the sub-heap corresponding to $\varphi_1[a_2]$ is well-formed for any model of φ_1. For example, let $\varphi_1 = \texttt{ls}(x, y) * \texttt{ls}(y, z) * z \mapsto \{(f, t)\}$, $a_2 = \texttt{ls}(x, z)$, and $\varphi_1[a_2] = \texttt{ls}(x, y) * \texttt{ls}(y, z)$. If we consider only models of φ_1, the sub-heaps corresponding to $\varphi_1[a_2]$ are all well-formed w.r.t. a_2, i.e., the location bound to z is not allocated in these sub-heaps. However, $\varphi_1[a_2]$ alone has lasso-shaped models where the location bound to z is allocated on the path between x and y.

Once $\varphi_1[a_2]$ is obtained, one needs to check that all sub-heaps modeled by $\varphi_1[a_2]$ 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). For predicate atoms a_2, a special entailment operator $\Rightarrow_{sh}$ (defined in Sec. 3.5) is used. We cannot use the usual entailment $\Rightarrow$ since, as we have seen in the example above, $\varphi_1[a_2]$ may have models which are not sub-heaps of models of φ_1. Thus, $\varphi_1[a_2] \Rightarrow_{sh} a_2$ holds iff all models of $\varphi_1[a_2]$, which are well-formed w.r.t. a_2, are also models of a_2.

If there exists an atom a_2 of φ_2, which 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 associated with two different atoms of φ_2 must not share spatial atoms. To this, the spatial atoms obtained from each application of `select` are marked and cannot be reused in the future. Note that the mapping is built by enumerating the atoms of φ_2 in a particular order: first, the points-to atoms and then the inductive predicates, in a decreasing order wrt $\prec_{\mathbb{P}}$. This is important for completeness (see Sec. 3.3).

The procedure `select` is detailed in Sec. 3.3. It returns emp if the construction of the sub-formula of φ_1 associated with the input atom fails (this implies that also the entailment $\varphi_1 \Rightarrow \varphi_2$ is not valid). If all entailments between formulas and atoms are valid, then $\varphi_1 \Rightarrow \varphi_2$ holds provided that all spatial atoms of φ_1 are marked (tested by

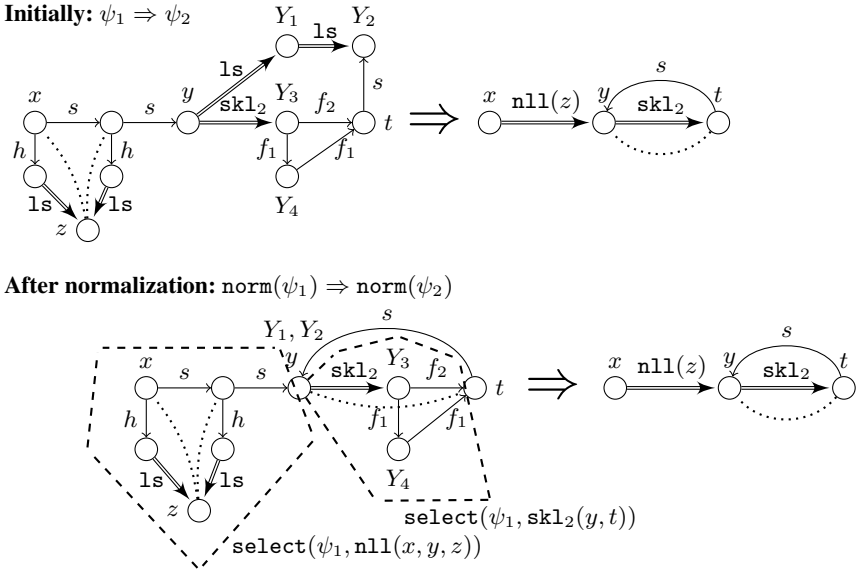


Fig. 4. An example of applying compositional entailment checking. Points-to edges are represented by simple lines, predicate edges by double lines, and disequality edges by dashed lines. For readability, we omit some of the labeling with existentially-quantified variables and some of the disequality edges in the normalized graphs.

isMarked). In Sec. 3.5, we introduce a procedure for checking entailments between a formula and a spatial atom.

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 Sec. 2). Thus, a formula φ is represented by a directed graph $G[\varphi]$ where each node represents a maximal set of variables equal w.r.t. the pure part of φ, and each edge represents a disequality $E \neq F$ or a spatial atom. Every node n is labeled by the set of variables $\texttt{Var}(n)$ it represents; for every variable E, $\texttt{Node}(E)$ denotes the node n s.t. $E \in \texttt{Var}(n)$. Next, (1) a disequality $E \neq F$ is represented by an undirected edge from $\texttt{Node}(E)$ to $\texttt{Node}(F)$, (2) a spatial atom $E \mapsto \{(f_1, E_1), \ldots, (f_n, E_n)\}$ is represented by n directed edges from $\texttt{Node}(E)$ to $\texttt{Node}(E_i)$ labeled by f_i for each $1 \leq i \leq n$, and (3) a spatial atom $P(E, F, \overrightarrow{B})$ is represented by a directed edge from $\texttt{Node}(E)$ to $\texttt{Node}(F)$ labeled by $P(\overrightarrow{B})$. Edges are referred to as disequality, points-to, or predicate edges, depending on the atom they represent. For simplicity, we may say that the graph representation of a formula is simply a formula.

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

$$\begin{aligned}\psi_1 \equiv \exists Y_1, Y_2, Y_3, Y_4, Z_1, Z_2, Z_3.\, x \neq z \wedge Z_2 \neq z \wedge \quad (3)\\ x \mapsto \{(s, Z_2), (h, Z_1)\} * Z_2 \mapsto \{(s, y), (h, Z_3)\} * \texttt{ls}(Z_1, z) * \texttt{ls}(Z_3, z) *\\ \texttt{ls}(y, Y_1) * \texttt{skl}_2(y, Y_3) * \texttt{ls}(Y_1, Y_2) *\\ Y_3 \mapsto \{(f_2, t), (f_1, Y_4)\} * Y_4 \mapsto \{(f_2, \texttt{NULL}), (f_1, t)\} * t \mapsto \{(s, Y_2)\}\\ \psi_2 \equiv y \neq t \wedge \texttt{nll}(x, y, z) * \texttt{skl}_2(y, t) * t \mapsto \{(s, y)\} \quad (4)\end{aligned}$$

The graph representations of these formulas are drawn in the top part of Fig. 4.

3.2 Normalization

To infer the implicit (dis-)equalities in a formula, we adapt the boolean abstraction proposed in [9] for our logic. Therefore, given a formula φ, we define an equisatisfiable boolean formula $\mathsf{BoolAbs}[\varphi]$ in CNF over a set of boolean variables containing the boolean variable $[E = F]$ for every two variables E and F occuring in φ and the boolean variable $[E, a]$ for every variable E and spatial atom a of the form $E \mapsto \{\rho\}$ or $P(E, F, \overrightarrow{B})$ in φ. 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.

Given $\varphi \triangleq \exists \overrightarrow{X}.\, \Pi \wedge \Sigma$, $\mathsf{BoolAbs}[\varphi] \triangleq F(\Pi) \wedge F(\Sigma) \wedge F_{=} \wedge F_{*}$ where $F(\Pi)$ and $F(\Sigma)$ encode the atoms of φ (using $\oplus$ to denote xor), $F_{=}$ encodes reflexivity, symmetry, and transitivity of equality, and F_{*} encodes the semantics of the separating conjunction:

$$F(\Pi) \triangleq \bigwedge_{E=F \in \Pi} [E = F] \;\wedge\; \bigwedge_{E \neq F \in \Pi} \neg[E = F] \qquad F(\Sigma) \triangleq \bigwedge_{a = E \mapsto \{\rho\} \in \Sigma} [E, a] \;\wedge\; \bigwedge_{a = P(E,F,\overrightarrow{B}) \in \Sigma} [E, a] \oplus [E = F]$$

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

$$F_{*} \triangleq \bigwedge_{\substack{E,\, F \text{ variables in } \varphi \\ a,\, a' \text{different atoms in } \Sigma}} ([E = F] \wedge [E, a]) \Rightarrow \neg[F, a']$$

For the formula ψ_1 in our running example (Eq. 3), $\mathsf{BoolAbs}[\psi_1]$ 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. $[Y_3, Y_3 \mapsto \{(f_1, Y_4), (f_2, t)\}]$ and $[t, t \mapsto \{(s, Y_2)\}]$, encoding points-to atoms,
3. $([y = t] \wedge [t, t \mapsto \{(s, Y_2)\}]) \Rightarrow \neg[y, \mathtt{skl}_2(y, Y_3)]$, which encodes the separating conjunction between $t \mapsto \{(s, Y_2)\}$ and $\mathtt{skl}_2(y, Y_3)$,
4. $([Y_3 = t] \wedge [t, t \mapsto \{(s, Y_2)\}]) \Rightarrow \neg[Y_3, Y_3 \mapsto \{(f_1, Y_4), (f_2, t)\}]$, which encodes the separating conjunction between $t \mapsto \{(s, Y_2)\}$ and $Y_3 \mapsto \{(f_1, Y_4), (f_2, t)\}$.

Proposition 1. *Let φ be a formula. Then, $\mathsf{BoolAbs}[\varphi]$ is equisatisfiable 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 example, $\mathsf{BoolAbs}[\psi_1] \Rightarrow \neg[y = t]$, which is a consequence of the sub-formulas we have given above together with $F_{=}$.

If $\mathsf{BoolAbs}[\varphi]$ is unsatisfiable, then the output of $\mathtt{norm}(\varphi)$ is *false*. Otherwise, the output of $\mathtt{norm}(\varphi)$ is the formula φ' obtained from φ by (1) adding all (dis-)equalities implied by $\mathsf{BoolAbs}[\varphi]$ and (2) removing all predicates $P(E, F, \overrightarrow{B})$ s.t. $E = F$ occurs in the pure part. For example, the normalizations of ψ_1 and ψ_2 are given in the bottom part of Fig. 4. Note that the `ls` atoms reachable from y are removed because $\mathsf{BoolAbs}[\psi_1] \Rightarrow [y = Y_1]$ and $\mathsf{BoolAbs}[\psi_1] \Rightarrow [Y_1 = Y_2]$.

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

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

If we assume for contradiction that $\mathtt{norm}(\varphi)$ contains two such sets of atoms, then, by the semantics of the separating conjunction, $\varphi \Rightarrow E = F$ where E and F label n and n', respectively. Therefore, $\mathtt{norm}(\varphi)$ does not include all equalities implied by φ, which contradicts its definition.

3.3 Selection of Spatial Atoms

Points-to atoms. Let $\varphi_1 \triangleq \exists \overrightarrow{X}. \Pi_1 \wedge \Sigma_1$ be a normalized formula. The procedure $\texttt{select}(\varphi_1, E_2 \mapsto \{\rho_2\})$ outputs the sub-formula $\exists \overrightarrow{X}. \Pi_1 \wedge E_1 \mapsto \{\rho_1\}$ s.t. $E_1 = E_2$ occurs in Π_1 if it exists, or emp otherwise. The procedure $\texttt{select}$ is called only if φ_1 is satisfiable and consequently, 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, $\texttt{select}(\psi_1, t \mapsto \{(s, y)\}) = \exists Y_2. y = Y_2 \wedge \ldots \wedge t \mapsto \{(s, Y_2)\}$ (we have omitted some existential variables and pure atoms).

Predicate atoms. Given an atom $a_2 = P_2(E_2, F_2, \overrightarrow{B_2})$, $\texttt{select}(\varphi_1, a_2)$ builds a sub-graph G' of $G[\varphi_1]$, and then it checks whether the sub-heaps described by G' are well-formed w.r.t. a_2. If this is not true or if G' is empty, then it outputs emp. Otherwise, it outputs the formula $\exists \overrightarrow{X}. \Pi_1 \wedge \Sigma'$ where Σ' consists of all atoms represented by edges of the sub-graph G'. Let $\texttt{Dangling}[a_2] = \texttt{Node}(F_2) \cup \{\texttt{Node}(B) \mid B \in \overrightarrow{B_2}\}$.

The sub-graph G' is defined as the union of all paths of $G[\varphi_1]$ that (1) consist of edges labeled by fields in $\mathbb{F}^*_{\mapsto}(P_2)$ or predicates Q with $P_2 \prec^*_{\mathbb{P}} Q$, (2) start in the node labeled by E_2, and (3) end either in a node from $\texttt{Dangling}[a_2]$ or in a cycle, in which case they must not traverse nodes in $\texttt{Dangling}[a_2]$. The paths in G' that end in a node from $\texttt{Dangling}[a_2]$ must not traverse other nodes from $\texttt{Dangling}[a_2]$. Therefore, G' does not contain edges that start in a node from $\texttt{Dangling}[a_2]$. The instances of G' for $\texttt{select}(\psi_1, \texttt{nll}(x, y, z))$ and $\texttt{select}(\psi_1, \texttt{skl}_2(y, t))$ are emphasized in Fig. 4.

Next, the procedure $\texttt{select}$ checks that in every model of φ_1, the sub-heap described by G' is well-formed w.r.t. a_2. Intuitively, this means that all cycles in the sub-heap are explicitly described in the inductive definition of P_2. For example, if $\varphi_1 = \texttt{ls}(x, y) * \texttt{ls}(y, z)$ and $\varphi_2 = a_2 = \texttt{ls}(x, z)$, then the graph G' corresponds to the entire formula φ_1 and it may have lasso-shaped models (z may belong to the path between x and y) that are not well-formed w.r.t. $\texttt{ls}(x, z)$ (whose inductive definition describes only acyclic heaps). Therefore, the procedure $\texttt{select}$ returns emp, which proves that the entailment $\varphi_1 \Rightarrow \varphi_2$ does not hold. For our running example, for any model of ψ_1, in the sub-heap modeled by the graph $\texttt{select}(\psi_1, \texttt{skl}_2(y, t))$ in Fig. 4, 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 labeled by Y_3 and Y_4.

The well-formedness test is equivalent to the fact that for every variable $V \in \{F_2\} \cup \overrightarrow{B_2}$ and every model of φ_1, the interpretation of V is different from all allocated locations in the sub-heap described by G'. This is in turn equivalent to the fact that for every variable $V \in \{F_2\} \cup \overrightarrow{B_2}$, the two following conditions hold:

1. 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 labeling the source and the destination of e, respectively, and $\varphi_1 \setminus G'$ is obtained from φ_1 by deleting all *spatial* atoms represented by edges of G'.
2. For every variable V' labeling the source of a points-to edge of G', $\varphi_1 \Rightarrow V \neq V'$.

The first condition 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 was not allocated in some model (S, H_1) of $E \neq$

$F \wedge (\varphi_1 \setminus G')$, then one could construct a model (S, H_2) of G' where e would be interpreted to a non-empty list and $S(V)$ would equal an allocated location inside this list. Therefore, there would exist a model of φ_1, defined as the union of (S, H_1) and (S, H_2), in which the heap region described by G' would not be well-formed w.r.t. a_2.

For example, in the graph $\texttt{select}(\psi_1, \texttt{skl}_2(y, t))$ in Fig. 4, t is not interpreted as an allocated location in the list segment $\texttt{skl}_2(y, Y_3)$ since t is allocated (due to the atom $t \mapsto \{(s, Y_2)\}$) in all models of $y \neq Y_3 \wedge (\psi_1 \setminus \texttt{select}(\psi_1, \texttt{skl}_2(y, t)))$.

To check that variables are allocated, we use the following property: given a formula $\varphi \triangleq \exists \overrightarrow{X} . \Pi \wedge \Sigma$, a variable V is allocated in every model of φ iff $\exists \overrightarrow{X} . \Pi \wedge \Sigma * V \mapsto \{(f, V_1)\}$ is unsatisfiable. Here, we assume that f and V_1 are not used in φ. Note that, by Prop. 1, unsatisfiability can be decided using the boolean abstraction $\mathsf{BoolAbs}$.

The second condition guarantees that V is different from all allocated locations represented by sources of points-to edges in G'. For the graph $\texttt{select}(\psi_1, \texttt{nll}(x, y, z))$ in Fig. 4, the variable z must be different from all existential variables labeling a node which is the source of a points-to edge. These disequalities appear explicitly in the formula. By Prop. 1, $\varphi_1 \Rightarrow V \neq V'$ can be decided using the boolean abstraction.

3.4 Soundness and Completeness

The following theorem states that the procedure given in Fig. 3 is sound and complete. The soundness is a direct consequence of the semantics. The completeness is a consequence of Prop. 1 and 2. In particular, Prop. 2 implies that the sub-formula returned by $\texttt{select}(\varphi_1, a_2)$ is the only one that can describe a heap region satisfying a_2.

Theorem 1. *Let φ_1 and φ_2 be two formulas s.t. φ_2 is quantifier-free. Then, $\varphi_1 \Rightarrow \varphi_2$ iff the procedure in Fig. 3 returns $true$.*

3.5 Checking Entailments between a Formula and an Atom

Given a formula φ and an atom $P(E, F, \overrightarrow{B})$, we define a procedure for checking that $\varphi \Rightarrow_{sh} P(E, F, \overrightarrow{B})$, which works 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 $P(E, F, \overrightarrow{B})$ is used to define a TA $\mathcal{A}[P]$ s.t. $\mathcal{T}[\varphi]$ belongs to the language of $\mathcal{A}[P]$ only if $\varphi \Rightarrow_{sh} P(E, F, \overrightarrow{B})$. Notice that we do not require the reverse implication in order to keep the size of $\mathcal{A}[P]$ polynomial in the size of the inductive definition of P. Thus, $\mathcal{A}[P]$ does not recognize the tree representations of all formulas φ s.t. $\varphi \Rightarrow_{sh} P(E, F, \overrightarrow{B})$. The transformation of graphs into trees is presented in Sec. 4 while the definition of the TA is introduced in Sec. 5. In the latter section, we also discuss how to obtain a complete method by generating a TA $\mathcal{A}[P]$ of an exponential size.

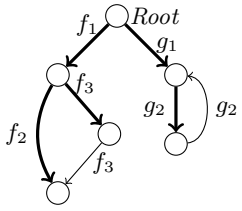
4 Representing SL Graphs as Trees

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

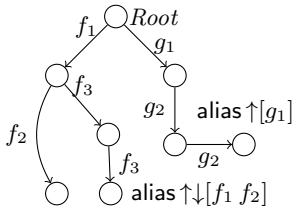
We start by explaining the main concepts of the tree encoding using the generic labeled graph in Fig. 5(a). We consider a graph G where all nodes are reachable from a

distinguished node called $Root$ (this property is satisfied by all SL graphs returned by the `select` procedure). To construct a tree representation of G, we start with its spanning tree (emphasized using bold edges) and proceed with splitting any node with at least two incoming edges, called a *join node*, into several copies, one for each incoming edge not contained in the spanning tree. The obtained tree is given in Fig. 5(b).

Not to loose any information, the copies of nodes should be labeled with the identity of the original node, which is kept in the spanning tree. However, since the representation does not use node identities, we label every original node with a representation of the path from $Root$ to this node in the spanning tree, and we assign every copied node a label describing how it can reach the original node in the spanning tree. For example, if a node n has the label $\mathsf{alias}\uparrow[g_1]$, this denotes the fact that n is a copy of some join node, which is the first ancestor of n in the spanning tree that is reachable from $Root$ by a path formed of a (non-empty) sequence of g_1 edges. Further, n labelled by $\mathsf{alias}\uparrow\downarrow[f_1\ f_2]$ denotes roughly that (1) the original node is reachable from $Root$ by a path formed by a (non-empty) sequence of f_1 edges followed by a (non-empty) sequence of f_2 edges, and (2) the original node can be reached from n by going up in the tree until the first node that is labelled by a prefix of $f_1\ f_2$ and then down until the first node labelled with $f_1\ f_2$. The exact definition of these labels can be found later in this section. In general, a label of the form $\mathsf{alias}\uparrow[\ldots]$ will be used when breaking loops while a label of the form $\mathsf{alias}\uparrow\downarrow[\ldots]$ will be used when breaking parallel paths between nodes. Moreover, if the original node is labeled by a variable, e.g., x, then we will use a label of the form $\mathsf{alias}\,[x]$. This set of labels is enough to obtain a tree representation from SL graphs that can entail a spatial atom from the considered fragment; for arbitrary graphs, this is not the case.



(a) A labeled graph G



(b) A tree representation of G

Fig. 5. The tree representation of a generic graph.

When applying this construction to an SL graph, the most technical part consists in defining the spanning tree. Based on the inductive definition of predicates, we consider a total order on fields $\prec_{\mathbb{F}}$ that is extended to sequences of fields, $\prec_{\mathbb{F}^*}$, in a lexicographic way. Then, the spanning tree is defined by the set of paths labeled by sequences of fields which are minimum according to the order $\prec_{\mathbb{F}^*}$.

Intuitively, the order $\prec_{\mathbb{F}}$ reflects the order in which the unfolding of the inductive definition of P is done: (1) Fields used in the atom $E \mapsto \rho$ of the matrix of P are ordered before fields of any other predicate called by P. (2) Fields appearing in ρ and going "one-step forward" (i.e., occurring in a pair (f, X_{tl})) are ordered before fields going "down" (i.e., occurring in a pair (f, Z) with $Z \in \overrightarrow{Z}$), which are ordered before fields going to the "border" (i.e., occurring in a pair (f, X) with $X \in \overrightarrow{B}$).

Formally, given a predicate P with the matrix Σ as in (2), we split the set $\mathbb{F}_{\mapsto}(P)$ in three disjoint sets: (a) $\mathbb{F}_{\mapsto X_{\mathtt{t1}}}(P)$ is the set of fields f occurring in a pair (f, X_{tl})

of ρ, (b) $\mathbb{F}_{\mapsto \overrightarrow{Z}}(P)$ the set of fields f occurring in a pair (f, Z) of ρ with $Z \in \overrightarrow{Z}$, and (c) $\mathbb{F}_{\mapsto \overrightarrow{B}}(P)$ the set of fields f occurring in a pair (f, X) of ρ with $X \in \overrightarrow{B}$. Then, we assume that there exists a total order $\prec_{\mathbb{F}}$ on fields s.t., for all P, P_1, P_2 in $\mathbb{P}$:

$$\forall f_1 \in \mathbb{F}_{\mapsto X_{\mathtt{t1}}}(P)\ \forall f_2 \in \mathbb{F}_{\mapsto \overrightarrow{Z}}(P)\ \forall f_3 \in \mathbb{F}_{\mapsto \overrightarrow{B}}(P).\ f_1 \prec_{\mathbb{F}} f_2 \prec_{\mathbb{F}} f_3 \text{ and}$$
$$(f_1 \in \mathbb{F}_{\mapsto}(P_1) \wedge f_2 \in \mathbb{F}_{\mapsto}(P_2) \wedge f_1 \neq f_2 \wedge P_1 \prec_{\mathbb{P}} P_2) \Rightarrow f_1 \prec_{\mathbb{F}} f_2.$$

For example, if $\mathbb{P} = \{\mathtt{nll}, \mathtt{ls}\}$ or $\mathbb{P} = \{\mathtt{nlcl}, \mathtt{ls}\}$, then $s \prec_{\mathbb{F}} h \prec_{\mathbb{F}} f$; and if $\mathbb{P} = \{\mathtt{skl}_2, \mathtt{skl}_1\}$, then $f_2 \prec_{\mathbb{F}} f_1$. The order $\prec_{\mathbb{F}}$ is extended to a lexicographic order $\prec_{\mathbb{F}^*}$ over sequences in $\mathbb{F}^*$.

An *f-edge* of an SL graph is a points-to edge labeled by f or a predicate edge labeled by $P(\overrightarrow{N})$ s.t. the minimum field in $\mathbb{F}_{\mapsto}(P)$ w.r.t. $\prec_{\mathbb{F}}$ is f.

Let G be an SL graph and $P(E, F, \overrightarrow{B})$ an atom for which we want to prove that $G \Rightarrow_{sh} P(E, F, \overrightarrow{B})$. We assume that all nodes of G are reachable from the node $Root$ labeled by E, which is ensured when G is constructed by `select`. The tree encoding of G is computed by the procedure $\mathtt{toTree}(G, P(E, F, \overrightarrow{B}))$ that consists of four consecutive steps that are presented below (see also App. B).

Node marking. First, `toTree` computes a mapping $\mathbb{M}$, called *node marking*, which defines the spanning tree of G. Intuitively, for each node n, $\mathbb{M}(n)$ is the sequence of fields labeling a path reaching n from $Root$ that is minimal w.r.t. $\prec_{\mathbb{F}^*}$. Formally, let π be a path in G starting in $Root$ and consisting of the sequence of edges $e_1\, e_2\, \ldots e_n$. The *labeling of* π, denoted by $\mathbb{L}(\pi)$, is the sequence of fields $f_1\, f_2 \ldots\, f_n$ s.t. for all i, e_i is an f_i-edge. The node marking is defined by

$$\forall n \in G \qquad \mathbb{M}(n) \triangleq \mathit{Reduce}(\mathit{min}_{\prec_{\mathbb{F}}}(\mathbb{F}_{\mapsto}(P))\ \mathbb{L}_{\min}(n)), \tag{5}$$
$$\mathbb{L}_{\min}(n) \triangleq \mathit{min}_{\prec_{\mathbb{F}^*}}\{\mathbb{L}(\pi) \mid \mathit{Root} \xrightarrow{\pi} n\} \tag{6}$$

where $Reduce$ rewrites the sub-words of the form f^+ to f, for any field f. For technical reasons, we add the minimum field (w.r.t. $\prec_{\mathbb{F}}$) in $\mathbb{F}_{\mapsto}(P)$ at the beginning of all $\mathbb{M}(n)$.

Fig. 6(b)–(c) depicts two graphs and the markings of their nodes. (For readability, we omit the markings of the nodes labeled by y and t.)

Splitting join nodes. The join nodes are split in two consecutive steps, denoted as `splitLabeledJoin` and `splitJoin`, depending on whether they are labeled by variables in $\{E, F\} \cup \overrightarrow{B}$ or not. In both cases, only the edges of the spanning tree are kept in the tree, the other edges are redirected to fresh copies labeled by some $\mathsf{alias}\,[..]$.

For any join node n, the spanning tree edge is the f-edge (m, n) such that $\mathit{Reduce}(\mathbb{M}(m)\, f) = \mathbb{M}(n)$, i.e., (m, n) is at the end of the minimum path leading to n. (For $Root$, all incoming edges are not in the spanning tree.)

In `splitLabeledJoin`, a graph G' is obtained by replacing in G any edge (m, n) such that n is labeled by some $V \in \{E, F\} \cup \overrightarrow{B}$ and (m, n) is not in the spanning tree by an edge (m, n') with the same label, where n' is a fresh copy of n labeled by $\mathsf{alias}\,[V]$. Moreover, for uniformity, all (even non-join) nodes labeled by a variable $V \in F \cup \overrightarrow{B}$ are labeled by $\mathsf{alias}\,[V]$ in G'. Fig. 6(a) gives the output graph of `splitLabeledJoin` on the SL graphs returned in our running example by $\mathtt{select}(\psi_1, \mathtt{nll}(x, y, z))$ and $\mathtt{select}(\psi_1, \mathtt{skl}_2(y, t))$.

Subsequently, `splitJoin` builds from G' a tree by splitting unlabeled 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 labeled by:

- $\mathsf{alias}{\uparrow}[\mathbb{M}(n)]$ if m is reachable from n and all predecessors of m (by a simple path) marked by $\mathbb{M}(n)$ are also predecessors of n. Intuitively, this label is used to break loops, and it refers to the closest predecessor of n' having the given marking. The use of this labeling is illustrated in Fig. 6(b).
- $\mathsf{alias}{\uparrow}{\downarrow}[\mathbb{M}(n)]$ if there is a node p which is a predecessor of m s.t. all predecessors of m that have a unique successor marked by $\mathbb{M}(n)$ are also predecessors of p, and n is the unique successor of p marked by $\mathbb{M}(n)$. Intuitively, this transformation is used to break multiple paths between p and n as illustrated in Fig. 6(c).[3]

If the relation between n and n' does not satisfy the constraints mentioned above, the result of `splitJoin` is an error, i.e., the $\bot$ tree.

At the end of these steps, we obtain a tree with labels on edges (using fields $f \in \mathbb{F}$ or predicates $Q(\overrightarrow{B})$) and labels on nodes of the form $\mathsf{alias}\,[..]$; its root is labeled 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 different from $\{F\} \cup \overrightarrow{B}$ by a label $\mathsf{alias}{\uparrow}[\mathbb{M}(n)]$ or $\mathsf{alias}{\uparrow}{\downarrow}[\mathbb{M}(n)]$, which describes the position of the node n labeled by X w.r.t. the node of G labeled by E.

Finally, as the generated trees will be tested for membership in the language of a TA which 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).

The following property ensures the soundness of the entailment procedure:

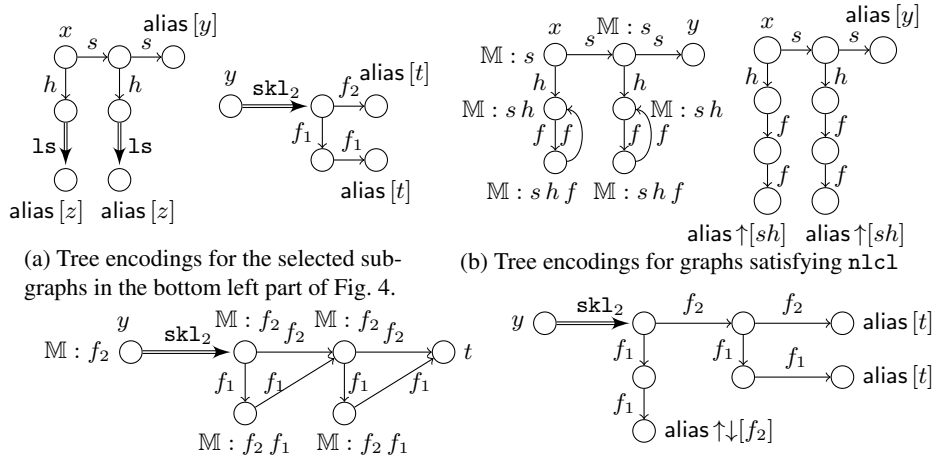
Proposition 3. *Let $P(E, F, \overrightarrow{B})$ be an atom and G an SL graph. If $\mathtt{toTree}(G, P(E, F, \overrightarrow{B})) = \bot$, then $G \not\Rightarrow P(E, F, \overrightarrow{B})$.*

5 Tree Automata Recognizing Tree Encodings of SL Graphs

Next, we proceed to the construction of tree automata $\mathcal{A}[P(E, F, \overrightarrow{B})]$ that recognize tree encodings of SL graphs that imply atoms of the form $P(E, F, \overrightarrow{B})$. Due to space constraints, we cannot provide a full description of the TA construction (which we give in App. C). Instead, we give an intuitive description only and illustrate it on two typical examples (for now, we leave our running examples, TAs for which are given in App. D).

Tree automata. A (non-deterministic) *tree automaton* recognizing tree encodings of SL graphs is a tuple $\mathcal{A} = (Q, q_0, \Delta)$ where Q is a set of states, $q_0 \in Q$ is the initial state, and Δ is a set of transition rules of the form $q \hookrightarrow a_1(q_1), \ldots, a_n(q_n)$ or $q \hookrightarrow a$, 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

[3] The combination of up and down arrows in the label corresponds to the need of going up and then down in the resulting tree—whereas in the previous case, it suffices to go up only.



(a) Tree encodings for the selected subgraphs in the bottom left part of Fig. 4.

(b) Tree encodings for graphs satisfying `nlcl`

(c) Tree encodings for graphs satisfying $\texttt{skl}_2$

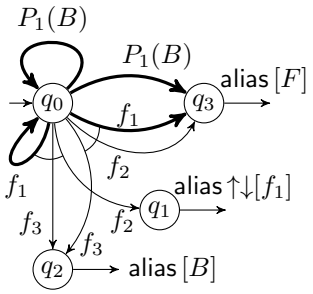
Fig. 6. Tree encodings.

w.r.t. the ordering of fields as for tree encodings), and a is $\mathsf{alias}{\uparrow}[m]$, $\mathsf{alias}{\uparrow\downarrow}[m]$, or $\mathsf{alias}\,[V]$. The set of trees $L(\mathcal{A})$ recognized by $\mathcal{A}$ is defined as usual.

Definition of $\mathcal{A}[P(E, F, \overrightarrow{B})]$. The tree automaton $\mathcal{A}[P(E, F, \overrightarrow{B})]$ is defined starting from the inductive definition of P. If P does not call other predicates, the TA simply recognizes the tree encodings of the SL graphs that are obtained by "concatenating" a sequence of Gaifman graphs representing the matrix $\Sigma(E, X_{\texttt{tl}}, \overrightarrow{B})$ and predicate edges $P(E, X_{\texttt{tl}}, \overrightarrow{B})$. In these sequences, occurrences of the Gaifman graphs representing the matrix and the predicate edges can be mixed in an arbitrary order and in an arbitrary number. 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 labeled by $X_{\texttt{tl}}$ in the first graph is merged with the node labeled by E in the other graph. This is illustrated on the following example.

Consider a predicate $P_1(E, F, B)$ that does not call other predicates and that has the matrix $\Sigma_1 \triangleq E \mapsto \{(f_1, X_{\texttt{tl}}), (f_2, X_{\texttt{tl}}), (f_3, B)\}$. The tree automaton $\mathcal{A}_1$ for $P_1(E, F, B)$ has transition rules given in Fig. 7. Rules (1)–(3) recognize the tree encoding of the Gaifman graph of Σ_1, assuming the following total order on the fields: $f_1 \prec_{\mathbb{F}} f_2 \prec_{\mathbb{F}} f_3$. Rule (4) is used to distinguish the "last" instance of this tree encoding, which ends in the node labeled by $\mathsf{alias}\,[F]$ accepted by Rule (5). Finally, Rules (6) and (7) recognize predicate edges labeled by $P_1(B)$. As in the previous case, we distinguish the predicate edge that ends in the node labeled by $\mathsf{alias}\,[F]$.

Note that the TA given above exhibits the simple and generic skeleton of TAs accepting tree encodings of list



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

Fig. 7. $\mathcal{A}[P_1(E, F, B)]$

segments defined in our SL fragment: The initial state q_0 is used in a loop to traverse over an arbitrary number of folded (Rule 6) and unfolded (Rule 1) occurrences of the list segments, and the state q_3 is used to recognize the end of the backbone (Rule 5). The other states (here, q_2) are used to accept alias labels only. The same skeleton can be observed in the TA recognizing tree encodings of singly linked lists, which is given in Fig. 8.

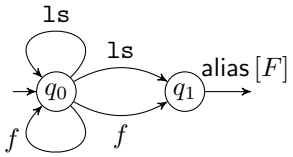


Fig. 8. $\mathcal{A}[\texttt{ls}(E, F)]$

When P calls other predicates, the automaton recognizes tree encodings of concatenations of more general SL graphs, obtained from $Gf[\Sigma]$ 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, alias [..] labels occurring in the TA for the nested predicate need to be modified. Labels of the form alias$\uparrow[m]$ and alias$\uparrow\downarrow[m]$ are adjusted by prefixing m with the marking of the source state of the transition. On the contrary, labels of the form alias $[V]$ are substituted by the marking of $\texttt{Node}(V)$ w.r.t. the higher-level matrix.

Let us consider a predicate $P_2(E, F)$ that calls P_1 and that has the matrix $\Sigma_2 \triangleq \exists Z.\, E \mapsto \{(g_1, X_{\texttt{t1}}), (g_2, Z)\} \wedge \circlearrowleft^{1+} P_1[Z, E]$. The TA $\mathcal{A}_2$ for $P_2(E, F)$ includes the following transition rules:

$(1')\ qq_0 \hookrightarrow g_1(qq_0), g_2(q_0)$

transition rules of $\mathcal{A}_1$, where
- alias $[F]$ is substituted by alias$\uparrow[g_1\ g_2]$,
- alias $[B]$ by alias$\uparrow[g_1]$, and
- alias$\uparrow\downarrow[f_1]$ is substituted by alias$\uparrow\downarrow[g_1\ g_2\ f_1]$

$(2')\ qq_0 \hookrightarrow g_1(qq_1), g_2(q_0)$
$(3')\ qq_1 \hookrightarrow$ alias $[F]$
$(4')\ qq_0 \hookrightarrow P_2(qq_0)$
$(5')\ qq_0 \hookrightarrow P_2(qq_1)$

Rule $(1')$ and the ones imported (after renaming of the labels) from $\mathcal{A}_1$ describe trees obtained from the tree encoding of $Gf[\Sigma_2]$ by replacing the edge looping in Z with a tree recognized by $\mathcal{A}_1$. According to $Gf[\Sigma_2]$, the node marking of Z is $g_1\, g_2$, and so the label alias $[F]$ shall be substituted by alias$\uparrow[g_1\ g_2]$, and the marking alias$\uparrow\downarrow[f_1]$ shall be substituted by alias$\uparrow\downarrow[g_1\ g_2\ f_1]$.

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

Theorem 2. *For any atom $P(E, F, \overrightarrow{B})$ and any SL graph G, if the tree generated by $\texttt{toTree}(G, P(E, F, \overrightarrow{B}))$ is recognized by $\mathcal{A}[P(E, F, \overrightarrow{B})]$, then $G \Rightarrow P(E, F, \overrightarrow{B})$.*

6 Completeness and Complexity

In general, there exist SL graphs that entail $P(E, F, \overrightarrow{B})$ whose tree encodings are *not* recognized by $\mathcal{A}[P(E, F, \overrightarrow{B})]$. The models of these SL graphs are nested list segments where inner pointer fields specified by the matrix of P are aliased. For example, the TA for $\texttt{skl}_2$ does not recognize tree encodings of SL graphs modeled by heaps where $X_{\texttt{t1}}$ and Z_1 are interpreted to the same location.

The construction of TAs explained above can be easily extended to cover such SL graphs (cf. App. C.2), but the size of the obtained automata may become exponential in the size of P (defined as the number of symbols in the matrices of all Q with $P \prec^*_{\mathbb{P}} Q$)

as the construction considers all possible aliasing scenarios of targets of inner pointer fields permitted by the predicate.

For the verification conditions that we have encountered in our experiments, the TAs defined above are precise enough in the vast majority of the cases. In particular, note that the TAs generated for the predicates for `ls` and `dll` (defined below) are precise. We have, however, implemented even the above mentioned extension and realized that it also provides acceptable performance.

In conclusion, the overall complexity of the semi-decision procedure (where aliases between variables in the definition of a predicate are ignored) runs in polynomial time modulo an oracle for deciding validity of a boolean formula (needed in normalization procedure). The complete decision procedure is exponential in the size of the predicates, and not of the formulas, which remains acceptable in practice.

7 Extensions

The procedures presented above can be extended to a larger fragment of SL that uses more general inductively defined predicates. In particular, they can be extended to cover finite nestings of singly or doubly linked lists. To describe doubly linked segments, we extend the definition of a predicate from Eq. 1 to the following:

$$\begin{aligned} R_{dl}(E,F,P,S,\overrightarrow{B}) \triangleq\ & (E = S \wedge F = P \wedge emp) \vee \\ & \big(E \neq S \wedge F \neq P \wedge \\ & \ \wedge\ \exists X_{\mathtt{tl}}.\, \Sigma(E, X_{\mathtt{tl}}, P, \overrightarrow{B}) * R_{dl}(X_{\mathtt{tl}}, F, E, S, \overrightarrow{B})\big) \end{aligned} \tag{7}$$

where Σ is an existentially-quantified matrix of the form:

$$\begin{aligned} \Sigma(E, X_{\mathtt{tl}}, P, \overrightarrow{B}) \ &\triangleq\ \exists \overrightarrow{Z}.\, E \mapsto \{\rho(\{X_{\mathtt{tl}}, P\} \cup \overrightarrow{V})\} * \Sigma' \qquad \text{where } \overrightarrow{V} \subseteq \overrightarrow{Z} \cup \overrightarrow{B} \text{ and} \\ \Sigma' \ &::=\ Q(Z, U, \overrightarrow{Y}) \mid \circlearrowleft^{1+} Q[Z, \overrightarrow{Y}] \mid \circlearrowleft^{1+} Q_{dl}[Z, \overrightarrow{Y}] \mid \Sigma' * \Sigma' \\ & \qquad \text{for } Z \in \overrightarrow{Z}, U \in \overrightarrow{Z} \cup \overrightarrow{B} \cup \{E, X_{\mathtt{tl}}, P\},\ \overrightarrow{Y} \subseteq \overrightarrow{B} \cup \{E, X_{\mathtt{tl}}, P\}, \\ \circlearrowleft^{1+} Q[Z, \overrightarrow{Y}] \ &\triangleq\ \exists Z'.\, \Sigma_Q(Z, Z', \overrightarrow{Y}) * Q(Z', Z, \overrightarrow{Y}) \text{ where } \Sigma_Q \text{ is the matrix of } Q, \text{ or} \\ \circlearrowleft^{1+} Q_{dl}[Z, \overrightarrow{Y}] \ &\triangleq\ \exists Z', Z_p.\, \Sigma_{Q_{dl}}(Z, Z', Z_p, \overrightarrow{Y}) * Q_{dl}(Z', Z_p, Z, Z, \overrightarrow{Y}) \\ & \qquad \text{where } \Sigma_{Q_{dl}} \text{ is the matrix of } Q_{dl}. \end{aligned}$$

In Eq. 7, P corresponds to the predecessor of E and S corresponds to the successor of F. For instance, to describe DLL segments between two locations E and F, one can use the predicate

$$\begin{aligned} \mathtt{dll}(E,F,P,S) \triangleq\ & (E = S \wedge F = P \wedge emp) \vee \\ & \big(E \neq S \wedge F \neq P \wedge \\ & \ \wedge\ \exists X_{\mathtt{tl}}.\, E \mapsto \{(next, X_{\mathtt{tl}}), (prev, P)\} * \mathtt{dll}(X_{\mathtt{tl}}, F, E, S)\big). \end{aligned} \tag{8}$$

To describe a singly linked list of cyclic doubly linked lists, we may use the following predicate:

$$\begin{aligned} \mathtt{nlcdl}(E,F) \triangleq\ & (E = F \wedge emp) \vee \\ & \big(E \neq F \wedge \exists X_{\mathtt{tl}}, Z.\, E \mapsto \{(s, X_{\mathtt{tl}}), (h, Z)\} * \circlearrowleft^{1+} \mathtt{dll}(Z) * \mathtt{nlcdl}(X_{\mathtt{tl}}, F)\big) \end{aligned}$$

where $\circlearrowleft^{1+}$ $\texttt{dll}(Z)$ is a macro for describing non-empty cyclic doubly linked lists defined by

$$\circlearrowleft^{1+} \texttt{dll}[Z] \triangleq \exists Z_1, Z_2.\, Z \mapsto \{(n, Z_1), (p, Z_2)\} * \texttt{dll}(Z_1, Z_2, Z, Z). \tag{9}$$

[TODO: talk about how the graph is modified before the entailment check]

Representing SL Graphs as Trees. The `splitJoin` operation from Sec. 4 is extended with considering the following two more possible labellings: $\mathsf{alias}\uparrow^2[\alpha]$ and $\mathsf{alias}\uparrow\downarrow_{\mathsf{last}}[\alpha]$. If n is a join node in a graph and (m, n) is an edge not in its spanning tree, then (m, n) is replaced by an edge (m, n') with the same edge label, s.t. n' is a fresh copy of n labeled by (in addition to the labellings from Sec. 4)

- $\mathsf{alias}\uparrow^2[\mathbb{M}(n)]$ if m is reachable from n, m further reaches n in the spanning tree of the graph and in the spanning tree there is exactly one node marked with $\mathbb{M}(n)$ between m and 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.
- $\mathsf{alias}\uparrow\downarrow_{\mathsf{last}}[\mathbb{M}(n)]$ if there is a node p which is a predecessor of m, s.t. all predecessors of of m that have a unique successor marked by $\mathbb{M}(n)$ are also predecessors of p, and n is a successor of p s.t. n does not occur on a path from p to any of its successors marked by $\mathbb{M}(n)$. Intuitively, the label is used for the copy of the predecessor of the header of the list.

Translation of predicates to TAs. The extension of the procedure from Sec. 5 is trivial. See App. E for more details. [TODO: talk about how pred2ta is modified]

[TODO: how do we handle the case when a marking is a parameter of an inductive predicate edge?]

8 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 unsatisfiablity of boolean formulas and the VATA library [15] as the tree automata backend.

We applied SPEN to entailment problems that use various recursive predicates. First, we considered the benchmark provided in [16], which uses only the `ls` predicate. It consists of three classes of entailment problems where the first two classes contain problems generated randomly according to the rules specified in [16], whereas the last class contains problems obtained from the verification conditions generated by the tool SMALLFOOT [2]. In all experiments[4], SPEN finished in less than 1 second with the deviation of running times ± 100 ms w.r.t. the ones reported for SELOGER [11], the

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

Table 1. Running SPEN on entailments between formulas and atoms.

φ_2	nll			nlcl			$\texttt{skl}_3$			dll		
φ_1	tc1	tc2	tc3	tc1	tc2	tc3	tc1	tc2	tc3	tc1	tc2	tc3
Time [ms]	344	335	319	318	316	317	334	349	326	358	324	322
Status	vld	vld	inv	vld	vld	inv	vld	vld	inv	vld	vld	inv
States/Trans. of $\mathcal{A}[\varphi_2]$	6/17			6/15			80/193			9/16		
Nodes/Edges of $T(Gf[\varphi_1])$	7/7	7/7	6/7	10/9	7/7	6/6	7/7	8/8	6/6	7/7	7/7	5/5

most efficient tool for deciding entailments of SL formulas with singly linked lists we are aware of (details are given in App. F).

The TA for `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 that, we further considered the experiments listed in Table 1 (among which, $\texttt{skl}_3$ required the extension of our approach to a full decision procedure as discussed at the end of Sec. 5). 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 start, in the middle, or at the end of various kinds of list segments (see App. F). Table 1 gives for each example the running time, the valid/invalid status, and the size of the tree encoding and TA for φ_1 and φ_2, respectively. We find the resulting times quite encouraging.

9 Related Work

Several decision procedures for fragments of SL have been introduced in the literature [1,5,6,9,13,12,16,17,4].

Some of these works [1,5,6,16] consider a fragment of SL that uses only one predicate describing singly linked lists, which is a much more restricted setting than what is considered in this paper. In particular, Cook et al. [6] prove that the satisfiability/entailment problem can be solved in polynomial time. Piskac et al. [17] show that the boolean closure of this fragment can be translated to a decidable fragment of first-order logic, and 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. [18].

A fragment of SL covering overlaid nested lists was considered in our previous work [9]. Compared with it, we currently do not consider overlaid lists, but we have enlarged the set of inductively-defined predicates to allow for 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. However, they do not provide a fragment for which completeness is guaranteed. Iosif et al. [13] also introduce a decidable fragment of SL that can describe more complex data

structures than those considered here, including, e.g., trees with parent pointers or trees with linked leaves. However, [13] reduces the entailment problem to MSO on graphs with a bounded tree width, resulting in a multiply-exponential complexity.

The recent work [12] considers a more restricted fragment than [13] (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 semi-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 DLLs viewed from the end). On the other hand, it 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.

10 Conclusion

We proposed a novel (semi-)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 it also efficiently handles practical verification conditions.

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.

References

1. J. Berdine, C. Calcagno, and P.W. O'Hearn. A Decidable Fragment of Separation Logic. In *Proc. of FSTTCS'04*, LNCS 3328, Springer, 2005.
2. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Proc. of FMCO'05*, LNCS 4111, Springer, 2006.
3. J. Brotherston, C. Fuhs, N. Gorogiannis, and J.A.N. Pérez. A Decision Procedure for Satisfiability in Separation Logic with Inductive Predicates. To appear in *Proc. of LICS'14*.
4. J. Brotherston, N. Gorogiannis, and R.L. Petersen. A Generic Cyclic Theorem Prover. In *Proc. of APLAS'12*, LNCS 7705, Springer, 2012.
5. C. Calcagno, H. Yang, P. O'Hearn. Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In *Proc. of FSTTCS'01*, LNCS 2245, Springer, 2001.
6. B. Cook, C. Haase, J. Ouaknine, M.J. Parkinson, and J. Worrell. Tractable Reasoning in a Fragment of Separation Logic. In *Proc. of CONCUR'11*, LNCS 6901, Springer, 2011.

7. C. Enea, O. Lengál, M. Sighireanu, and T. Vojnar. Compositional Entailment Checking for a Fragment of Separation Logic. Technical Report FIT-TR-2014-01, FIT BUT, 2014. `http://www.fit.vutbr.cz/~ilengal/pub/FIT-TR-2014-01.pdf`.
8. C. Enea, O. Lengál, M. Sighireanu, and T. Vojnar. SPEN, 2014. `http://www.liafa.univ-paris-diderot.fr/spen`.
9. C. Enea, V. Saveluc, and M. Sighireanu. Compositional Invariant Checking for Overlaid and Nested Linked Lists. In *Proc. of ESOP'13*, LNCS 7792, Springer, 2013.
10. A. Gotsman, J. Berdine, and B. Cook. Precision and the Conjunction Rule in Concurrent Separation Logic. *Electronic Notes in Theoretical Computer Science*, 276:171–190, 2011.
11. C. Haase, S. Ishtiaq, J. Ouaknine, and M.J. Parkinson. SeLoger: A Tool for Graph-based Reasoning in Separation Logic. In *Proc. of CAV'13*, LNCS 8044, Springer, 2013.
12. R. Iosif, A. Rogalewicz, and T. Vojnar. Deciding Entailments in Inductive Separation Logic with Tree Automata. Technical Report arXiv:1402.2127, 2014.
13. R. Iosif, A. Rogalewicz, and J. Šimáček. The Tree Width of Separation Logic with Recursive Definitions. In *Proc. of CADE-24*, LNCS 7898, Springer, 2013.
14. S. Ishtiaq and P.W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *Proc. of POPL'01*, ACM, 2001.
15. O. Lengál, J. Šimáček, and T. Vojnar. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In *Proc. of TACAS'12*, LNCS 7214, Springer, 2012.
16. J.A.N. Pérez and A. Rybalchenko. Separation Logic + Superposition Calculus = Heap Theorem Prover. In *Proc. of PLDI'11*, ACM, 2011.
17. R. Piskac, T. Wies, and D. Zufferey. Automating Separation Logic Using SMT. In *Proc. of CAV'13*, LNCS 8044, Springer, 2013.
18. X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural Proofs for Structure, Data, and Separation. In *Proc. of PLDI'13*, ACM, 2013.
19. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*, IEEE, 2002.

A Semantics

The relation $(S, H) \models \varphi$ is defined by the following ($\uplus$ denotes the disjoint union of sets and $S[X \leftarrow \ell]$ denotes the function S' s.t. $S'(X) = \ell$ and $S'(Y) = S(Y)$ for any $Y \neq X$):

$(S, H) \models E = F$	iff $S(E) = S(F)$
$(S, H) \models E \neq F$	iff $S(E) \neq S(F)$
$(S, H) \models \varphi \wedge \psi$	iff $(S, H) \models \varphi$ and $(S, H) \models \psi$
$(S, H) \models \mathit{emp}$	iff $\mathit{dom}(H) = \emptyset$
$(S, H) \models E \mapsto \{\rho\}$	iff $\mathit{dom}(H) = \{(S(E), f_i) \mid (f_i, E_i) \in \{\rho\}\}$ and for every $(f_i, E_i) \in \{\rho\}$, $H(S(E), f_i) = S(E_i)$
$(S, H) \models \Sigma_1 * \Sigma_2$	iff $\exists H_1, H_2$ s.t. $\mathit{ldom}(H) = \mathit{ldom}(H_1) \uplus \mathit{ldom}(H_2)$, $(S, H_1) \models \Sigma_1$, and $(S, H_2) \models \Sigma_2$
$(S, H) \models P(E, F, \overrightarrow{B})$	iff there exists $k \in \mathbb{N}$ s.t. $(S, H) \models P^k(E, F, \overrightarrow{B})$ and $\mathit{ldom}(H) \cap (\{S(F)\} \cup \{S(B) \mid B \in \overrightarrow{B}\}) = \emptyset$
$(S, H) \models P^0(E, F, \overrightarrow{B})$	iff $(S, H) \models E = F \wedge \mathit{emp}$
$(S, H) \models P^{k+1}(E, F, \overrightarrow{B})$	iff $(S, H) \models E \neq \{F\} \cup \overrightarrow{B} \wedge \exists X_{\mathtt{t1}}.\, \Sigma(E, X_{\mathtt{t1}}, \overrightarrow{B}) \ * \ P^k(X_{\mathtt{t1}}, F, \overrightarrow{B})$
$(S, H) \models \exists X.\, \varphi$	iff there exists $\ell \in \mathit{Locs}$ s.t. $(S[X \leftarrow \ell], H) \models \varphi$

B Algorithm `toTree`

Let G be an SL graph returned by `select` for the atom $P(E, F, \overrightarrow{B})$. The post-condition of `select` ensures that all nodes of G are reachable from the node Root labeled by E. The tree encoding of G is computed by the procedure $\texttt{toTree}(G, P(E, F, \overrightarrow{B}))$ below:

Input: G : an SL graph with all nodes reachable from the node Root labeled by E,
$P(E, F, \overrightarrow{B})$: an atom
Output: A labeled tree that encodes G

```
// compute the spanning tree
M ← nodeMarking(G, P, E, ≺_{F*})
// split nodes of Vars
G' ← splitJoinLabeled(G, M, E, {F} ∪ B)
// split unlabeled join nodes
T ← splitJoin(G', M)
// move labels from edges to src nodes
T' ← updateLabels(T)
return T'
```

Fig. 9. Tree encoding of SL graphs by `toTree`()

C Construction of Tree Automata for Predicates

This appendix first describes the basic algorithm for construction of tree automata accepting unfoldings of the predicate where every (singly linked) list segment (both top-

level and nested) is non-empty. Later, the algorithm is extended for list segments which might be empty and for doubly linked list segments.

C.1 Basic Algorithm for Non-Empty List Segments

Consider the definition of the matrix of the predicate $P(E, F, \overrightarrow{B})$ as given in Section 2 repeated for the sake of convenience here:

$$P(E, F, \overrightarrow{B}) \triangleq (E = F \wedge emp) \vee \big(E \neq \{F\} \cup \overrightarrow{B} \wedge \exists X_{\mathtt{tl}}.\, \Sigma(E, X_{\mathtt{tl}}, \overrightarrow{B}) * P(X_{\mathtt{tl}}, F, \overrightarrow{B})\big)$$

where Σ is of the form

$$\Sigma(E, X_{\mathtt{tl}}, \overrightarrow{B}) \triangleq \exists \overrightarrow{Z}.\, E \mapsto \rho(X_{\mathtt{tl}}, \overrightarrow{Z}, \overrightarrow{B}) * \Sigma'$$

$$\Sigma' ::= Q(Z, U, \overrightarrow{Y}) \mid \circlearrowleft^{1+} Q[Z, \overrightarrow{Y}] \mid \Sigma' * \Sigma'$$

for $Z \in \overrightarrow{Z}$, $U \in \overrightarrow{Z} \cup \overrightarrow{B} \cup \{E, X_{\mathtt{tl}}\}$, $\overrightarrow{Y} \subseteq \overrightarrow{B} \cup \{E, X_{\mathtt{tl}}\}$, and

$\circlearrowleft^{1+} Q[Z, \overrightarrow{Y}] \triangleq \exists Z'.\, \Sigma_Q(Z, Z', \overrightarrow{Y}) * Q(Z', Z, \overrightarrow{Y})$ where Σ_Q is the matrix of Q.

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 $\Sigma(E, X_{\mathtt{tl}}, \overrightarrow{B}) \triangleq \exists \overrightarrow{Z}.\, E \mapsto \{(f_1, Z_1), \ldots, (f_n, Z_n)\} * \Sigma'$. W.l.o.g. we further assume that $f_1 \prec_{\mathbb{F}} \ldots \prec_{\mathbb{F}} f_n$, i.e. f_1 is the minimum field in $\mathbb{F}_{\mapsto}(P)$.

The construction uses the SL graph of the following formula which unfolds two times the recursive definition of the predicate:

$$\exists X_{\mathtt{tl}}.\, \Sigma(E, X_{\mathtt{tl}}, \overrightarrow{B}) * \Sigma(X_{\mathtt{tl}}, F, \overrightarrow{B}). \tag{10}$$

The unfolding is done two times in order to capture all the markings (including the ones of nodes allocated inside the list segment) that may appear in tree encodings that shall be recognized by $\mathcal{A}[P]$. The graph G is obtained from the SL graph of (10) such that the macro $\circlearrowleft^{1+} Q[Z, \overrightarrow{Y}]$ is not expanded but translated into a predicate edge from $\mathtt{Node}(Z)$ to $\mathtt{Node}(Z)$ labelled with $Q(\overrightarrow{Y})$.

Then, we get the tree encoding $\mathcal{T}[G]$ 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 G, so $\mathcal{T}[G]$ does not use alias relation $\mathsf{alias}\,[X_{\mathtt{tl}}]$ but only $\mathsf{alias}\,[U]$ with $U \in \{E, F\} \cup \overrightarrow{B}$. The nodes aliasing the one labeled by $X_{\mathtt{tl}}$ in G use the relations $\mathsf{alias}\uparrow[f_1]$ or $\mathsf{alias}\uparrow\downarrow[f_1]$ because the marking of $\mathtt{Node}(X_{\mathtt{tl}})$ is f_1. Recall also that the nodes of G labeled by parameters or existentially quantified variables are pushed directly in $\mathcal{T}[G]$. So, we overload the notation $\mathtt{Node}(Z)$ to denote the node of $\mathcal{T}[G]$ obtained from the node of G labeled by Z.

The construction starts with an empty automaton $\mathcal{A}[P]$. It calls a procedure `buildTACall` which adds states and transitions to $\mathcal{A}[P]$ to recognize tree encodings of unfoldings of the atom $P(E, F, \overrightarrow{B})$. This procedure is recursive, because it is called for all atoms $Q(U, V, \overrightarrow{W})$ inside the formula (10). The formal parameters of

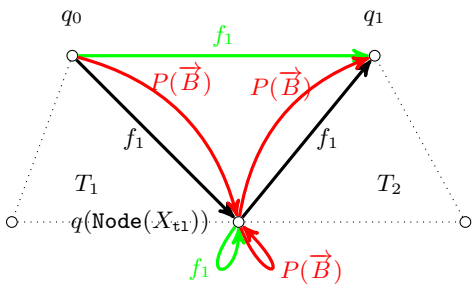


Fig. 10. General schema of $\mathcal{A}[P]$ built in `buildTACall`

`buildTACall` are: the predicate called, the mapping σ of its formal parameters to an aliasing relation, the states q_0 and q_1 to be used for the source resp. the destination of the construction, and the marking m_0 of state q_0. The initial values of these parameters are, respectively, P, $[E \mapsto \mathsf{alias}\,[E], F \mapsto \mathsf{alias}\,[F], \overrightarrow{B \mapsto \mathsf{alias}\,[B]}]$, fresh states q_0, q_1, and f_1. The state q_0 is marked as the root of $\mathcal{A}[P]$.

The procedure `buildTACall` has four steps, each step filling one of the parts of the general schema of $\mathcal{A}[P]$ given in Fig. 10.

I. Import the tree encoding $\mathcal{T}[G]$: This first step create 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 $\texttt{Node}(E)$ and $\texttt{Node}(F)$ for which are used the states q_0 resp. q_1.

(b) If the node u is labelled in $\mathcal{T}[G]$ with an aliasing relation $r \in \{\mathsf{alias}\,[B] \mid B \in \overrightarrow{B}\} \cup \{\mathsf{alias}\,\#[m] \mid \# \in \{\uparrow, \uparrow\downarrow\}, m \text{ a marking}\}$, we add the transition

$$q(u) \hookrightarrow\ r[\sigma \circ m_0] \tag{11}$$

where $\sigma \circ m_0$ substitutes a relation $\mathsf{alias}\,[B]$ by $\sigma(B)$ for any $B \in \overrightarrow{B}$, and a relation $\mathsf{alias}\,\#[m]$ by $\mathsf{alias}\,\#[m_0 \odot m]$.

(c) If there is a predicate edge from u to v labelled with $Q(\overrightarrow{Y})$, we add the transition

$$q(u) \hookrightarrow\ Q(\overrightarrow{Y}[\sigma \circ m_0])(q(v)). \tag{12}$$

where $\sigma \circ m_0$ replaces each variable in the tuple $\overrightarrow{Y}$ by:
- $\sigma(Y)$, if Y is a parameter,
- a relation $\mathsf{alias}\,\#[m_0 \odot m]$ where m is the marking of $\texttt{Node}(Y)$ and $\#$ is the relation between $\texttt{Node}(E)$ and $\texttt{Node}(Y)$, if Y is an existential variables in (10).

(d) If 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 $h_1 \prec_{\mathbb{F}} \ldots \prec_{\mathbb{F}} h_k$, and entering nodes $v_1, \ldots, v_k$, in this order, we add the transition

$$q(u) \hookrightarrow h_1(q(v_1)), \ldots, h_k(q(v_k)). \tag{13}$$

Note that this rule also creates the transition

$$q_0 \hookrightarrow f_1(q(\texttt{Node}(X_{\texttt{t1}}))), f_2(q(Z_2)), \ldots, f_n(q(Z_n)). \tag{14}$$

corresponding to the black edge labeled by f_1 in Fig. 10 between states q_0 and $q(\texttt{Node}(X_{\texttt{t1}}))$, and the black edge between $q(\texttt{Node}(X_{\texttt{t1}}))$ and q_1.

(e) If the call to `buildTACall` is not nested, we add the transition

$$q_1 \hookrightarrow \sigma(F). \tag{15}$$

Note that this skeleton is able to accept precisely two unfoldings of the predicate P between E and F such that nested predicates are not unfolded. This skeleton is represented two triangles built with black edges in Fig. 10; each triangle corresponds to an unfolding of the predicate definition encoded in $\mathcal{T}[G]$.

II. Accepting non empty list segments: Next, we make $\mathcal{A}[P]$ accept an arbitrary number of these unfoldings along the backbone field of the predicate. To do this, we take the initial transition (14) insert into $\mathcal{A}[P]$ the following transitions (represented in green in Fig. 10):

(a) a transition to accept exactly one unfolding:

$$q_0 \hookrightarrow f_1(q_1), f_2(q(Z_2)), \ldots, f_n(q(Z_n)). \tag{16}$$

(b) a transition to inserting more unfoldings:

$$q(\texttt{Node}(X_{\texttt{t1}})) \hookrightarrow f_1(q(\texttt{Node}(X_{\texttt{t1}}))), f_2(q(Z_2)), \ldots, f_n(q(Z_n)). \tag{17}$$

III. Interleave with predicate edges: We add transitions (represented in red in Fig. 10) allowing an arbitrary interleaving of folded and unfolded occurrences of the translated predicate P:

$$q_0 \hookrightarrow P(\overrightarrow{B}[\sigma]])(q(\texttt{Node}(X_{\texttt{t1}}))) \tag{18}$$

$$q(\texttt{Node}(X_{\texttt{t1}})) \hookrightarrow P(\overrightarrow{B}[\sigma])(q(\texttt{Node}(X_{\texttt{t1}}))) \tag{19}$$

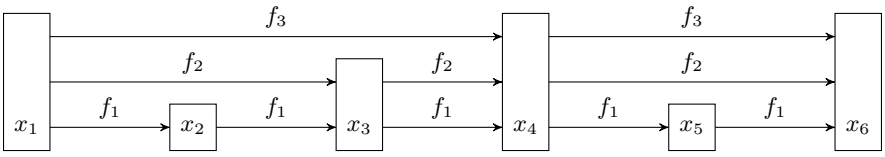
$$q(\texttt{Node}(X_{\texttt{t1}})) \hookrightarrow P(\overrightarrow{B}[\sigma])(q_1). \tag{20}$$

IV. Inserting tree automata of nested predicate edges: For each transition inserted in $\mathcal{A}[P]$ of the form:

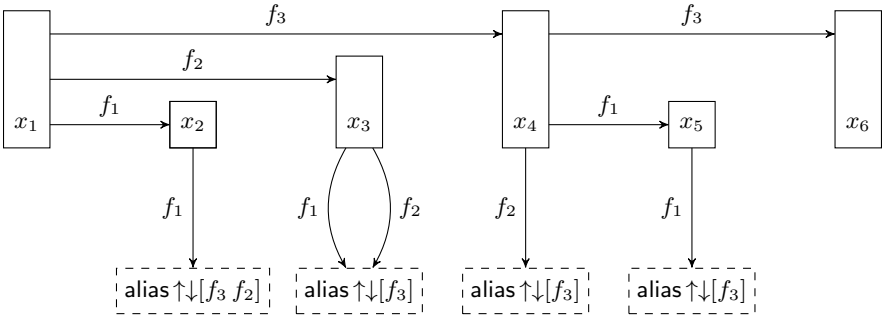
$$q(\texttt{Node}(R)) \hookrightarrow Q(\overrightarrow{Y})(q(\texttt{Node}(S))), \tag{21}$$

with $Q \neq P$, we call recursively the procedure `builTACall` to insert in $\mathcal{A}[P]$ the automaton for the call of the predicate Q with the parameters $(R, S, \overrightarrow{Y})$. The states created by each call of `builTACall` are new. The procedure `builTACall` is called with the process identifier Q,

- the mapping $[E \mapsto r_R, F \mapsto r_S, \overrightarrow{B \mapsto r_Y}]$, where for any $Z \in \{R, S\} \cup \overrightarrow{Y}$:
  - if $Z \in \{E, F\} \cup \in \overrightarrow{B}$ then r_Z is $\sigma(Z)$,
  - if $Z \in \overrightarrow{Z}$ (set of existentially quantified variables in P) then r_Z is $\mathsf{alias}{\uparrow}{\downarrow}[m_Z]$ where m_Z is the marking of $\texttt{Node}(Z)$ in $\mathcal{T}[G]$,
- the states $q(\texttt{Node}(R))$ and $q(\texttt{Node}(S))$, and
- the marking $m_0 \odot m_R$, where m_R is the marking of $\texttt{Node}(R)$ in $\mathcal{T}[G]$.



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



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

Fig. 11. Illustration of the issue with possibly empty nested list segments. 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.

C.2 Extension for Possibly Empty Nested List Segments

This extension creates tree automata that can accept such unfoldings of the predicate where nested list segments may be empty. The difficulties this creates are shown in Fig. 11. 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. Therefore, when the automaton is traversing the segment between x_4 and x_6, it needs to remember that if the second level list segment leaving x_4 is empty, the label at the end of the first level list segment leaving x_4 is not $\mathsf{alias}{\uparrow\downarrow}[f_3 f_2]$ but $\mathsf{alias}{\uparrow\downarrow}[f_3]$. Note that the top-level list segment predicate is always non-empty; the case when it is empty is dealt with during the normalization phase (see Section 3.2).

Suppose there are nested list segments $R_1, \ldots, R_n$ in the matrix $\Sigma(E, X_{\mathtt{tl}}, \overrightarrow{B})$ of the predicate $P(E, F, \overrightarrow{B})$ (note that the predicate of some distinct R_i and R_j can be the same, e.g. $R_i = \mathtt{ls}(S, T)$ and $R_j = \mathtt{ls}(U, V)$). For every subset S of the set of nested list segments, $S \subseteq \{R_1, \ldots, R_n\}$, we run the procedure in Section C.1 such that we first modify $\Sigma(E, X_{\mathtt{tl}}, \overrightarrow{B})$ in such a way that all nested list segments not in S are substituted by their ground case and obtain the automaton $\mathcal{A}^S$. We then obtain the automaton $\mathcal{A}[P(E, F, \overrightarrow{B})]$ by uniting all the automata retrieved in the previous step together and merging their initial states into one.

Formally, given the automata $\mathcal{A}^S = (Q^S, \mathcal{F}, q_0^S, \Delta^S)$ for all $S \subseteq \{R_1, \ldots, R_n\}$ (supposing the sets of states are pairwise disjoint) we create $\mathcal{A}[P(E, F, \overrightarrow{B})] =$

$(Q, \mathcal{F}, q_0, \Delta)$ in the following way.

$$Q = \{q_0\} \cup \bigcup_{S \subseteq \{R_1, \ldots, R_n\}} \left(Q^S \setminus \{q_0^S\}\right) \tag{22}$$

$$\Delta = \bigcup_{S \subseteq \{R_1, \ldots, R_n\}} \Delta^S \left[q_0 / q_0^S\right] \tag{23}$$

where $\Delta^S \left[q_0/q_0^S\right]$ denotes the set of transitions Δ^S where every occurrence of q_0^S is substituted with q_0.

C.3 Extension for Doubly Linked Lists

For DLLs, the procedure is more complex. Recall the definition of a doubly linked predicate:

$$P(E, F, P, S, \overrightarrow{B}) \triangleq (E = S \wedge F = P \wedge emp) \vee \\ \left(E \neq \{S\} \cup \overrightarrow{B} \wedge F \neq \{P\} \cup \overrightarrow{B} \wedge \exists X_{\mathtt{t1}}.\, \Sigma(E, X_{\mathtt{t1}}, P, \overrightarrow{B}) * P(X_{\mathtt{t1}}, F, E, S, \overrightarrow{B})\right)$$

First, we start with obtaining the SL formula use for the base of the algorithm. This is done by unfolding the predicate P several times.

D Tree Automata for the Running Example

The automaton $\mathcal{A}[\mathtt{ls}(E, F)]$ contains the following set of transition rules (with q_0 being the initial state):

$$\begin{array}{ll} q_0 \hookrightarrow f(q_0) & q_0 \hookrightarrow \mathtt{ls}(q_0) \\ q_0 \hookrightarrow f(q_1) & q_0 \hookrightarrow \mathtt{ls}(q_1) \\ q_1 \hookrightarrow \mathsf{alias}\,[F] & \end{array}$$

The automaton $\mathcal{A}[\mathtt{nll}(G, H, B)]$ contains the following set of transition rules (with qq_0 being the initial state):

$$\begin{array}{ll} qq_0 \hookrightarrow s(qq_0), h(q_0) & qq_0 \hookrightarrow s(qq_1), h(q_0) \\ qq_1 \hookrightarrow \mathsf{alias}\,[H] & qq_0 \hookrightarrow \mathtt{nll}(B)(qq_0) \\ \text{transition rules of } \mathcal{A}[\mathtt{ls}(E, B)] & qq_0 \hookrightarrow \mathtt{nll}(B)(qq_1) \end{array}$$

The automaton $\mathcal{A}[\mathtt{skl}_1(K, L)]$ contains the following set of transition rules (p_0 is the initial state):

$$\begin{array}{ll} p_0 \hookrightarrow f_3(p_\perp), f_2(p_\perp), f_1(p_0) & p_0 \hookrightarrow \mathtt{skl}_1(p_0) \\ p_0 \hookrightarrow f_3(p_\perp), f_2(p_\perp), f_1(p_1) & p_0 \hookrightarrow \mathtt{skl}_1(p_1) \\ p_1 \hookrightarrow \mathsf{alias}\,[L] & p_\perp \hookrightarrow \mathsf{alias}\,[\mathtt{NULL}] \end{array}$$

The automaton $\mathcal{A}[\mathtt{skl}_2(M, N)]$ contains the following set of transition rules (pp_0 is the initial state):

$$\begin{array}{ll} pp_0 \hookrightarrow f_3(p_\perp), f_2(pp_0), f_1(p_0) & pp_0 \hookrightarrow \mathtt{skl}_2(pp_0) \\ pp_0 \hookrightarrow f_3(p_\perp), f_2(pp_1), f_1(p_0) & pp_0 \hookrightarrow \mathtt{skl}_2(pp_1) \\ \text{transition rules of } \mathcal{A}[\mathtt{skl}_1(K, L)], \text{ where} & pp_1 \hookrightarrow \mathsf{alias}\,[N] \\ \quad \mathsf{alias}\,[L] \text{ is substituted by } \mathsf{alias}\uparrow\downarrow[f_2] & \end{array}$$

The automaton $\mathcal{A}[\mathtt{skl}_3(P, R)]$ contains the following set of transition rules (ppp_0 is the initial state):

$ppp_0 \hookrightarrow f_3(ppp_0), f_2(pp_0), f_1(p_0)$ $\quad$ $ppp_0 \hookrightarrow \mathtt{skl}_3(ppp_0)$
$ppp_0 \hookrightarrow f_3(ppp_1), f_2(pp_0), f_1(p_0)$ $\quad$ $ppp_0 \hookrightarrow \mathtt{skl}_3(ppp_1)$
transition rules of $\mathcal{A}[\mathtt{skl}_2(M, N)]$, where $\quad$ $ppp_1 \hookrightarrow \mathsf{alias}\,[R]$
$\mathsf{alias}\,[N]$ is substituted by $\mathsf{alias}{\uparrow}{\downarrow}[f_3]$
$\mathsf{alias}{\uparrow}{\downarrow}[f_2]$ is substituted by $\mathsf{alias}{\uparrow}{\downarrow}[f_3\ f_2]$

The automaton $\mathcal{A}[\mathtt{nlcl}(S, T)]$ contains the following set of transition rules (with qq_0 being the initial state):

$qq_0 \hookrightarrow s(qq_0), h(q_0)$ $\quad$ $qq_0 \hookrightarrow s(qq_1), h(q_0)$
$qq_1 \hookrightarrow \mathsf{alias}\,[T]$ $\quad$ $qq_0 \hookrightarrow \mathtt{nlcl}(qq_0)$
transition rules of $\mathcal{A}[\mathtt{ls}(E, F)]$, where $\quad$ $qq_0 \hookrightarrow \mathtt{nlcl}(qq_1)$
$\mathsf{alias}\,[F]$ is substituted by $\mathsf{alias}{\uparrow}[s\ h]$

E Extending the Tree Encoding to Deal with Doubly Linked Lists

The SL graphs of two formulas that entail $\mathtt{dll}(E, F, P, S)$ and $\mathtt{nlcdl}(E, F)$ from Sec. 7 and their tree encodings are given in Fig. 12 and Fig. 13 respectively. To deal with doubly linked lists, one has to modify the step of the procedure `toTree` that removes join nodes which are not labeled by P or S in the case of `dll` and by F in the case of `nlcdl`. These are the arguments that are not supposed to be allocated in any model of the predicate. Basically, we need to consider the labels $\mathsf{alias}{\uparrow}^2[\alpha]$ and $\mathsf{alias}{\uparrow}{\downarrow}_{\mathsf{last}}[\alpha]$ introduced in Sec. 7.

F Details of the Experiments

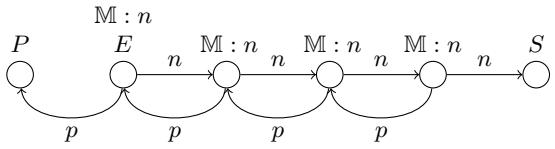
In this appendix, we give details of the experiments described in Sec. 8[5]. First, we considered the benchmark provided in [16], which uses only the `ls` predicate. It consists of three classes of entailment problems called *Spaguetti*, *Bolognesa*, and *Clones*. The first two classes contain 110 problems each (split into 11 groups) generated randomly according to the rules specified in [16], whereas the last class contains 100 problems (split into 10 groups) obtained from the verification conditions generated by the tool SMALLFOOT [2]. For the first two benchmark suites, we observe a deviation of the running times of ± 100 ms w.r.t. the ones reported for SELOGER [11][6]. The results are listed in Table 2. We give the average time for running SPEN on the 10 problems of each group.

In the following, we give the formulas for φ_1 used in the experiments for checking entailments between formulas and atoms. For all cases, entailments are valid for `tc1` and `tc2`, and invalid for `tc3`.

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

[6] The times reported for SELOGER in [11] have been obtained on an Intel Core TM i5-2467M 1.60 GHz processor with 4 GiB DDR3 1066 MHz under Windows 7 Home Premium (64-bit)

An SL graph which entails $\texttt{dll}(E, F, P, S)$:



and its tree encoding:

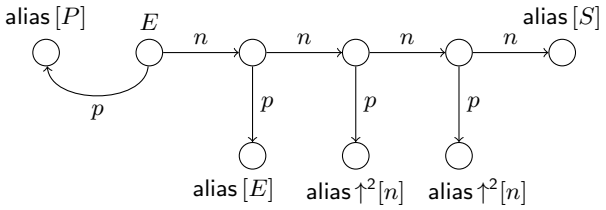


Fig. 12. Tree encodings for doubly linked lists.

Table 2. Running SPEN on the benchmarks from [16].

Bolognesa	bo-10	bo-11	bo-12	bo-13	bo-14	bo-15	bo-16	bo-17	bo-18	bo-19	bo-20
Average time [ms]	352	386	385	394	483	562	424	510	503	516	522
Spaguetti	sp-10	sp-11	sp-12	sp-13	sp-14	sp-15	sp-16	sp-17	sp-18	sp-19	sp-20
Average time [ms]	146	156	145	153	189	258	198	254	249	252	282
Clones	cl-01	cl-02	cl-03	cl-04	cl-05	cl-06	cl-07	cl-08	cl-09	cl-10	
Average time [ms]	316	314	335	336	321	334	351	374	407	436	

- $\varphi_2 = \texttt{nll}(x, y, z)$

  $\texttt{tc1} \triangleq x \mapsto \{(s, u), (h, a)\} * u \mapsto \{(s, y), (h, b)\} * \texttt{ls}(a, z) * \texttt{ls}(b, z)$

  $\texttt{tc2} \triangleq \texttt{nll}(x, u, z) * u \mapsto \{(s, w), (h, a)\} * a \mapsto \{(f, b)\} * \texttt{ls}(b, z) *$
  $\texttt{nll}(w, y, z)$

  $\texttt{tc3} \triangleq \texttt{nll}(x, u, z) * u \mapsto \{(s, w), (h, a)\} * a \mapsto \{(f, b)\} * b \mapsto \{(f, a)\} *$
  $\texttt{nll}(w, y, z)$

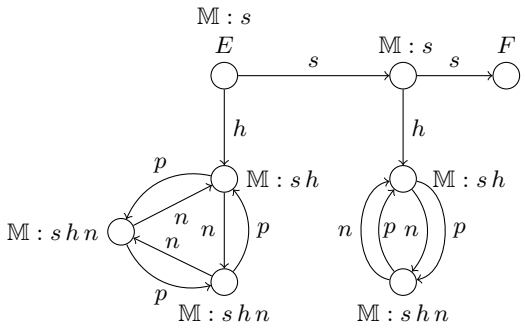
- $\varphi_2 = \texttt{nlcl}(x, y)$

  $\texttt{tc1} \triangleq x \mapsto \{(s, u), (h, a)\} * a \mapsto \{(f, b)\} * b \mapsto \{(f, a)\} *$
  $u \mapsto \{(s, y), (h, c)\} * c \mapsto \{(f, d)\} * \texttt{ls}(d, c)$

  $\texttt{tc2} \triangleq \texttt{nlcl}(x, u) * u \mapsto \{(s, v), (h, a)\} * a \mapsto \{(f, b)\} * \texttt{ls}(b, a) *$
  $\texttt{nlcl}(v, y)$

  $\texttt{tc3} \triangleq \texttt{nlcl}(x, u) * u \mapsto \{(s, v), (h, a)\} * a \mapsto \{(f, y)\} * \texttt{nlcl}(v, y)$

An SL graph which entails $\texttt{nlcdl}(E, F)$:



and its tree encoding:

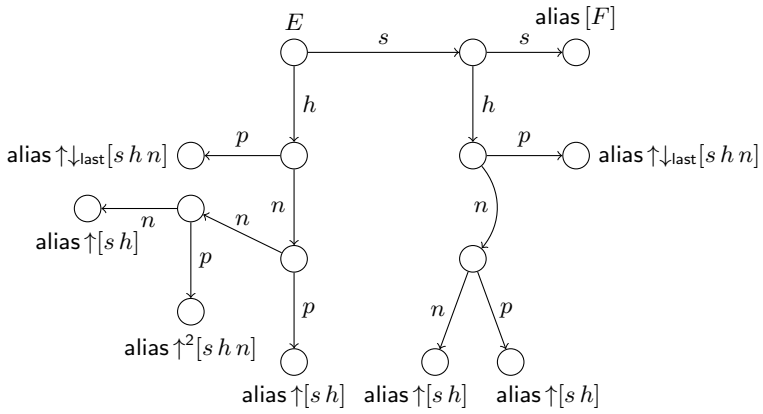


Fig. 13. Tree encodings for lists of cyclic doubly linked lists.

- $\varphi_2 = \mathtt{skl}_3(x, y)$

$$
\begin{array}{rcl}
\mathtt{tc1} & \triangleq & x \mapsto \{(f_1, z), (f_2, z), (f_3, z)\} * z \mapsto \{(f_1, y), (f_2, y), (f_3, y)\} \\
\mathtt{tc2} & \triangleq & \mathtt{skl}_3(x, z) * z \mapsto \{(f_3, w), (f_2, z_2)(f_1, z_1)\} * \mathtt{skl}_1(z_1, z_2) * \\
 & & \mathtt{skl}_2(z_2, w) * \mathtt{skl}_3(w, y) \\
\mathtt{tc3} & \triangleq & x \mapsto \{(f_1, w), (f_2, w), (f_3, w)\} * w \mapsto \{(f_1, z), (f_2, w_2), (f_3, z)\} * \\
 & & \mathtt{skl}_2(w_2, z) * \mathtt{skl}_3(z, y)
\end{array}
$$

- $\varphi_2 = \mathtt{dll}(x, y, z, v)$

$$
\begin{array}{rcl}
\mathtt{tc1} & \triangleq & x \mapsto \{(n, u), (p, z)\} * u \mapsto \{(n, y), (p, x)\} * y \mapsto \{(n, v), (p, u)\} \\
\mathtt{tc2} & \triangleq & x \mapsto \{(n, u), (p, z)\} * \mathtt{dll}(u, w, x, y) * y \mapsto \{(n, v), (p, w)\} \\
\mathtt{tc3} & \triangleq & x \mapsto \{(n, u), (p, z)\} * \mathtt{dll}(u, w, x, y) * y \mapsto \{(n, v)\}
\end{array}
$$