Compositional Invariant Checking for Overlaid and Nested Linked Lists*

Constantin Enea, Vlad Saveluc, and Mihaela Sighireanu

Univ Paris Diderot, Sorbonne Paris Cite, LIAFA CNRS UMR 7089, Paris,
{cenea,sighirea}@liafa.univ-paris-diderot.fr
vlad.saveluc@gmail.com

Abstract. We introduce a fragment of separation logic, called *NOLL*, for automated reasoning about programs manipulating overlaid and nested linked lists, where overlaid means that the lists share the same set of objects. The distinguishing features of *NOLL* are: (1) it is parametrized by a set of user-defined predicates specifying nested linked list segments, (2) a "per-field" version of the separating conjunction allowing to share object locations but not record field locations, and (3) it can express sharing constraints between list segments. We prove that checking the entailment between two *NOLL* formulas is co-NP complete using a small model property. We also provide an effective procedure for checking entailment in *NOLL*, which first constructs a Boolean abstraction of the two formulas in order to infer all the implicit constraints, and then, it checks the existence of a homomorphism between the two formulas, viewed as graphs. We have implemented this procedure and applied it on verification conditions generated from several interesting case studies that manipulate overlaid and nested data structures.

1 Introduction

Reasoning about behaviors of programs that manipulate dynamic data structures is a challenging problem because of the difficulty of representing (potentially infinite) sets of configurations, and of manipulating these representations for the analysis of the execution of program statements. For instance, pre/post-condition reasoning requires being able, given pre- and post-conditions ϕ resp. ψ, and a straight-line code P, (1) to compute the (strongest) post-condition of executing P starting from ϕ, denoted $\mathtt{post}(P,\phi)$, and (2) to check that it entails ψ. Therefore, an important issue is to investigate logic-based formalisms where pre/post conditions are expressible for the class of programs under interest, and for which it is possible to compute effectively post-conditions, and to efficiently check the entailment. The latter can be done either using theorem provers, where user-provided tactics are needed to guide the proof system, or using decision procedures, when the given annotations are in a decidable fragment. An essential ingredient in order to scale to large programs is being able to perform compositional reasoning and, in this context, Separation Logic [17] (SL) has emerged as a fundamental approach. Its main tool is the frame rule, which states that if the Hoare triple $\{\phi\}P\{\psi\}$ holds and P does not alter free variables in σ then $\{\phi * \sigma\}P\{\psi * \sigma\}$ also holds, where $*$ denotes the

* This work has been partially supported by the French ANR project Veridyc and by FSMP.

separating conjunction. Therefore, when reasoning about P one has to manipulate only specifications for the heap region altered by P.

In this paper, we define a fragment of SL, called *NOLL*, suitable for compositional reasoning about programs that manipulate *overlaid and nested* linked lists, built with an arbitrary set of fields. Such data structures are used in low-level code to link objects with respect to different aspects. For example, the network monitoring software Nagios (`www.nagios.com`) manipulates hash-tables with closed addressing, implemented as arrays of linked lists, where all the elements in the lists are also linked in the order of their insertion time. Here, we have two overlaid data structures, i.e., which share a set of objects: an array of linked lists and a singly-linked list.

To specify such data structures, *NOLL* is parametrized by a fixed, but arbitrary, set of recursive predicates defined in a higher-order extension of *NOLL* and which are expressive enough to specify various types of (nested) linked lists, e.g., singly-linked lists of singly-linked lists, where all the elements point to some fixed object.

To specify that these list segments are overlapped, *NOLL* includes, besides the classical operator $*$, that we will call object separating conjunction, a field separating conjunction operator $*_w$. Both operators separate the heap into disjoint regions, the only difference being the granularity of the separated heap cells. For $*$, a heap cell corresponds to a heap object. For $*_w$, a heap cell corresponds to a field from a heap object. Thus, the $*_w$ operator allows to specify data structures sharing sets of objects as long as they are built over disjoint sets of fields. In the example above, if $\mathsf{ArrOfSl}$ and Sl are formulas specifying the array of lists, resp. the list, then $\mathsf{ArrOfSl} *_w \mathsf{Sl}$ expresses the fact that the two structures share some objects.

However, $*w$ alone is not enough to describe precisely overlaid data structures. In the example above, we would also need to express the fact that the objects of the list described by Sl are exactly *all* the list objects in $\mathsf{ArrOfSl}$; let $\mathsf{Sl\_type}$ be their type. To this, we index each atomic formula specifying list segments by a variable, called a *set of locations variable* and interpreted as the set of all heap objects in the list segment. The values of these new variables can be constrained in a logic that uses classical set operators $\subseteq$ and $\cup$. For example, the specification $\mathsf{ArrOfSl}_\alpha *_w \mathsf{Sl}_\beta \wedge \alpha(\mathsf{Sl\_type}) = \beta$ constrains the set of objects in the linked list to be exactly the set of objects of type $\mathsf{Sl\_type}$ in the array of linked lists. (A *NOLL* formula φ can also put constrains over some set of locations variables, which are not associated to atomic formulas in φ.)

The semantics of the field separating conjunction $*_w$ allows us to establish another frame rule, which is essential for compositional reasoning about overlaid data structures: if the Hoare triple $\{\phi\}\ P\ \{\psi\}$ holds then $\{\phi *_w \sigma\}\ P\ \{\psi *_w \sigma\}$ also holds, where P is a straight-line code that does not alter fields described by σ, and the set of locations variables in σ are not bound to atomic formulas in ϕ or ψ. The consequences of this frame rule are that, to reason about a program fragment P, one has to provide only specifications for the data structures built with fields altered by P.

We prove that checking satisfiability of *NOLL* formulas is NP-complete and that the problem of checking entailments between *NOLL* formulas is co-NP complete. The upper bound on the complexity of checking satisfiability/entailment is first proved using a small model argument, and subsequently, following the approach in [8]. The second proof provides also an effective decision procedure for proving the validity of an en-

tailment $\varphi \Rightarrow \psi$ by (1) computing a normal form for the two formulas and (2) checking the existence of a homomorphism from the graph representation of the normal form of ψ to the graph representation of the normal form of φ. The main advantages of this decision procedure are: (i) by defining a Boolean abstraction for *NOLL* formulas, the construction of the normal form is reduced to (un)satisfiability queries to a SAT solver and (ii) checking the existence of a homomorphism between graph representations of formulas can be done in polynomial time.

To summarize, this work makes the following contributions:

- defines a fragment of SL, called *NOLL*, that can be used to perform compositional reasoning about overlaid and nested linked structures,
- proves that checking satisfiability, resp. entailment, of *NOLL* formulas is NP-complete, resp. co-NP complete,
- defines effective procedures for checking satisfiability and entailment of *NOLL* formulas based on SAT solvers, which are implemented in a prototype tool and proven to be efficient in practice.

Related Work: SL has been widely used in the literature for the analysis and the verification of programs with dynamic data structures [1–8, 12, 13, 17, 19].

The *NOLL* fragment incorporates several existing features of SL: the separating conjunction $*$ introduced in [12], the separating conjunction $*_w$ introduced in [6], and the inductive predicates describing nested linked structures introduced in [1]. The set of location variables are an abstraction of the sequences defined in [17]. However, [1, 6] use these features in order to define abstract domains for program analysis. The (partial) order relation on elements of these abstract domains can be seen as a sound, but not complete, decision procedure for entailment.

The works in [2, 5, 8] introduce results concerning the decidability/complexity of the satisfiability/entailment problem in fragments of SL. Berdine et al. [2] defines a fragment that allows to reason about programs with singly-linked lists and proves that the satisfiability of a formula can be decided in NP and that checking the validity of an entailment between two formulas belongs to the co-NP complexity class. A decision procedure for entailments in the same fragment is introduced in [16], which combines SL inference rules with a superposition calculus to deal with (in)equalities between variables. These complexity results were improved in [8] where it is proved that the satisfiability/entailment problem for the previous fragment can be solved in polynomial time. In fact, the procedure for checking entailments of *NOLL* formulas based on normal forms and graph homomorphism is inspired by the work in [8]. The differences are that (a) the procedure for computing the normal form of a *NOLL* formula is based on a new approach that uses Boolean abstractions (the procedure in [8] works only for singly-linked lists and can not be extended to *NOLL*) and (b) the notion of graph homomorphism is extended in order to handle the two versions of the separating conjunction, the constraints on set of locations variables, and more general recursive predicates.

The (sound) decision procedures for satisfiability/entailment introduced in [18, 15] are also based on Boolean abstractions of formulas. As in our case, the Boolean abstractions are used to transform logical validity into simpler decidable problems. However, they concern different types of logics: algebraic data types specifications for reason-

ing about functional programs in [18] and a recursive extension of first-order logic for reasoning about programs manipulating tree data structures in [15].

Semi-automatic frameworks for reasoning about programs within SL, based on theorem provers, have been defined in [7, 4, 13]. In this paper, we target a completely automatic framework based on decision procedures.

2 Overview

In general, *NOLL* formulas have the form $\Pi \wedge \Sigma \wedge \Lambda$, where Π is the pure part, i.e., a conjunction of equalities and inequalities between program variables expressing aliasing constraints, Σ is the spatial part specifying the data structures and the separation properties, and Λ specifies the sharing constraints between the data structures. The objects building the data structures in the heap are sets of record fields, called simply fields in the following.

$$\varphi := x \neq \texttt{NULL} \wedge Hash_{\alpha}(x, y, \texttt{NULL}) \ *_w \ List_{\beta}(z, \texttt{NULL}) \wedge \alpha(\mathsf{Sl\_type}) = \beta \quad (1)$$

$$Hash(in, out, dest) \triangleq (in = out) \vee (\exists u, v.\ in \mapsto \{(g,u);(h,v)\} * LowList(v, dest) * Hash(u, out, dest)) \quad (2)$$

$$LowList(in, out) \triangleq (in = out) \vee (\exists u.\ in \mapsto \{(s,u)\} * LowList(u, out)) \quad (3)$$

$$List(in, out) \triangleq (in = out) \vee (\exists u.\ in \mapsto \{(f,u)\} * List(u, out)) \quad (4)$$

Fig. 1: *NOLL* specification of a hash table whose elements are shared with a list.

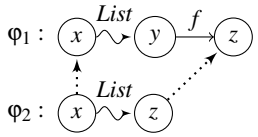
Examples of *NOLL* formulas: Fig. 1 contains a *NOLL* formula φ describing a list of lists, using the predicate $Hash_{\alpha}(x, y, \texttt{NULL})$, such that the elements of the nested lists are shared with another list, represented by the predicate $List_{\beta}(z, \texttt{NULL})$. This is an abstraction of the hash table sharing all its elements with a singly-linked list, presented in Sec. 1, in the sense that we use a linked list to represent the array structure.

The predicate $Hash_{\alpha}(in, out, dest)$ has a recursive definition, written in a higher-order extension of *NOLL*: either $in = out$, which means that the nested list segment is empty, or in contains a field h pointing to an inner singly-linked list ($in \mapsto \{\ldots;(h,v)\} * LowList(v, dest)$) and also a field g pointing to a new location u ($in \mapsto \{(g,u);\ldots\}$), which is the starting point of another nested list segment. Note that the elements of the lists described by $LowList(v, dest)$ are linked by the field s. In general, we suppose that variables and fields are typed. Thus, if $\mathsf{Sl\_type}$ is the type of the variables used in the predicate *LowList*, all the objects in the nested lists are of type $\mathsf{Sl\_type}$. Moreover, the use of the object separating conjunction $*$ implies that all the nested lists are disjoint.

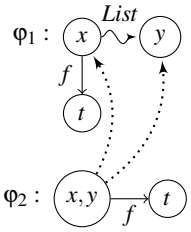
The overlapping property is expressed using two features of this logic. The first one is the field separating conjunction operator $*_w$ which allows to share object locations but not the locations of fields in these objects. The second feature is the ability to speak about the set of all object locations in a list segment. This set of locations is given by the interpretation of the variable that indexes some recursive predicate, e.g., α in $Hash_{\alpha}(\ldots)$. These variables are constrained in the Λ part of a formula. For example, $\alpha(\mathsf{Sl\_type}) = \beta$ says that all the locations of type $\mathsf{Sl\_type}$ in the list of lists are also present in the list starting in z (β stands for the set of locations in $List_{\beta}(z, \texttt{NULL})$).

The operators $*$ and $*_w$ can be nested. This is essential to specify a similar data structure (considered in [11]) where the elements stored in a hash table are shared between two disjoint linked lists (using the predicates from Fig. 1):

$$x \neq \texttt{NULL} \wedge \mathit{Hash}_{\alpha}(x,y,\texttt{NULL}) *_w (\mathit{List}_{\beta}(z,\texttt{NULL}) * \mathit{List}_{\gamma}(u,\texttt{NULL})) \wedge \alpha(\mathsf{Sl\_type}) = \beta \cup \gamma,$$



(a)



(b)

Fig. 2

where $*$ is used to specify the disjointness of the linked lists starting in z and u.

Decision procedure for entailment: We define a procedure for checking entailments of *NOLL* formulas, which is based on the graph homomorphism approach in [8]. The basic idea is to think of formulas as graphs, where nodes represent variables (sets of equal variables) and edges represent list segments, and then, given φ_1 and φ_2 two formulas, if there exists a homomorphism from φ_2 to φ_1 then $\varphi_1 \Rightarrow \varphi_2$ holds. Roughly, the homomorphism is a function mapping each node of φ_2 to a node of φ_1 representing at least the same set of variables. It is required that this function defines a mapping from edges of φ_2 to disjoint paths in φ_1. (Note that the homomorphism is unique.) For example, there exists such a homomorphism from φ_2 to φ_1 in Fig. 2(a), where a snaked edge labeled by *List* from x to y denotes a predicate $\mathit{List}(x,y)$, a straight edge labeled by f from y to z denotes a points-to constraint $y \mapsto \{(f,z)\}$, all these constraints are supposed to be separated by $*$, and the dotted edges represent the homomorphism.

In order to be complete, this procedure needs that the formulas of an entailment contain the maximum number of equalities and inequalities; in this case, we say that the formula is in *normal form*. Also, if it contains an equality $u = v$ then, it contains no spatial constraint defining a list segment from u to v (as usual in separation logic, $u = v \wedge \mathit{List}(u,v)$ is equivalent to $u = v$). For example, although the entailment $\varphi_1 \Rightarrow \varphi_2$ in Fig. 2(b) holds, there exists no homomorphism from φ_2 to φ_1. (Because the field f is already defined in x, the list segment using this field and starting in x is empty. Thus, φ_1 implies $x = y$, which is needed to show that $\varphi_1 \Rightarrow \varphi_2$.)

Boolean abstractions of *NOLL* formulas: Our first insight in defining such a decision procedure is that the normal form of a *NOLL* formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$ can be constructed through a boolean abstraction of φ, denoted $F(\varphi)$. For the moment, let us consider the case when $\Lambda = \mathit{true}$. Then, the formula $F(\varphi)$ is defined over a set of boolean variables denoting (in)equalities between variables and atomic formulas from the spatial part Σ.

We illustrate the definition of $F(\varphi)$ on the formula:

$$\varphi := \mathit{List}(x,y) * \mathit{List}(x,z) * y \mapsto \{(f,t)\} * \mathit{List}(y,s). \tag{5}$$

The set of boolean variables in $F(\varphi)$ consists of:

- a variable $[u = v]$, for every two variables u and v in φ,
- a variable $[y,t,f]$ to represent the points-to constraint $y \mapsto \{(f,t)\}$, and

– a variable $[List(u,v)]$, for every spatial constraint $List(u,v)$ in φ.

In this case, the formula $F(\varphi) \triangleq F_{eq} \wedge F(\Sigma)$, where F_{eq} encodes the reflexivity and the transitivity of the equality relation, i.e.,

$$\bigwedge_{u,v,w \text{ variables in } \varphi} [u=u] \quad \wedge \quad ([u=v] \wedge [v=w]) \Rightarrow [u=w],$$

and $F(\Sigma)$ models the spatial part of φ, i.e.,

$$F(\Sigma) \triangleq [y,t,f] \quad \wedge \bigwedge_{List(u,v) \text{ atom in } \varphi} [List(u,v)] \oplus [u=v] \quad \wedge \bigwedge_{A,B \text{ atoms in } \Sigma} F_*(A,B).$$

The sub-formula $[y,t,f]$ ensures that the points-to constraint is satisfied by any model of φ; the sub-formula $[List(u,v)] \oplus [u=v]$ models the fact that in any model of φ, either $u=v$ or $List(u,v)$ describes a non-empty list segment. The sub-formula $F_*(A,B)$ contains the in(equalities) implied by the use of $*$, i.e,

$$F_*(y \mapsto \{(f,t)\}, List(u,v)) \triangleq \neg[y=u] \vee [u=v], \text{ for any } u,v,$$
$$F_*(List(u_1,v_1), List(u_2,v_2)) \triangleq \neg[u_1=u_2] \vee [u_1=v_1] \vee [u_2=v_2], \text{ for any } u_1,v_1,u_2,v_2.$$

In general, the size of $F(\varphi)$ is polynomial in the size of the formula φ. Also, φ is satisfiable iff $F(\varphi)$ is satisfiable.

Computing the normal form: The formula $F(\varphi)$ can be used to compute the normal form of φ since $\varphi \Rightarrow (u=v)$ iff $F(\varphi) \Rightarrow [u=v]$, for any u and v. Thus, for any valid entailment $F(\varphi) \Rightarrow [u=v]$, the equality $u=v$ is added to φ, and all predicates describing list segments between u and v are removed. For example, the normal form of φ in (5) is $y=s \wedge x=z \wedge List(x,y) * y \mapsto \{(f,t)\}$ (the formula $F(\varphi)$ implies $[y=s]$ and $[x=z]$).

Handling sharing constraints: For *NOLL* formulas with sharing constraints, computing the normal form before checking the existence of a graph homomorphism is not enough. Besides (in)equalities, we may have implicit spatial constraints which are not exposed in some formula. Consider the entailment $\varphi_1 \Rightarrow \varphi_2$, where:

$$\varphi_1 := List_\alpha(x,y) *_w LowList_\beta(n,m) \wedge \beta \subseteq \alpha \tag{6}$$
$$\varphi_2 := \big(List_\delta(x,n) * List_\gamma(n,y)\big) *_w LowList_{\beta'}(n,m) \wedge \beta' \subseteq \delta \cup \gamma \tag{7}$$

Note that $\beta \subseteq \alpha$ implies that n is a location on the list segment described by $List_\alpha(x,y)$ and thus $\varphi_1 \Rightarrow \varphi_2$ holds. In this case, $F(\varphi_1)$ includes constraints over a set of boolean variables $[u \in \varepsilon]$ representing the fact that u is a location in the set of locations denoted by ε, for any u and $\varepsilon \in \{\alpha,\beta\}$ (we defer the reader to Sec. 5 for more details).

In general, if the formula $F(\varphi)$ implies $[u \in \varepsilon]$, for some u and ε, then the graph representation of φ includes some additional edges induced by the fact that u is a location on the list segment indexed by ε. In this case, $F(\varphi_1) \Rightarrow [n \in \alpha]$ and the graph representation of φ_1 completed with these additional edges is the graph $\overline{G}(\varphi_1)$ in Fig. 3. Now, it is easy to see that there exists a homomorphism from $G(\varphi_2)$ to $\overline{G}(\varphi_1)$ (the homomorphism must satisfy additional constraints explained in Sec. 6.3).

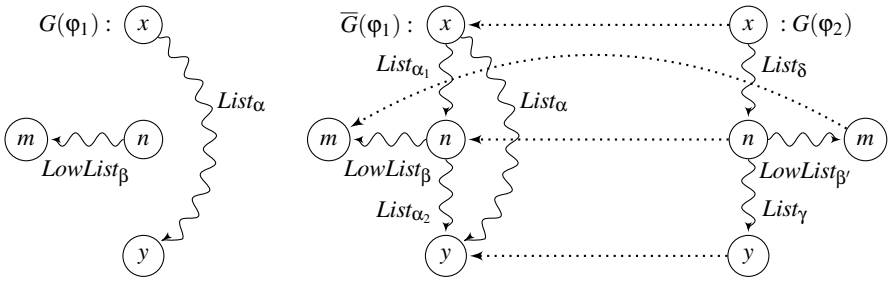


Fig. 3: The graph representations $G(\varphi_1)$ resp. $G(\varphi_2)$ of the (normal forms of the) formulas in eq. (6–7). $\overline{G}(\varphi_1)$ is the graph representation of φ_1 that includes the implicit spatial constraints. Dotted edges represent the homomorphism proving that $\varphi_1 \Rightarrow \varphi_2$.

3 Logic *NOLL*

The logic *NOLL* is a multi-sorted fragment of Separation Logic [17]. Let $\mathcal{T}$ be a set of sorts (corresponding to record types defined in the program), *Flds* a set of field names, and τ a typing function mapping each field name into a function type over $\mathcal{T}$. A field $f \in Flds$ is called recursive iff $\tau(f) = R \rightarrow R$ with $R \in \mathcal{T}$ and non-recursive, otherwise. The set of recursive fields is denoted by $Flds_{rec}$.

Syntax: Let *LVars* and *SetVars* be two sets of variables, called *location variables* and *set of locations variables*, respectively. We assume that the typing function τ associates a sort, resp. a set of sorts, to every variable in *LVars*, resp. *SetVars*. For simplicity, we assume that *LVars* contains the constant `NULL`. The syntax of *NOLL* is given in Fig. 4.

$x, y, y_i \in LVars$	location variables	$\overrightarrow{z} \in LVars^+$	tuples of location variables
$f, f_i \in Flds$	field names	$\alpha \in SetVars$	set of locations variables
$R \in \mathcal{T}$	sort	$P \in \mathcal{P}$	list segment predicates

$$\varphi ::= \Pi \wedge \Sigma \wedge \Lambda \quad \text{NOLL formula}$$
$$\Pi ::= true \mid x \neq y \mid x = y \mid \Pi \wedge \Pi \quad \text{pure constraints}$$
$$\Sigma ::= emp \mid x \mapsto \{(f_1, y_1); \ldots; (f_k, y_k)\} \mid P_\alpha(x, y, \overrightarrow{z}) \mid \Sigma * \Sigma \mid \Sigma *_w \Sigma \quad \text{spatial constraints}$$
$$\Lambda ::= true \mid t \subseteq t' \mid x \in t \mid x \notin t \mid \Lambda \wedge \Lambda \quad \text{sharing constraints}$$
$$t ::= \{x\} \mid \alpha \mid \alpha(R) \mid t \cup t' \quad \text{set of locations terms}$$

Fig. 4: Syntax of *NOLL* formulas.

An atomic *points-to constraint* $x \mapsto \{(f_1, y_1); \ldots; (f_k, y_k)\}$ is used to specify the values of fields $f_1, \ldots, f_k$ in the location denoted by x: the value stored by the field f_i is y_i, for all $1 \leq i \leq k$. The fields shall be pairwise disjoint and the formula shall be well typed, i.e., for any f_i, $\tau(f_i) = \tau(x) \rightarrow \tau(y_i)$.

In every *list segment constraint* $P_\alpha(x, y, \overrightarrow{z})$, P is a predicate from a fixed, but arbitrary, set $\mathcal{P}$. The predicates in $\mathcal{P}$ have recursive definitions with the following syntax:

$$
\begin{array}{rcl}
P(in, out, \overrightarrow{nhb}) & \triangleq & (in = out) \vee \\
& & (\exists u, \overrightarrow{v}. \Sigma_0(in, u \cup \overrightarrow{v} \cup \overrightarrow{nhb}) * \Sigma_1(\overrightarrow{v}, \overrightarrow{nhb}) \ * \ P(u, out, \overrightarrow{nhb})) \\
\Sigma_0(in, V) & ::= & in \mapsto \theta, \text{ where } \theta \subseteq \{(f, w) \mid f \in Flds, w \in V\} \\
\Sigma_1(\overrightarrow{v}, \overrightarrow{nhb}) & ::= & emp \mid Q(v, b, \overrightarrow{b}) \mid \Sigma_1(\overrightarrow{v}, \overrightarrow{nhb}) * \Sigma_1(\overrightarrow{v}, \overrightarrow{nhb}) \text{ with } b, \overrightarrow{b} \subseteq \overrightarrow{nhb}, \text{ and } Q \in \mathcal{P}
\end{array}
$$

where $in, out, u \in LVars$ and $\overrightarrow{nhb}, \overrightarrow{v}, \overrightarrow{b} \in LVars^+$. The definition of every $P \in \mathcal{P}$ is well typed and satisfies the additional typing constraints $\tau(in) = \tau(out) = \tau(u)$, and $\tau(in) \neq \tau(v)$, for every $v \in \overrightarrow{v}$. Moreover, the definitions in $\mathcal{P}$ are not mutually recursive.

A predicate $P(in, out, \overrightarrow{nhb})$ defines possibly empty list segments starting from *in* and ending in *out*. The fields of each element in this list segment are defined by Σ_0 while the nested lists to which it points to are defined by Σ_1. The parameters $\overrightarrow{nhb}$ are used to define the "boundaries" of the nested list segment described by P, in the sense that every location described by P belongs to a path between *in* and some location in $out \cup \overrightarrow{nhb}$ (this path may be defined by more than one field). Every element of the list segment described by P points to several nested lists, each one of them being described by a predicate Q in $\mathcal{P}$. The use of $*$ in the definition of P implies that the inner list segments are disjoint. The typing constraints ensure bounded nesting.

For simplicity of the presentation, we have restricted ourselves to such inductive definitions, which are not expressive enough to describe doubly-linked lists or nested lists containing cyclic lists on their inner levels. However, our techniques can be extended to cover such cases. For example, to describe doubly-linked lists, one must allow further points-to constraints and use a special type of existential variables representing the next to last location in a doubly-linked list segment like, e.g., in [1].

For any predicate P, $\Sigma_0(P)$, resp. $\Sigma_1(P)$, denotes the sub-formula Σ_0, resp. Σ_1 of P. Moreover, $Flds_0(P)$ denotes the set of fields of *in* that point to u according to the formula $\Sigma_0(P)$, i.e., $f \in Flds_0(P)$ iff $\Sigma_0(P) = in \mapsto \theta$ and $(f, u) \in \theta$.

In every spatial constraint $P_\alpha(x, y, \overrightarrow{z})$, α is a set of locations variable, which is said to *be bounded to* or to *index* the spatial constraint. The constraint Λ may contain set of locations variables which are not bounded to some spatial constraint. For simplicity, we assume that a variable in *SetVars* appears in Σ at most once. Also, we consider that all atomic constraints in Λ are well typed, i.e., for any $t \subseteq t'$ in Λ, $\tau(t) \subseteq \tau(t')$ and for any $(x \in t)$ in Λ, $\tau(x) \in \tau(t)$, where τ is extended to set of locations terms as usual.

In the following, we denote by $LVars(\varphi)$ (and $SetVars(\varphi)$) the set of location variables (resp. set of locations variables) used in φ. Also, $atoms(\varphi)$ denotes the set of atomic formulas in φ. Two atoms in Σ are *object separated*, resp. *field separated*, if their least common ancestor in the syntactic tree of φ is $*$, resp. $*_w$.

Semantics: Let *Loc* be a multi-sorted set of *locations* typed by the typing function τ, and let Loc_R denote the set of locations in *Loc* of sort R.

A *program heap* is modeled by a pair $C = (S, H)$, where $S : LVars \to Loc$ maps location variables to locations in *Loc* and $H : Loc \times Flds \rightharpoonup Loc$ defines values of fields for a subset of locations. Intuitively, each allocated object is denoted by a location in *Loc* and then, H defines the fields for the allocated objects and S gives for each variable, the object it points to. The set of locations l for which there exists f s.t. $H(l, f)$ is defined

$(C,J) \models \varphi_1 \wedge \varphi_2$ iff $(C,J) \models \varphi_1$ and $(C,J) \models \varphi_2$

$(C,J) \models x = y$ iff $S(x) = S(y)$

$(C,J) \models x \mapsto \cup_{i\in I}\{(f_i,y_i)\}$ iff $H(S(x),f_i) = S(y_i)$ for all $i \in I$

$(C,J) \models P_\alpha(x,y,\overrightarrow{z})$ iff there exists $k \in \mathbb{N}$ s.t. $(C,J) \models P^k_\alpha(x,y,\overrightarrow{z})$

$(C,J) \models P^0_\alpha(x,y,\overrightarrow{z})$ iff $S(x) = S(y)$ and $J(\alpha) = \emptyset$

$(C,J) \models P^{k+1}_\alpha(x,y,\overrightarrow{z})$ iff $S(x) \neq S(y)$ and there exists $\rho : \{u\} \cup \overrightarrow{v} \to Loc$ and J' s.t.
$(C[S \mapsto S \cup \rho], J') \models \Sigma_0(x, u \cup \overrightarrow{v} \cup \overrightarrow{z}) * \Sigma_1(\overrightarrow{v},\overrightarrow{z}) * P^k_\alpha(u,y,\overrightarrow{z})$,
$\mathtt{img}(\rho) \cap \mathtt{img}(S) = \emptyset$,
$J'(\alpha) = J(\alpha) \setminus (\{S(x)\} \cup \rho(\overrightarrow{v}))$, and $J'(\beta) = J(\beta)$, for any $\beta \neq \alpha$

$(C,J) \models \Sigma_1 * \Sigma_2$ iff there exist program heaps C_1 and C_2 s.t. $C = C_1 * C_2$,
$(C_1,J) \models \Sigma_1$, and $(C_2,J) \models \Sigma_2$

$(C,J) \models \Sigma_1 *_w \Sigma_2$ iff there exist program heaps C_1 and C_2 s.t. $C = C_1 *_w C_2$,
$(C_1,J) \models \Sigma_1$, and $(C_2,J) \models \Sigma_2$

$(C,J) \models x \in t$ iff $S(x) \in [t]_J$

$(C,J) \models t \subseteq t'$ iff $[t]_J \subseteq [t']_J$

Separation operators over program heaps:

$C = C' * C''$ iff $Loc(C) = Loc(C') \cup Loc(C'')$ and $Loc(C') \cap Loc(C'') = \varnothing$,
$S^{C'} = S^C|_{Loc(C')}$ and $S^{C''} = S^C|_{Loc(C'')}$

$C = C' *_w C''$ iff $\mathtt{dom}(H^C) = \mathtt{dom}(H^{C'}) \cup \mathtt{dom}(H^{C''})$ and $\mathtt{dom}(H^{C'}) \cap \mathtt{dom}(H^{C''}) = \varnothing$,
$S^{C'} = S^C|_{Loc(C')}$ and $S^{C''} = S^C|_{Loc(C'')}$

Interpretation of a term t, $[t]_J$:

$[\{x\}]_J = \{S(x)\}$, $[\alpha]_J = J(\alpha)$, $[\alpha(R)]_J = J(\alpha) \cap Loc_R$, $[t \cup t']_J = [t]_J \cup [t']_J$.

Fig. 5: Semantics of *NOLL* formulas. $\mathtt{dom}(F)$ denotes the domain of the function F and $S \cup \rho$ denotes a new mapping $K : \mathtt{dom}(S) \cup \mathtt{dom}(\rho) \to Loc$ s.t. $K(x) = \rho(x)$, $\forall x \in \mathtt{dom}(\rho)$ and $K(y) = S(y)$, $\forall y \in \mathtt{dom}(S)$).

is called the set of locations in C, and denoted by $Loc(C)$. The component S (resp. H) of a heap C is denoted by S^C (resp. H^C).

NOLL interpretations are pairs (C,J), where $C = (S,H)$ is a program heap and $J : SetVars \to 2^{Loc}$ interprets variables in *SetVars* to finite subsets of *Loc*. We assume that S, H, and J are well-typed w.r.t. τ. A *NOLL* interpretation (C,J) is a model of a formula φ iff $(C,J) \models \varphi$, where $\models$ is defined in Fig. 5 for its non trivial cases. For simplicity, we consider the intuitionistic semantics of SL [17]: if a formula is true on a model then it remains true for any extension of that model with more locations. Our techniques can be adapted to work also for the non-intuitionistic semantics [10].

Note the difference between the two kinds of separation of heaps: $C = C' * C''$ holds iff the set of locations in C' and C'' are disjoint while $C = C' *_w C''$ holds iff the domains of the H component in C' and C'' are disjoint.

W.l.o.g., we suppose that the sharing constraints in Λ are in a simplified form obtained as follows. First, inclusion constraints are put in the form $\alpha \subseteq t$, where t contains at most two set of locations variables. Second, for any atomic formula $\alpha \subseteq t$ in Λ such that α is bound to some spatial constraint $P_\alpha(x,y,\overrightarrow{z})$, we remove from t (1) all the variables α' such that α and α' are bound to object separated spatial constraints and (2) all the terms of the form $\{x\}$ such that φ contains a points-to constraint $x \mapsto \theta$, which is

object separated from the spatial constraint indexed by α. If t becomes empty then, the equality $x = y$ is added to φ.

We denote by $[\varphi]$ the set of pairs (C,J) which are models of φ. The entailment between two *NOLL* formulas is denoted by $\Rightarrow$ and it is defined by $\varphi \Rightarrow \psi$ iff $[\varphi] \subseteq [\psi]$.

Fragment *MOLL*: To illustrate some constructions in this paper, we consider the fragment *MOLL* which does not allow to specify nested lists, but only overlaid multi-linked lists. Formally, the fragment *MOLL* contains all the *NOLL* formulas defined over a set of predicates $\mathcal{P}$ such that, for any $P \in \mathcal{P}$, $\Sigma_1(P) = emp$, i.e., P is defined by $P(in, out, \overrightarrow{nhb}) \triangleq (in = out) \vee (\exists u. \Sigma_0(in, u \cup \overrightarrow{nhb}) * P(u, out, \overrightarrow{nhb}))$.

4 A model-theoretic procedure for checking entailment

We prove that satisfiability, resp. entailment checking, of *NOLL* formulas is NP-complete, resp. co-NP complete. The upper bound for the complexity of satisfiability is proved using a small model property: if $\varphi \in NOLL$ has a model, then it has also a model of size polynomial in the size of φ and $\mathcal{P}$ (the size of $\mathcal{P}$ is defined as the size of all recursive definitions for predicates in $\mathcal{P}$). The co-NP upper bound for entailment checking is obtained by proving a small model property for formulas of the form $\varphi \not\Rightarrow \psi$ (a model for this formula corresponds to a counter-example for $\varphi \Rightarrow \psi$).

4.1 Satisfiability problem

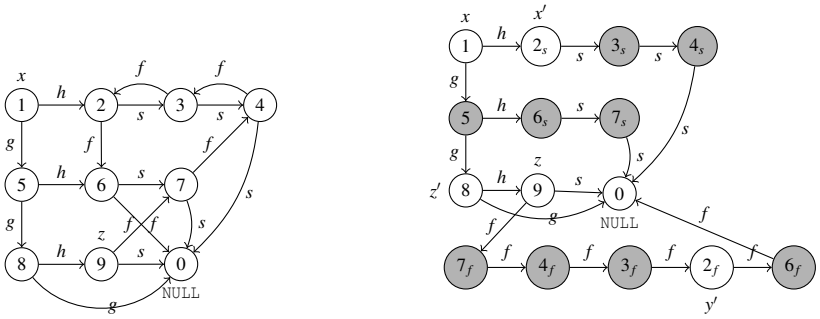
The NP lower bound of the satisfiability problem for *NOLL* formulas is given by the next theorem. The proof is based on a reduction of 3SAT, the satisfiability problem for CNF formulas with three literals in each clause, to the satisfiability problem for *MOLL* formulas. The proof of this result is detailed in [10].

Theorem 1. *The satisfiability problem for NOLL (MOLL) is NP-hard.*

To prove the small model property for the NP upper bound, we use an abstraction of the models of *NOLL* formulas by *colored heap graphs*. Intuitively, a model (C,J) of a *NOLL* formula is represented by a colored graph where each location ℓ from C is represented by a set of graph nodes V_ℓ. V_ℓ is a singleton when ℓ is the interpretation of a location variable or it is not shared between list segments described in φ. Otherwise, each node in V_ℓ represents a subset of fields at location ℓ such that two nodes in V_ℓ represent disjoint sets of fields. All nodes in V_ℓ are colored by ℓ and are called *sibling nodes*. The abstraction is built such that the sub-graphs corresponding to list segments defined using different atoms of φ share only nodes which are interpretations of location variables. Thus, we can collapse in these sub-graphs most of nodes and still obtain a model of φ. The collapsed nodes shall not be colored by the interpretation of a location variable, i.e., they are *anonymous* nodes. We show that for any model (C,J), one can identify a set of anonymous nodes, whose size is polynomial in the size of φ and $\mathcal{P}$, called *crucial nodes*, such that by collapsing all the non-crucial anonymous nodes one can still obtain a model of φ. Formally,

Definition 1 (Colored heap graph). *A* colored heap graph *over LVars, Flds, and SetVars is a tuple* $G = (V, E, \mathcal{P}, \mathcal{L}, \mathcal{S})$*, where (1)* V *is a finite set of nodes, (2)* $E : V \times Flds \rightharpoonup V$ *is a set of edges, (3)* $\mathcal{P} : LVars(\varphi) \to V$ *is a labeling of nodes with location variables, (4)* $\mathcal{L} : V \to Loc$ *is a coloring of nodes with locations, and (5)* $\mathcal{S} : SetVars \to 2^V$ *is an interpretation of variables in SetVars to sets of nodes.*

Fig. 6 pictures a model of φ in eq. (1) and its colored heap graph abstraction. We denote the components of a colored heap graph G using superscripts, e.g., the set V in G is denoted by V^G. The semantics of *NOLL* formulas on colored heap graphs is defined similarly to the one on *NOLL* interpretations, except for $*$ and the constraints in Λ. A colored heap graph G satisfies a formula $\varphi_1 * \varphi_2$ iff G can be split into two disjoint graphs G_1 and G_2 such that $G_1 \models \varphi_1$, $G_2 \models \varphi_2$, and for any two nodes $v_1 \in V^{G_1}$ and $v_2 \in V^{G_2}$, $\mathcal{L}^{G_1}(v_1) \neq \mathcal{L}^{G_2}(v_2)$. Also, for any constraint $P_\alpha(x, y, \overrightarrow{z})$, $\mathcal{S}(\alpha)$ is interpreted as the union of $\mathcal{L}(v)$, for all nodes v in the unique subgraph defined by P_α.



(a) (b)

Fig. 6: A program heap satisfying φ in (1) and its colored heap graph. For any $0 \leq n \leq 9$, the nodes n_s and n_f in (b) are colored by the location n from (a). Primed variables x', y', z' label crucial nodes. A small model is obtained by collapsing filled nodes in (b).

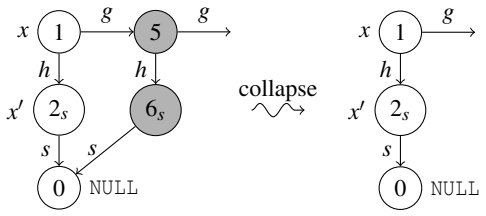
Lemma 1. *If a NOLL formula* φ *has a model* (C, J) *then it also has a model* (C_s, J_s) *of size polynomial in the size of* φ *and* $\mathcal{P}$*.*

Proof. (Idea) The proof builds a small model following the steps given in Fig. 7a. Roughly, we show that anonymous locations from (C, J) can be collapsed until the list segments are of bounded length. The bounds are determined by the sharing constraints in φ and the levels of nesting in the definition of the recursive predicates. To collapse anonymous locations on list segments, we use the colored heap graph abstraction. However, some distinguished set of *crucial* anonymous nodes shall not be collapsed because this will invalidate spatial or sharing constraints in φ (an example is shown below). Also, to preserve the truth value of sharing constraints, if a node is found crucial on some list segment, then all its sibling nodes are also marked as crucial (this corresponds to the fact that the small model contains all the fields for that location).

The procedure purify removes from (C,J) all the locations not involved in spatial constraints from φ. This is possible because the minimal part of C satisfying some spatial constraint is unique. splitLocations builds the colored heap graph abstraction of (C',J') by splitting the nodes not labeled by location variables but shared between several list segments described by predicates in φ. An example is given in Fig. 6.

1: $(C',J') := \mathsf{purify}(\varphi)(C,J)$
2: $G := \mathsf{splitLocations}(C',J')$
3: $V' := \mathsf{crucialNodes}(\varphi, G)$
4: $G' := \mathsf{labelCrucial}(G,V')$
5: $G'' := \mathsf{collapseAnonymous}(G')$
6: $(C_s,J_s) := \mathsf{mergeNodes}(G'')$

(a) Steps for computing a small model.



(b) Example of collapsing.

Fig. 7: Computing a small model for *NOLL* formulas.

crucialNodes computes the set of crucial nodes V' as the closure under the sibling relation of the set of (anonymous) nodes in G which are either (1) the successor of a labeled node by a non recursive field (e.g., node 2_s in Fig. 6), or (2) the source or the target of a non recursive field on a fixed path between two nodes labeled by location variables (e.g., node 8 in Fig. 6). Because the nesting of recursive predicates is bounded, the size of the set V' is bounded by a polynomial in the size of φ and $\mathcal{P}$ (the number of variables, the nesting depth, and the size of *Flds*). The crucial nodes are labeled with a set of additional location variables $LVars'$ in labelCrucial.

Afterwards, the anonymous nodes (not labeled by variables in $LVars(\varphi) \cup LVars'$) are collapsed by collapseAnonymous in a bottom up manner, i.e., starting from the inner list segments to the upper ones. Roughly, the collapsing removes a node (and the sub-graph representing the nested, anonymous structure) if it is between two recursive fields (see Fig. 7b). Intuitively, this process preserves a model of φ because no edges are added and the nodes marked as important for the satisfaction of the spatial and sharing constraints are kept. Due to the special syntax of predicates in $\mathcal{P}$, we can compute for each list segment the minimal number of anonymous nodes that must be preserved in order to satisfy some given spatial constraint. This number depends only on the size of $\mathcal{P}$ and it is obtained when all the spatial constraints in the predicate definition are interpreted as list segments of length one. Thus, we obtain a colored heap graph G'' where all labeled nodes are preserved and with them some sub-graphs with a bounded number of anonymous nodes. Finally, from G'', a (small) model (C_s,J_s) of φ is built, by applying mergeNodes, which roughly merges sibling nodes in locations. □

Since the complexity of the model-checking problem for *NOLL* formulas is polynomial, the following result holds.

Theorem 2. *The satisfiability problem for NOLL is NP-complete.*

4.2 Entailment problem

The colored heap graph abstraction is also used to prove a small counter-example property for entailments $\varphi \Rightarrow \psi$ when φ and ψ are in *NOLL*. The proof is similar to the proof of Lemma 1, with two main differences. Let (C,J) be a counter-example for $\varphi \Rightarrow \psi$. First, in purify, the locations not used in φ are removed from (C,J) except for locations that are witnesses for some unsatisfied sharing constraint in ψ. It is enough to keep one location per sharing constraint in ψ and thus, their number is bounded by the size of ψ. We label these locations with variables from some set $LVars''$. Second, crucialNodes marks some additional nodes as crucial, in order to keep track if two list segments are sharing at least one location and in order to distinguish between list segments of size 1 and list segments of size at least 2. However, this process adds at most one more node per constraint, and thus the bound on the number of nodes is increased by a linear term in the size of φ and ψ. This property and the NP-completeness of satisfiability imply:

Theorem 3. *Checking the validity of an entailment between two NOLL formulas is co-NP complete.*

5 Computing the normal form

This section makes a first step towards the effective procedure for checking entailments of *NOLL* formulas by presenting the procedure for computing the normal form of a *NOLL* formula. We say that a *NOLL* formula is in *normal form* if it contains the maximum set of equalities and disequalities between location variables and the minimum set of list segment constraints. Formally,

Definition 2 (Normal form). *A NOLL formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$ is in normal form iff:*

- *for any $x,y \in LVars(\varphi)$, if $\varphi \Rightarrow x = y$, resp. $\varphi \Rightarrow x \neq y$, then Π contains the atom $x = y$, resp. $x \neq y$, and*
- *for any atomic formula $P_\alpha(x,y,\overrightarrow{z})$ in Σ, there exists a model (C,J) of φ such that $S^C(x) \neq S^C(y)$.*

The normal form *of φ is a formula φ' in normal form and equivalent to φ.*

We now describe the main ideas behind the procedure that computes the normal form and to this, we must define the class of reduced, explicit *NOLL* formulas.

A *NOLL* formula is called *explicit* if it contains $x = y$ or $x \neq y$, for any constraint $P_\alpha(x,y,\overrightarrow{z})$ in φ, and $x \in \alpha$ or $x \notin \alpha$, for any x and α in φ. Then, an explicit formula ψ is called *reduced* if it does not contain both the atoms $x = y$ and $P_\alpha(x,y,\overrightarrow{z})$.

Any *NOLL* formula φ is equivalent to a disjunction of reduced, explicit formulas $\psi_1 \vee \ldots \vee \psi_n$. The formulas ψ_i are obtained from φ by (1) adding in all possible ways atoms $x = y$, $x \neq y$, $x \in \alpha$, and $x \notin \alpha$ until the obtained formula is explicit and then, (2) if a formula contains $x = y$, by removing atoms $P_\alpha(x,y,\overrightarrow{z})$ together with all occurrences of α in the sharing constraints (e.g., every atom $x \in \alpha$ or $\beta \subseteq \alpha$, where β indexes a constraint $Q_\beta(u,v,\overrightarrow{w})$ and $u \neq v$ belongs to the formula, is replaced by *false*).

The equivalent formula $\psi_1 \vee \ldots \vee \psi_n$ can be used to compute the normal form of φ as follows. An atom $x = y$ or $x \neq y$ is implied by φ iff this atom is included in all the

satisfiable formulas ψ_i. Also, for any $P(x,y,\overrightarrow{z})$ in φ, there exists a model (C,J) of φ s.t. $S^C(x) \neq S^C(y)$ iff this atom is included in some satisfiable ψ_i.

In general, the number of satisfiable formulas in the disjunction $\psi_1 \vee \ldots \vee \psi_n$ may be exponential w.r.t. the size of φ. However, all these formulas can be represented symbolically as the satisfying assignments of a boolean formula, denoted by $F(\varphi)$.

In order to simplify the presentation, we give below the construction of $F(\varphi)$ only for *MOLL* formulas where variables are of the same type; [10] gives the general case. $F(\varphi)$ is defined over the set of boolean variables $BVars(F(\varphi))$ defined in Tab. 1.

Table 1: Definition of the set $BVars(F(\varphi))$ of boolean variables used in $F(\varphi)$.

$[x=y]$	for every $x,y \in LVars(\varphi)$
$[x,y,f]$	for every atom $x \mapsto \theta$ of φ with $(f,y) \in \theta$
$[P_\alpha(x,y,\overrightarrow{z})]$	for every atom $P_\alpha(x,y,\overrightarrow{z})$ of φ
$[x \in \alpha]$	for every $x \in LVars(\varphi)$ and $\alpha \in SetVars(\varphi)$

Given a satisfying assignment $\sigma : BVars(F(\varphi)) \to \{0,1\}$ for $F(\varphi)$ such that $\sigma([x,y,f]) = 1$, for any $[x,y,f] \in BVars(F(\varphi))$, we define the *MOLL* formula ψ_σ to be φ to which the following transformations are applied:

- if $\sigma([x=y])$ is 0, resp. 1, then ψ_σ includes the pure constraint $x \neq y$, resp. $x = y$,
- if $\sigma([P_\alpha(x,y,\overrightarrow{z})]) = 0$ then $P_\alpha(x,y,\overrightarrow{z})$ and α are removed from φ,
- if $\sigma([x \in \alpha])$ is 0, resp. 1, then $x \notin \alpha$, resp. $x \in \alpha$, is added to ψ_σ.

Let $\varphi = \Pi \wedge \Sigma \wedge \Lambda$ be a *MOLL* formula. The formula $F(\varphi)$ is defined by:

$$F(\varphi) = F(\Pi) \wedge F_{eq} \wedge F(\Sigma) \wedge F_{det} \wedge F(\Lambda) \wedge F_{\in}, \tag{8}$$

where $F(\Pi)$, $F(\Sigma)$, and $F(\Lambda)$ encode the semantics of the atomic formulas of φ, F_{eq} encodes the reflexivity and the transitivity of the equality relation in Π, F_{det} encodes the semantics of the field separating conjunction, and $F_\in$ encodes the properties of the membership relation $\in$. These sub-formulas are defined inductively on the syntax of *MOLL* formulas. Most of them are not difficult to follow. We provide here some intuition for the most interesting ones.

In $F(\Sigma)$, an atom $P_\alpha(x,y,\overrightarrow{z})$ is translated into $F(P_\alpha(x,y,\overrightarrow{z})) = [P_\alpha(x,y,\overrightarrow{z})] \oplus [x = y]$, where $\oplus$ is the exclusive or. This expresses the fact that the atom is kept in a reduced, explicit *MOLL* formula only if its endpoints are not equal.

The separation of fields (defined for locations which are interpretations of location variables) induced by the use of the field separating conjunction is expressed in the formula F_{det} in Fig. 8. Thus, F_{det} states that for any location variable x and any field $f \in Flds$, at most one of the following conditions is true:

1. the reduced, explicit formula contains the equality $x = x'$ and a points-to constraint $x' \mapsto \theta$ such that $(f,y) \in \theta$, for some y,
2. the reduced, explicit formula contains the atoms $x \in \alpha$ and $P_\alpha(x',y,\overrightarrow{z})$ (therefore it also includes $x' \neq y$), for some y and $\overrightarrow{z}$, such that $f \in Flds_0(P_\alpha)$.

$$
\begin{aligned}
F_{det} = & \bigwedge \text{ for any } [x_1,y_1,f],[x_2,y_2,f] \in \mathit{BVars}(F(\varphi)) \text{ different variables} \\
& [x_1 = x_2] \wedge [x_1,y_1,f] \;\Rightarrow\; \neg[x_2,y_2,f] && (9) \\
& \bigwedge \text{ for any } [x_1,y_1,f],[P_\alpha(x_2,y_2,\overrightarrow{z_2})] \in \mathit{BVars}(F(\varphi)) \text{ s.t. } f \in \mathit{Flds}_0(P) \text{ and } x \in \mathit{LVars}(\varphi) \\
& [x_1 = x] \wedge [x \in \alpha] \wedge [x_1,y_1,f] \;\Rightarrow\; \neg[P_\alpha(x_2,y_2,\overrightarrow{z_2})] && (10) \\
& \bigwedge \text{ for any } [P_\alpha(x_1,y_1,\overrightarrow{z_1})],[Q_\beta(x_2,y_2,\overrightarrow{z_2})] \in \mathit{BVars}(F(\varphi)) \text{ different variables} \\
& \text{s.t. } \mathit{Flds}_0(P) \cap \mathit{Flds}_0(Q) \neq \emptyset \text{ and } x,x' \in \mathit{LVars}(\varphi) \\
& [x \in \alpha] \wedge [x' \in \beta] \wedge [x = x'] \wedge [P_\alpha(x_1,y_1,\overrightarrow{z_1})] \;\Rightarrow\; \neg[Q_\beta(x_2,y_2,\overrightarrow{z_2})] && (11)
\end{aligned}
$$

Fig. 8: Definition of F_{det} for a *MOLL* formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$.

Fig. 9 gives the main definitions of $F(\Lambda)$. For instance, $F(\alpha_1 \subseteq \alpha_2)$ in eq. (14) expresses the fact that if there exists some variable x such that $x \in \alpha_1$ is true then $x \in \alpha_2$ also holds. In eq. (15), $F_\in$ encodes the closure of $\in$ under the equality, the fact that if a boolean variable $[x_1 \in \alpha]$ is true then the list segment bound to α in φ, if any, is not empty, and if α is bound to a non-empty list segment $P_\alpha(x,y,\overrightarrow{z})$ in φ, then α contains the first element of the segment, i.e., x.

$$
\begin{aligned}
F(x \in \alpha_1) &= [x \in \alpha_1] && (12) \\
F(x \in \bigcup_{1 \le i \le n} \{u_i\}) &= \bigvee_{1 \le i \le n} [x = u_i] && (13) \\
F(\alpha_1 \subseteq \alpha_2) &= \bigwedge_{x \in \mathit{LVars}(\varphi)} [x \in \alpha_1] \Rightarrow [x \in \alpha_2] && (14) \\
F_\in &= \bigwedge_{u,v,\alpha \text{ in } \varphi} \big([u = v] \wedge [u \in \alpha]\big) \Rightarrow [v \in \alpha] && (15) \\
&\quad \wedge \bigwedge_{x_1,P_\alpha(x,y,\overrightarrow{z}) \text{ in } \varphi} \big([x_1 \in \alpha] \Rightarrow [P_\alpha(x,y,\overrightarrow{z})]\big) \quad \wedge \quad \big([P_\alpha(x,y,\overrightarrow{z})] \Rightarrow [x \in \alpha]\big)
\end{aligned}
$$

Fig. 9: Main definitions of $F(\Lambda)$ and $F_\in$ for a *MOLL* formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$.

Proposition 1. *The size of $F(\varphi)$ is polynomial in the size of φ.*

Proposition 2. *Let φ be a NOLL formula. For any satisfying assignment σ of $F(\varphi)$, ψ_σ is an explicit, reduced, and satisfiable formula. Also, φ is equivalent to the disjunction of ψ_σ, for all satisfying assignments σ of $F(\varphi)$.*

Theorem 4. *The problem of computing the normal form of a formula φ is in co-NP.*

Proof. To compute the maximum set of (in)equalities that should be included in the normal form of φ, we iterate over every pair of location variables x, y in φ and check if $F(\varphi) \Rightarrow [x = y]$ or $F(\varphi) \Rightarrow \neg[x = y]$ is valid. In the first (resp., second) case, $x = y$ (resp., $x \neq y$) is included in the normal form. When some equality $x = y$ is added to the normal form, the atoms $P_\alpha(x,y,\overrightarrow{z})$ in φ are removed, and all occurrences of α are interpreted as the empty set. Since we need to perform a polynomial number of Boolean formula validity tests, the overall complexity of this procedure is co-*NP* time. □

6 An effective procedure for checking entailment

The procedure for checking the validity of the entailments $\varphi \Rightarrow \psi$ between two *NOLL* formulas is detailed in Fig. 10. It has three main steps: (a) compute (lines 1–2) the normal form of φ and ψ, denoted by φ' and ψ', respectively, (b) compute (line 3) additional spatial constraints, which are implied by φ, and (c) check (lines 3–6) if the graph representation of ψ' is *homomorphic* to the graph representation of both φ' and the additional constraints computed in the previous step.

```
procedure CheckEntl(φ ⇒ ψ)
1: φ' := the normal form of φ
2: ψ' := the normal form of ψ
3: G1 := the complete NOLL graph of φ'
4: G2 := the NOLL graph of ψ'
5: h := the function h : V(G2) ⇀ V(G1) s.t.
       vars_G2(n) ⊆ vars_G1(h(n)), ∀n ∈ V(G2)
6: return (h is total) and
          (h is a homomorphism)
```

Fig. 10

In the following, we first describe the step (b) above, then we define graph representations for *NOLL* formulas, called (complete) *NOLL* graphs, and finally, we define the notion of homomorphism between *NOLL* graphs. *Moreover, we assume that* φ *and* ψ *are satisfiable.* Otherwise, Proposition 2 implies that a formula φ is satisfiable iff $F(\varphi)$ is satisfiable, which allows to decide in co-NP time entailments of the form $\varphi \Rightarrow \psi$ when φ or ψ is unsatisfiable.

6.1 Inferring additional spatial constraints

In order to give an intuition about the additional spatial constraints deduced from φ, recall the entailment $\varphi_1 \Rightarrow \varphi_2$, where φ_1 and φ_2 are defined in eq. (6–7) at page 6. The entailment holds because the list segments linking x to n and n to y, and described by $List_\delta(x,n) * List_\gamma(n,y)$, exist in every model of φ_1. To obtain a complete decision procedure for entailment, such constraints must be made explicit before checking the existence of a homomorphism between the two formulas viewed as graphs.

Observe that φ_1 does not imply $\varphi_1 *_w \big(List_\delta(x,n) * List_\gamma(n,y)\big)$ but, $\varphi_1 \wedge \big(List_\delta(x,n) * List_\gamma(n,y)\big)$. Thus, these implicit constraints will be added only to the graph representation of *NOLL* formulas and not to the formula itself, as explained in Sec. 6.2.

For simplicity, we give the definition only for *MOLL* formulas φ. Let ξ be a set of atoms in φ of the form $Q_\beta(u, v, \overrightarrow{w})$. For any such ξ, $\mathcal{P}(\xi)$ denotes the set of recursive predicates in ξ, $SetVars(\xi)$ denotes the set of variables $\beta \in SetVars$ bounded to atoms in ξ, and t_ξ is the term defined as the union of all variables in $SetVars(\xi)$.

An atom $P_\alpha(x, y, \overrightarrow{z})$ is called *implicit in* ξ iff one of the following holds:

- ξ consists of one atom $P_\beta(u, v, \overrightarrow{z})$, the source of P_α is the same as the source of P_β, i.e., $\varphi \Rightarrow x = u$, and the destination of P_α is included in the list segment defined by P_β, i.e., $\varphi \Rightarrow y \in \beta$;
- (1) $\varphi \Rightarrow x \in t_\xi$, (2) t_ξ is a minimal term t such that $\varphi \Rightarrow x \in t$, i.e., for every other term t', which is the union of the variables from a strict subset of $SetVars(\xi)$, $\varphi \not\Rightarrow x \in t'$, (3) $Flds_0(P) = \bigcap_{Q \in \mathcal{P}(\xi)} Flds_0(Q)$, and (4) $\varphi \Rightarrow \bigwedge_{Q_\beta(u,v,\overrightarrow{z}) \in \xi} y = v$.

Similarly, an atom $x \mapsto \{(f,y)\}$ is called implicit in ξ iff the conditions (1) and (2) above hold, (3′) an atom $u \mapsto \theta_i$ with $(f,d_i) \in \theta_i$ is included in the definition of Q, for all $Q \in \mathcal{P}(\xi)$, and (4′) $\varphi \Rightarrow \bigwedge_{1 \leq i \leq n} y = d_i$.

For example, for $\xi = \{List_\alpha(x,y)\}$ a set of atoms in φ_1 from eq. (6), the atom $List_\delta(x,n)$ is implicit in ξ because $\beta \subseteq \alpha$ in φ_1 implies that $n \in \alpha$ and the equality $x = x$ is trivially implied by φ_1. Also, the atom $List_\gamma(n,y)$ is implicit in ξ because the conditions (1–4) above hold.

By definition, the Boolean abstraction $F(\varphi)$ defined in Sec. 5 can be used to check that φ implies the equalities and the sharing constraints in the above conditions. The conditions (3) and (3′) can be checked syntactically. Thus, the computation of the implicit spatial constraints for a formula is co-NP complete.

6.2 *NOLL* graphs

We define *NOLL graphs*, a graph representation for *NOLL* formulas. Roughly, the nodes of these graphs represent sets of equal location variables and the edges represent spatial or difference constraints. The object separated spatial constraints are represented by a binary relation Ω_* over edges while the sharing constraints are kept unchanged.

Definition 3 (*NOLL* **graph).** *Given a NOLL formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$ over a set of predicates $\mathcal{P}$, the NOLL* graph *of φ, denoted $G(\varphi)$, is a tuple $(V, E_P, E_R, E_D, \ell, \Omega_*, \Lambda)$ or the error graph $\perp$, where:*

- *each node in V denotes an equivalence class over elements of LVars w.r.t. the equality relation defined in Π; the equivalence class of x is denoted by $[x]$. If Π contains both $x \neq y$ and $x = y$ then G is the error graph $\perp$;*
- *$E_P \subseteq V \times Flds \times V$ represents the points-to constraints: $([x], f, [y]) \in E_P$ iff $x \mapsto \theta$ with $(f,y) \in \theta$ is an atomic formula in Σ;*
- *$E_R \subseteq V \times \mathcal{P} \times V^+ \times V$ represents list segment constraints: $([x], P_\alpha, [\overrightarrow{z}], [y]) \in E_R$ iff $P_\alpha(x, y, \overrightarrow{z})$ is an atomic formula in Σ;*
- *$E_D \subseteq V \times V$ represents inequalities: $([x],[y]) \in E_D$ iff $x \neq y$ is an atom in Π;*
- *$\ell : LVars \to V$, called* variable labeling*, it is defined by $\ell(x) = [x]$, for any $x \in LVars$;*
- *Ω_* contains all pairs of edges in $E_P \cup E_R$ denoting object separated atoms in Σ.*

In the following, $V(G)$, denotes the set of nodes in the *NOLL* graph G. We use a similar notation for all the other components of G. Also, for any $n \in V(G)$, $vars_G(n)$ denotes the set of all the variables labeling the node n in G. The graph $G(\varphi_2)$ in Fig. 3 represents the *NOLL* graph of φ_2, where $V = \{x,y,n,m\}$, $E_P = E_D = \emptyset$, E_R contains the three edges corresponding to the three list segments, Ω_* contains only one pair $\langle([x], List_\alpha, [n]), ([n], List_{\beta'}, [y])\rangle$, and Λ is $\beta' \subseteq \delta \cup \gamma$.

A graph representation for φ which includes an edge for each implicit spatial constraint of φ is called a *complete NOLL graph*. This representation has an additional attribute Δ, which identifies the set of atoms where a spatial constraint is implicit in.

Definition 4 (complete *NOLL* **graph).** *Given a NOLL formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$, the* complete *NOLL* graph *of φ, denoted by $\overline{G}(\varphi)$ is a tuple (G, Δ) where:*

- *G is a NOLL graph where all components except E_R, E_P, Ω_*, and Λ are equal to the components of $G(\varphi)$;*
- *$E_R(G)$ (resp. $E_P(G)$) includes $E_R(G(\varphi))$ (resp. $E_P(G(\varphi))$) and, for any atom $P_\alpha(x,y,\overrightarrow{z})$ (resp. $x \mapsto \{(f,y)\}$) which is implicit in some set of atoms ξ, $e = ([x],P_\alpha,[\overrightarrow{z}],[y]) \in E_R(G)$ (resp. $e = ([x],f,[y]) \in E_P(G)$);*
- *$\Omega_*(G)$ consists of $\Omega_*(G(\varphi))$ plus all pairs (e,e') s.t. e represents an implicit constraint in ξ and $(e',e'') \in \Omega_*(G)$ for some e'' representing an atom in ξ;*
- *$\Delta \subseteq (E_P \cup E_R) \times 2^{E_R}$ represents the relation between edges and the sets of list segments where they are implicit in, i.e., for every $P_\alpha(x,y,\overrightarrow{z})$ (resp. $x \mapsto \{(f,y)\}$) implicit in ξ, $(([x],P_\alpha,[\overrightarrow{z}],[y]),E_\xi) \in \Delta$ (resp. $(([x],f,[y]),E_\xi) \in \Delta$), where E_ξ is the set of edges representing the atoms in ξ;*
- *if $P_{\alpha_1}(x,y,\overrightarrow{z})$ and $P_{\alpha_2}(y,t,\overrightarrow{z})$ are implicit in $\xi = \{P_\alpha(x,t,\overrightarrow{z})\}$ then, $\alpha = \alpha_1 \cup \alpha_2$ is added to Λ.*

The graph in the middle of Fig. 3 represents the complete *NOLL* graph of φ_1, $\overline{G}(\varphi_1)$, where $V = \{x,y,n,m\}$, $E_P = E_D = \Omega_* = \emptyset$, and E_P contains the four edges: two edges represent the spatial constraints in φ_1, and the edges $([x],List_{\alpha_1},[n])$ and $([n],List_{\alpha_2},[m])$ represent implicit constraints in $\xi = \{List_\alpha(x,y)\}$. Λ is $\beta \subseteq \alpha \wedge \alpha = \alpha_1 \cup \alpha_2$ and Δ is the relation $\{(([x],List_{\alpha_1},[n]),\xi),(([n],List_{\alpha_2},[m]),\xi)\}$.

6.3 *NOLL* graph homomorphism

Given a *NOLL* graph G_1 and a complete *NOLL* graph G_2, a *homomorphism from G_1 to G_2* is a mapping $h : V(G_1) \mapsto V(G_2)$, which:

1. preserves the labeling with location variables, i.e., $vars_{G_1}(n) \subseteq vars_{G_2}(h(n))$, for any $n \in V(G_1)$,
2. maps each difference, resp., points-to, edge of G_1 to a difference, resp., points-to, edge of G_2, (e.g., for any $(n,f,n') \in E_P(G_1)$, $(h(n),f,h(n')) \in E_P(G_2)$),
3. maps each edge representing a list segment in G_1 to a path in G_2 formed of edges in $E_P(G_2) \cup E_R(G_2)$, and
4. satisfies the constraints required by the semantics of the separating conjunctions, the special status of the implicit spatial constraints, and the sharing constraints.

To explain the mapping of edges in $E_R(G_1)$ to paths of G_2, let us consider the case of an edge $(n,P_\alpha,m,p) \in E_R(G_1)$, where $n,m,p \in V(G_1)$ and P is a *MOLL* predicate, i.e., $P(in,out,b) \triangleq (in = out) \vee (\exists u.\, \Sigma_0(in,u \cup b) * P(u,out,b))$. The definition of h requires that there exists a sequence of nodes $\pi = \pi_1 \ldots \pi_k$, $k \geq 1$, in G_2 s.t. $\pi_1 = h(n)$, $\pi_k = h(p)$, and for every two consecutive nodes π_i and π_{i+1}, either

- $E_P(G_2)$ contains some set of edges between π_i, π_{i+1}, and $h(m)$, which prove that $\Sigma_0(x_i,x_{i+1} \cup x_{h(m)})$ holds, where x_i, x_{i+1}, and $x_{h(m)}$ are some variables labeling π_i, π_{i+1}, and $h(m)$, respectively, or
- there exists an edge $(\pi_i,P'_\beta,\overrightarrow{q},\pi_{i+1})$ in $E_R(G_2)$, representing a stronger predicate than P_α, i.e., $h(m) \in \overrightarrow{q}$ and $P'_\beta(x_i,x_{i+1},\overrightarrow{z}) \Rightarrow P_\alpha(x_i,x_{i+1},x_{h(m)})$, where x_i, x_{i+1}, and $x_{h(m)}$ are as above, and $\overrightarrow{z}$ is a set of variables labeling $\overrightarrow{q}$ s.t. $x_{h(m)} \in \overrightarrow{z}$ (this is possible because $h(m) \in \overrightarrow{q}$). The entailment between recursive predicates can be checked syntactically in polynomial time.

In the following, we explain the constraints required by the 4^{th} item in the definition of the homomorphism. For any edge e in $E_P(G_1) \cup E_R(G_1)$, we define a set $used(e) \subseteq E_P(G_2) \cup 2^{(E_R(G_2) \times Flds)}$, which represents all the edges/fields used in the path from G_2 to which e is mapped by h. If $e \in E_P(G_1)$ then $used(e) = \{e'\}$, where e' is the edge of G_2 to which e is mapped by h. If $e \in E_R(G_1)$ represents a list segment P_α then, $used(e)$ consists of (1) the set of points-to edges in the path associated to e and (2) the set of pairs (e', f), where e' represents a list segment Q_β from the same path, if such an edge exists, and $f \in Flds_0(P) \cap Flds_0(Q)$. When the path associated to $e \in E_R(G_1)$ labeled by P_α (resp. $e \in E_P(G_1)$ labeled by f) contains an edge e' representing a constraint implicit in some ξ, i.e., $(e', E_\xi) \in \Delta(G_2)$, then $used(e)$ includes all pairs (e'', f) with $e'' \in E_\xi$ labeled by $Q_\beta \in \xi$, and $f \in Flds_0(P) \cap Flds_0(Q)$ (resp. $f \in Flds_0(Q)$).

Then, to express the semantics of $*_w$, we require that $used(e_1) \cap used(e_2) = \emptyset$, for any two edges e_1 and e_2 in $E_P(G_1) \cup E_R(G_1)$. Concerning $*$, it is required that for any two edges e_1 and e_2 in $E_P(G_1) \cup E_R(G_1)$ s.t. $(e_1, e_2) \in \Omega_*(G_1)$, we have that $(e'_1, e'_2) \in \Omega_*(G_2)$, for any e'_1 an edge appearing in $used(e_1)$ and e'_2 an edge appearing in $used(e_2)$.

Finally, for the sharing constraints, the mapping by h of edges in $E_R(G_1)$ to paths in G_2 defines a substitution Γ for set of locations variables in $\Lambda(G_1)$ to terms over set of locations variables in $\Lambda(G_2)$. For example, the homomorphism in Fig. 3 defines the substitution $\Gamma(\delta) = \alpha_1$, $\Gamma(\gamma) = \alpha_2$, and $\Gamma(\beta') = \beta$. Then, it is required that $\Lambda(G_2) \Rightarrow \Lambda(G_1)[\Gamma]$. Such a formula belongs for instance, to the fragment of BAPA [14], and thus its validity can be decided in NP-time. For the example in Fig. 3, we obtain the trivial entailment $\beta \subseteq \alpha \wedge \alpha = \alpha_1 \cup \alpha_2 \;\Rightarrow\; \beta \subseteq \alpha_1 \cup \alpha_2$.

6.4 Checking entailments of *NOLL* formulas

The following theorem states the correctness and the complexity of the procedure `CheckEntl` given in Fig. 10; the proof is given in [10].

Theorem 5. *Given two NOLL formulas* φ *and* ψ, $\varphi \Rightarrow \psi$ *holds iff* `CheckEntl`$(\varphi \Rightarrow \psi)$ *returns* `true`. *Moreover, the complexity of* `CheckEntl` *is co-NP time.*

7 Experimental results

We have implemented the procedure for entailment checking in a solver which takes as input the specification of predicates in $\mathcal{P}$ and two formulas $\varphi, \psi \in NOLL$ defined over $\mathcal{P}$ and returns as result either the homomorphism found when $\varphi \Rightarrow \psi$ or a diagnosis explaining why the entailment is not valid. The diagnosis is given as a list of variables or atomic spatial constraints in φ and ψ for which the conditions for the homomorphism are not satisfied. The solver is implemented in C. It uses MiniSat [9] to compute normal forms and an ad-hoc solver for the sharing constraints.

We have used this solver to check verification conditions generated for procedures working on singly linked lists, doubly linked lists, and overlaid hash tables and lists in the Nagios network monitoring example. We have considered mainly the procedures for inserting or moving elements in these data structures. The post-condition computation follows the standard approach: introducing primed variables to denote old values and

unfolding recursive predicates for statements that involve fields. To generate simpler verification conditions, we use the frame rules for the separating conjunction operators. In this way, the graph representations for the *NOLL* formulas have less than ten vertices and twenty edges (including the inferred edges), and less than five set of locations variables. Each verification condition is decided in less than 0.1 seconds.

References

1. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.
2. J. Berdine, C. Calcagno, and P.W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, volume 3328 of *LNCS*, pages 97–109. Springer, 2004.
3. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *LNCS*, pages 115–137. Springer, 2005.
4. F. Bobot and J.C. Filliâtre. Separation predicates: a taste of separation logic in first-order logic. In *ICFEM*, volume 7635 of *LNCS*, pages 167–181. Springer, 2012.
5. C. Calcagno, H. Yang, and P.W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS*, volume 2245 of *LNCS*, pages 108–119, 2001.
6. B.-Y.E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260. ACM, 2008.
7. A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245. ACM, 2011.
8. B. Cook, C. Haase, J. Ouaknine, M. J. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, volume 6901 of *LNCS*, pages 235–249, 2011.
9. N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
10. C. Enea, V. Saveluc, and M. Sighireanu. Composite invariant checking for nested, overlaid linked lists, 2012. Extended version available as HAL-00768389 report.
11. P. Hawkins, A. Aiken, K. Fisher, M.C. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI*, pages 38–49. ACM, 2011.
12. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26. ACM, 2001.
13. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, volume 6461 of *LNCS*, pages 304–311. Springer, 2010.
14. V. Kuncak, H.H. Nguyen, and M.C. Rinard. An algorithm for deciding BAPA: Boolean algebra with presburger arithmetic. In *CADE*, volume 3632 of *LNCS*, pages 260–277. Springer, 2005.
15. P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *POPL*, pages 123–136. ACM, 2012.
16. J.A. Navarro Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566. ACM, 2011.
17. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
18. P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, pages 199–210. ACM, 2010.
19. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, volume 6901 of *LNCS*, pages 385–398. Springer, 2008.